Breaking EvilQuest | Reversing A Custom macOS Ransomware File Encryption Routine

labs.sentinelone.com/breaking-evilquest-reversing-a-custom-macos-ransomware-file-encryption-routine/

Jason Reaves

Executive Summary

- A new macOS ransomware threat uses a custom file encryption routine
- The routine appears to be partly based on RC2 rather than public key encryption
- SentinelLabs has released a public decryptor for use with “EvilQuest” encrypted files

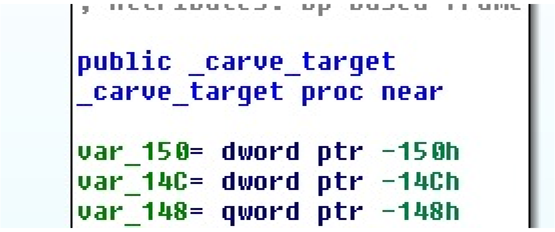
Background

Researchers recently uncovered a new macOS malware threat[1], initially dubbed ‘EvilQuest’ and later ‘ThiefQuest'[2]. The malware exhibits multiple behaviors, including file encryption, data exfiltration and keylogging[3].

Of particular interest from a research perspective is the custom encryption routine. A cursory inspection of the malware code suggests that it is not related to public key encryption. At least part of it uses a table normally associated with RC2. The possible usage of RC2 and time-based seeds for file encryption led me to look deeper at the code, which allowed me to understand how to break the malware’s encryption routine. As a result, our team created a decryptor for public use.

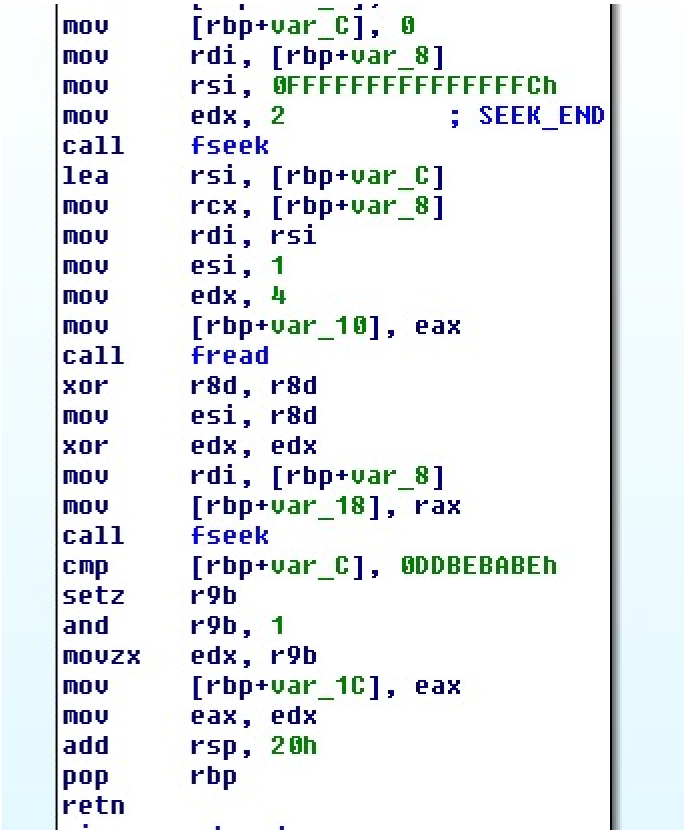
Uncarving the Encryption Routine

As mentioned in other reports[4], the function responsible for file encryption is labelled internally as **carve_target**.



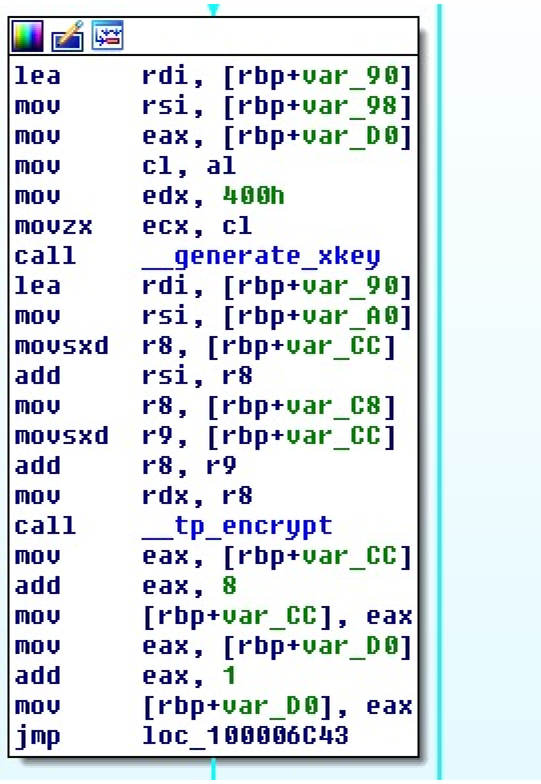
```
public _carve_target
_carve_target proc near

var_150= dword ptr -150h
var_14C= dword ptr -14Ch
var_148= qword ptr -148h
```

Before encrypting the file, the function checks whether the file is already encrypted by comparing the last 4 bytes of the file to a hardcoded DWORD value.



