Reversing Teerac

Jason Reaves

February 11, 2016

1 Introduction

2016 is filled with what seems like a new Ransomware every day, whether the influx is due to the recent sale of the CryptoWall source code or the actors involved in Dyre have since moved on to something profitable after the reported takedown, it would appear that for the time being pushing Ransomware is the new hip thing in the malware world. Most of the big names in Ransomware have had plenty of papers and research done but lots of the newer variants while possibly being based on either leaked or sold code will more often than not make changes in order to make themselves unique. Teerac which is a variant of TorrentLocker with a subdomain generation feature to the hardcoded domain is no exception to this as the malware matches multiple reports on TorrentLocker with the exception of an added subdomain generation.

2 Setup

Teerac, which is a TorrentLocker variant, will commonly run from an orphaned explorer.exe process before doing some system setup, first it creates two Cryptographic Service Provider handles, one will be used to create handles from three different AES 256 bit keys in CBC mode all using an IV of 16 NULL bytes and the other will be used with CryptGenRandom. Using one of the AES