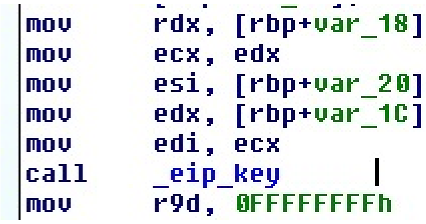
```
mov     [rbp+var_C], 0
mov     rdi, [rbp+var_8]
mov     rsi, 0FFFFFFFFFFFFFFFCh
mov     edx, 2          ; SEEK_END
call    fseek
lea     rsi, [rbp+var_C]
mov     rcx, [rbp+var_8]
mov     rdi, rsi
mov     esi, 1
mov     edx, 4
mov     [rbp+var_10], eax
call    fread
xor     r8d, r8d
mov     esi, r8d
xor     edx, edx
mov     rdi, [rbp+var_8]
mov     [rbp+var_18], rax
call    fseek
cmp     [rbp+var_C], 0DDBEBABEh
setz    r9b
and     r9b, 1
movzx   edx, r9b
mov     [rbp+var_1C], eax
mov     eax, edx
add     rsp, 20h
pop     rbp
retn
```

If the test fails, then file encryption begins by generating a 128 byte key and calling the **tpcrypt** function, which basically ends up calling **generate_xkey**. This function is the key expansion portion followed by **tp_encrypt**, which takes the expanded key and uses it to encrypt the data.



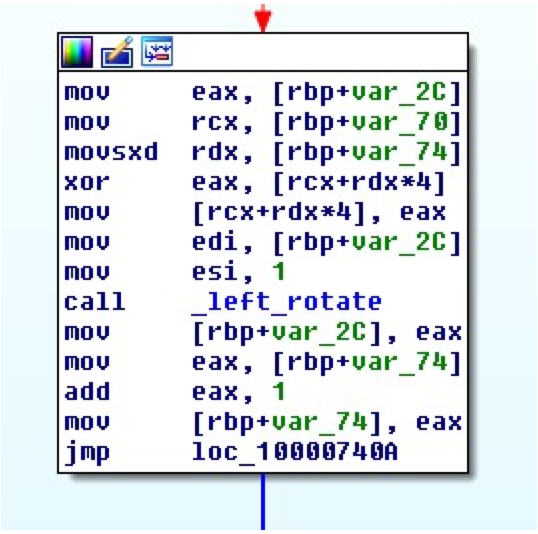
```
lea     rdi, [rbp+var_90]
mov     rsi, [rbp+var_98]
mov     eax, [rbp+var_D0]
mov     cl, al
mov     edx, 400h
movzx   ecx, cl
call    __generate_xkey
lea     rdi, [rbp+var_90]
mov     rsi, [rbp+var_A0]
movsxd  r8, [rbp+var_CC]
add     rsi, r8
mov     r8, [rbp+var_C8]
movsxd  r9, [rbp+var_CC]
add     r8, r9
mov     rdx, r8
call    __tp_encrypt
mov     eax, [rbp+var_CC]
add     eax, 8
mov     [rbp+var_CC], eax
mov     eax, [rbp+var_D0]
add     eax, 1
mov     [rbp+var_D0], eax
jmp     loc_100006C43
```

Following this, the key will then be encoded, using time as a seed. A DWORD value will be generated and utilized.



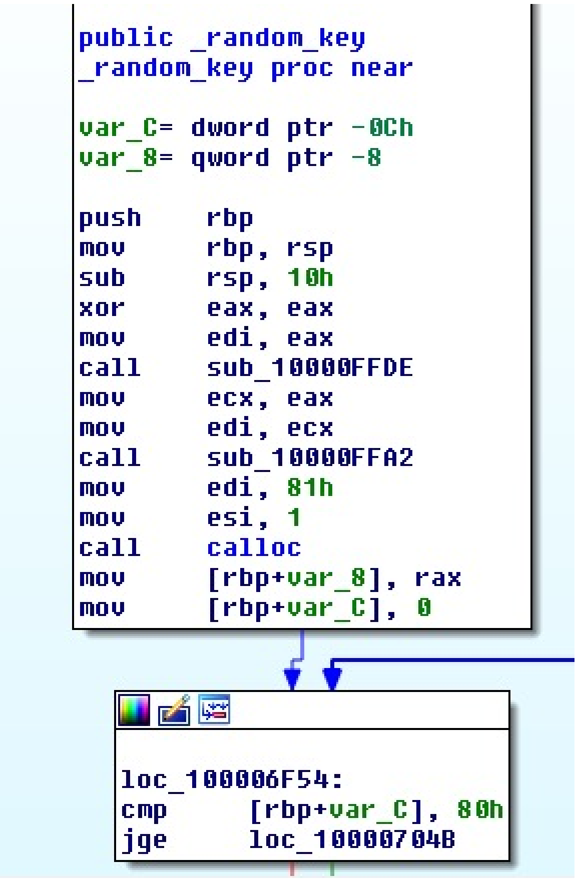
```
mov     rdx, [rbp+var_18]
mov     ecx, edx
mov     esi, [rbp+var_20]
mov     edx, [rbp+var_1C]
mov     edi, ecx
call    _eip_key
mov     r9d, 0FFFFFFFFh
```

The encoding routine is simply a ROL-based XOR loop:



```
mov     eax, [rbp+var_2C]
mov     rcx, [rbp+var_70]
movsxd  rdx, [rbp+var_74]
xor     eax, [rcx+rdx*4]
mov     [rcx+rdx*4], eax
mov     edi, [rbp+var_2C]
mov     esi, 1
call    _left_rotate
mov     [rbp+var_2C], eax
mov     eax, [rbp+var_74]
add     eax, 1
mov     [rbp+var_74], eax
jmp     loc_10000740A
```

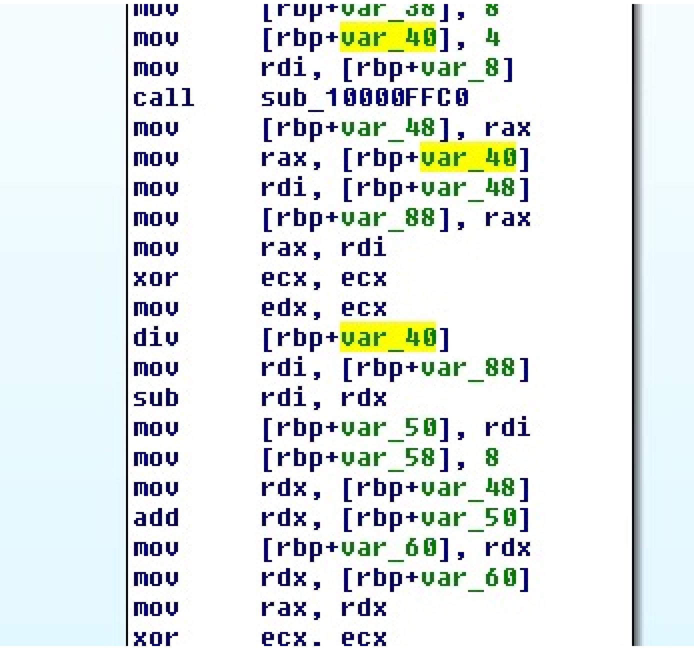
At this point, we can see that something interesting happens, and I am unsure if it is intentional by the developer or not. The key generated is 128 bytes, as we previously mentioned.



```
public _random_key
_random_key proc near

var_C= dword ptr -0Ch
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 10h
xor     eax, eax
mov     edi, eax
call    sub_10000FFDE
mov     ecx, eax
mov     edi, ecx
call    sub_10000FFA2
mov     edi, 81h
mov     esi, 1
call    calloc
mov     [rbp+var_8], rax
mov     [rbp+var_C], 0
```

```
loc_100006F54:
cmp     [rbp+var_C], 80h
jge     loc_10000704B
```

The calculations then used for encoding the key end up performing the loop 4 extra times, producing 132 bytes.

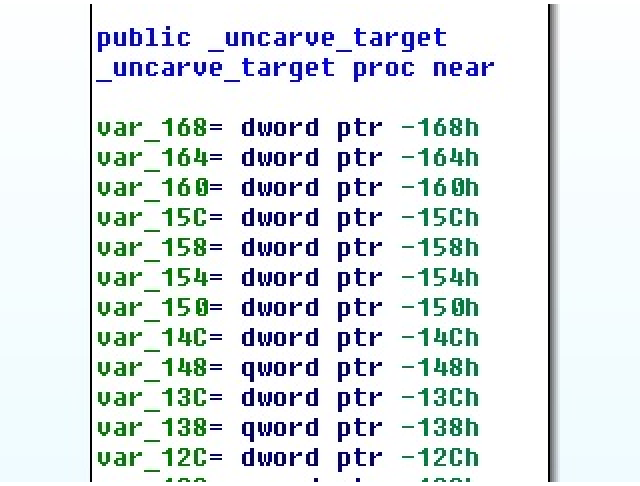


```
mov     [rbp+var_38], 8
mov     [rbp+var_40], 4
mov     rdi, [rbp+var_8]
call    sub_10000FFC0
mov     [rbp+var_48], rax
mov     rax, [rbp+var_40]
mov     rdi, [rbp+var_48]
mov     [rbp+var_88], rax
mov     rax, rdi
xor     ecx, ecx
mov     edx, ecx
div     [rbp+var_40]
mov     rdi, [rbp+var_88]
sub     rdi, rdx
mov     [rbp+var_50], rdi
mov     [rbp+var_58], 8
mov     rdx, [rbp+var_48]
add     rdx, [rbp+var_50]
mov     [rbp+var_60], rdx
mov     rdx, [rbp+var_60]
mov     rax, rdx
xor     ecx, ecx
```

This means that the clear text key used for encoding the file encryption key ends up being appended to the encoded file encryption key. Taking a look at a completely encrypted file shows that a block of data has been appended to it.

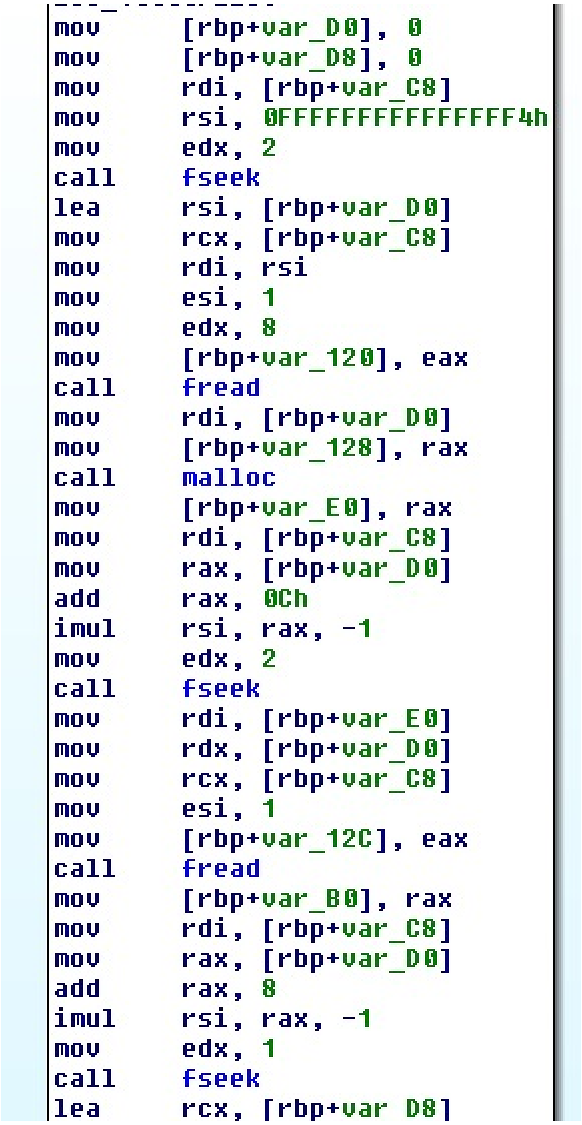
Reversing the File Encryption

Fortunately, we don't have to reverse that much as the actor has left the decryption function, **uncarve_target**, in the code. This function takes two parameters: a file location and a seed value that will be used to decode the onboard file key.



```
public _uncarve_target
_uncarve_target proc near

var_168= dword ptr -168h
var_164= dword ptr -164h
var_160= dword ptr -160h
var_15C= dword ptr -15Ch
var_158= dword ptr -158h
var_154= dword ptr -154h
var_150= dword ptr -150h
var_14C= dword ptr -14Ch
var_148= qword ptr -148h
var_13C= dword ptr -13Ch
var_138= qword ptr -138h
var_12C= dword ptr -12Ch
```

After checking if the file is an encrypted file by examining the last 4 bytes, the function begins reading a structure of data from the end of the file.

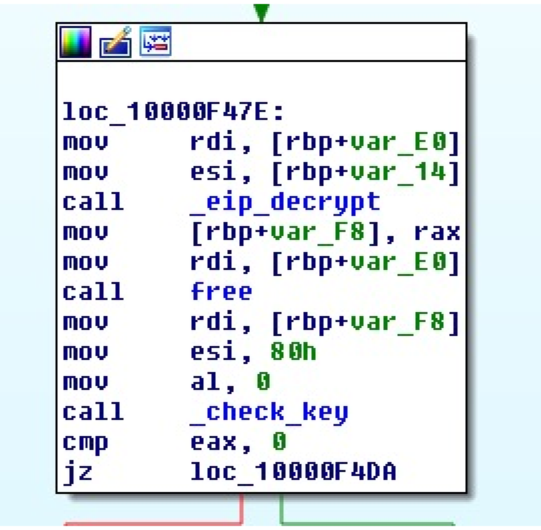


```
mov     [rbp+var_D0], 0
mov     [rbp+var_D8], 0
mov     rdi, [rbp+var_C8]
mov     rsi, 0FFFFFFFFFFFFFFF4h
mov     edx, 2
call    fseek
lea     rsi, [rbp+var_D0]
mov     rcx, [rbp+var_C8]
mov     rdi, rsi
mov     esi, 1
mov     edx, 8
mov     [rbp+var_120], eax
call    fread
mov     rdi, [rbp+var_D0]
mov     [rbp+var_128], rax
call    malloc
mov     [rbp+var_E0], rax
mov     rdi, [rbp+var_C8]
mov     rax, [rbp+var_D0]
add     rax, 0Ch
imul    rsi, rax, -1
mov     edx, 2
call    fseek
mov     rdi, [rbp+var_E0]
mov     rdx, [rbp+var_D0]
mov     rcx, [rbp+var_C8]
mov     esi, 1
mov     [rbp+var_12C], eax
call    fread
mov     [rbp+var_B0], rax
mov     rdi, [rbp+var_C8]
mov     rax, [rbp+var_D0]
add     rax, 8
imul    rsi, rax, -1
mov     edx, 1
call    fseek
lea     rcx, [rbp+var_D8]
```

Following the code execution, we can statically rebuild a version of what this structure might look like:

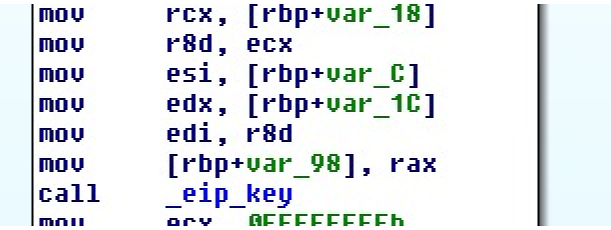
```
struct data
{
enc blob[size+12]
long long size
int marker
}

struct enc
{
long long val
int val2                                    // 3rd param to eip_key
long long val3                              // 1st param to eip_key
char encoded_blob[4 - val % 4 + val]        // for 0x80 this is 132
}
```

The encoded file key will then be decrypted and checked using the two values from the structure and the other seed value passed to **uncarve_target**. The file key will be decrypted by **eip_decrypt**, which is the encrypt-in-place decrypt routine.



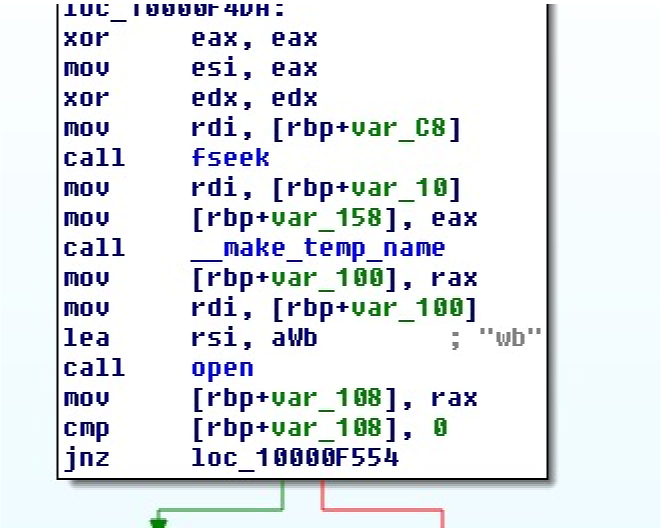
```
loc_10000F47E:
mov     rdi, [rbp+var_E0]
mov     esi, [rbp+var_14]
call    _eip_decrypt
mov     [rbp+var_F8], rax
mov     rdi, [rbp+var_E0]
call    free
mov     rdi, [rbp+var_F8]
mov     esi, 80h
mov     al, 0
call    _check_key
cmp     eax, 0
jz      loc_10000F4DA
```

The function **eip_key** will take the two DWORD values and the seed argument to generate the XOR key to decode the filekey.



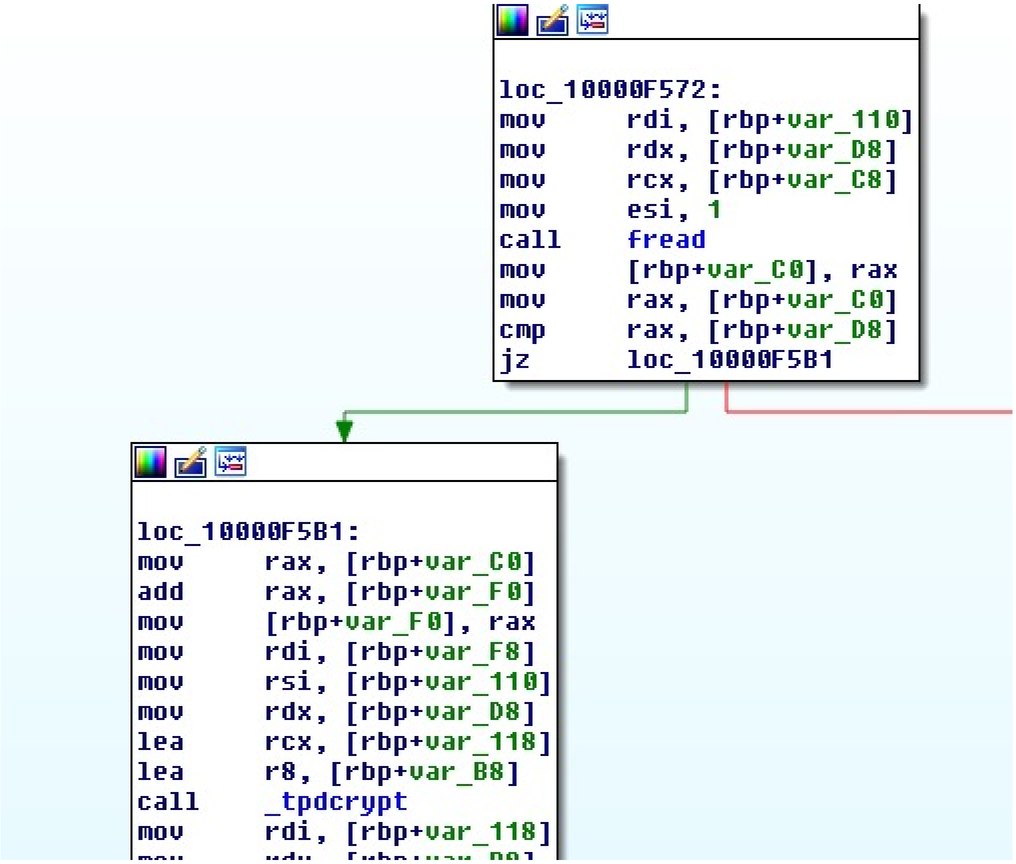
```
mov     rcx, [rbp+var_18]
mov     r8d, ecx
mov     esi, [rbp+var_C]
mov     edx, [rbp+var_1C]
mov     edi, r8d
mov     [rbp+var_98], rax
call    _eip_key
mov     ecx, 0FFFFFFFFh
```

Next, the file is set to the beginning and then a temporary file is opened for writing.



```
loc_10000F4DA:
xor     eax, eax
mov     esi, eax
xor     edx, edx
mov     rdi, [rbp+var_C8]
call    fseek
mov     rdi, [rbp+var_10]
mov     [rbp+var_158], eax
call    __make_temp_name
mov     [rbp+var_100], rax
mov     rdi, [rbp+var_100]
lea     rsi, aWb        ; "wb"
call    open
mov     [rbp+var_108], rax
cmp     [rbp+var_108], 0
jnz     loc_10000F554
```

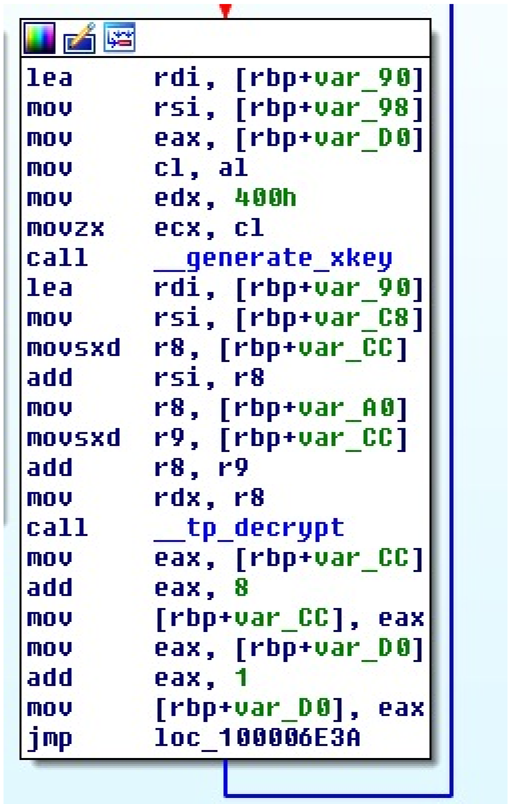
The file is then read into an allocated buffer and the key and encoded file data are passed to **tpdcrypt**.



```
loc_10000F572:
mov     rdi, [rbp+var_110]
mov     rdx, [rbp+var_D8]
mov     rcx, [rbp+var_C8]
mov     esi, 1
call    fread
mov     [rbp+var_C0], rax
mov     rax, [rbp+var_C0]
cmp     rax, [rbp+var_D8]
jz      loc_10000F5B1
```

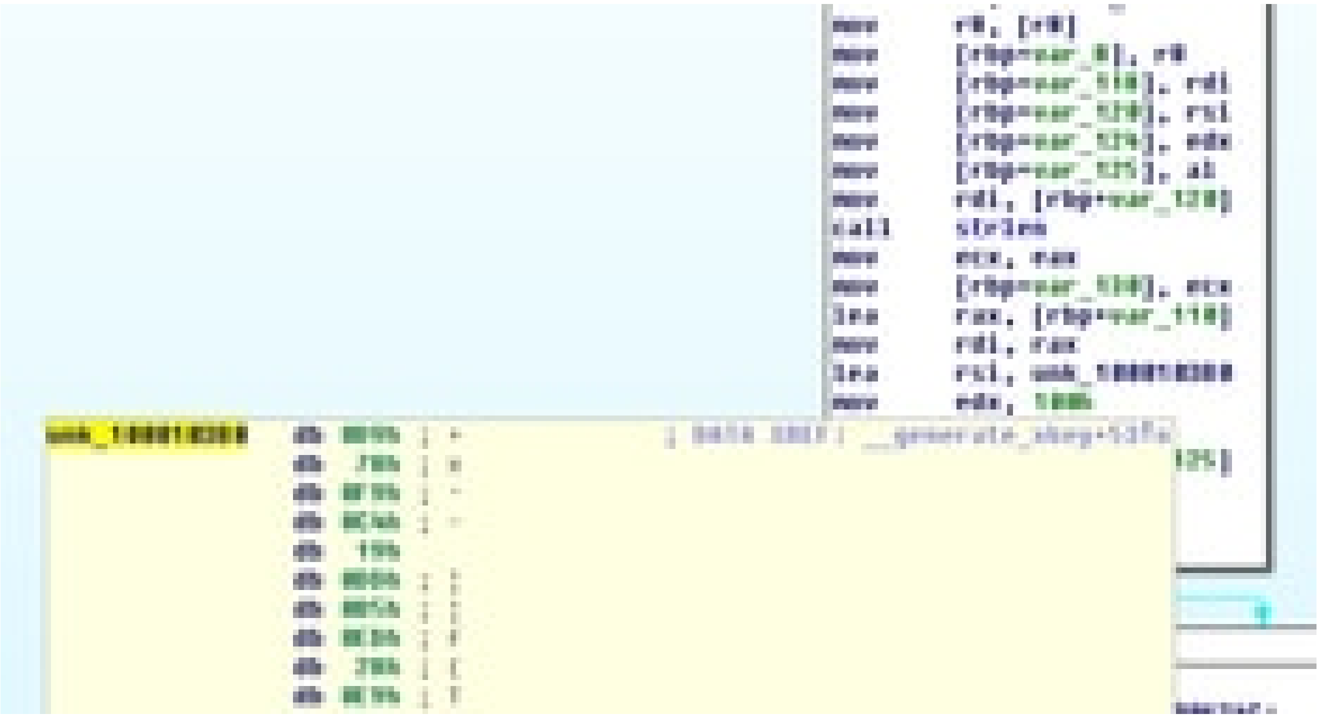
```
loc_10000F5B1:
mov     rax, [rbp+var_C0]
add     rax, [rbp+var_F0]
mov     [rbp+var_F0], rax
mov     rdi, [rbp+var_F8]
mov     rsi, [rbp+var_110]
mov     rdx, [rbp+var_D8]
lea     rcx, [rbp+var_118]
lea     r8, [rbp+var_B8]
call    _tpdcrypt
mov     rdi, [rbp+var_118]
```

As before, we have a key expansion followed this time by a call to **tp_decrypt**.



```
lea     rdi, [rbp+var_90]
mov     rsi, [rbp+var_98]
mov     eax, [rbp+var_D0]
mov     cl, al
mov     edx, 400h
movzx   ecx, cl
call    __generate_xkey
lea     rdi, [rbp+var_90]
mov     rsi, [rbp+var_C8]
movsxd  r8, [rbp+var_CC]
add     rsi, r8
mov     r8, [rbp+var_A0]
movsxd  r9, [rbp+var_CC]
add     r8, r9
mov     rdx, r8
call    __tp_decrypt
mov     eax, [rbp+var_CC]
add     eax, 8
mov     [rbp+var_CC], eax
mov     eax, [rbp+var_D0]
add     eax, 1
mov     [rbp+var_D0], eax
jmp     loc_100006E3A
```

A glance inside the key expansion function shows a reference to a hardcoded table which matches RC2 code that can be found online.



```
mov     r8, [r8]
mov     [rbp+var_8], r8
mov     [rbp+var_[illegible]], rdi
mov     [rbp+var_[illegible]], rsi
mov     [rbp+var_[illegible]], edx
mov     [rbp+var_[illegible]], al
mov     rdi, [rbp+var_[illegible]]
call    strlen
mov     ecx, eax
mov     [rbp+var_[illegible]], ecx
lea     rax, [rbp+var_[illegible]]
mov     rdi, rax
lea     rsi, unk_[illegible]
mov     edx, [illegible]

unk_[illegible]  db 0D9h ; [illegible]      ; DATA XREF: __generate_xkey+[illegible]
                 db  78h ; x
                 db 0F9h ; [illegible]
                 db 0C4h ; [illegible]
                 db  19h
                 db 0DDh ; [illegible]
                 db 0B5h ; [illegible]
                 db 0EDh ; [illegible]
                 db  28h ; (
                 db 0E9h ; [illegible]
```

So now we have enough information to recover the file key:

```
import struct
import sys

rol = lambda val, r_bits, max_bits=32:
(val << r_bits%max_bits) & (2**max_bits-1) |  ((val & (2**max_bits-1)) >> (max_bits-
(r_bits%max_bits)))

data = open(sys.argv[1], 'rb').read()

test = data[-4:]
if test != 'xbexbaxbexdd':
  print("Unknown version")
  sys.exit(-1)

append_length = struct.unpack_from('<I', data[-12:])[0]

append_struct = data[-(append_length+12):]

keySize = struct.unpack_from('<I', append_struct)[0]

if keySize != 0x80:
  print("Weird key?")
  sys.exit(-1)

encoded_data = append_struct[20:20+132]

xorkey = struct.unpack_from('<I', encoded_data[-4:])[0]

def decode(blob, key):
  out = ""
  for i in range(len(blob)/4):
        temp = struct.unpack_from('<I', blob[i*4:])[0]
        temp ^= key
        key = rol(key, 1)
        out += struct.pack('<I', temp)
  return out[:0x80]

temp = decode(encoded_data, xorkey)
print(temp)
```

Attempting to RC2 decrypt the data, however, only seems to work partially at this time using RC2 routines in both Python and Golang libraries. Further analysis will be needed to verify what is different.

However, for the purpose of decrypting victim files, we need only take the file key and call the **tp_decrypt** function that is located inside the malware itself instead. Dumping the assembly for this function and building it into a shared object to be executed using the recovered file key appears to work correctly.

Using this method, SentinelLabs created a public decryptor which is available here (this tool is released under the MIT software license).

Sample

SHA-1: 178b29ba691eea7f366a40771635dd57d8e8f7e8
SHA-256: f409b059205d9a7700d45022dad179f889f18c58c7a284673975271f6af41794

References

1: https://twitter.com/dineshdina04/status/1277668001538433025
2: https://www.bleepingcomputer.com/news/security/thiefquest-ransomware-is-a-file-stealing-mac-wiper-in-disguise/
3: https://blog.malwarebytes.com/mac/2020/06/new-mac-ransomware-spreading-through-piracy/
4: https://objective-see.com/blog/blog_0x59.html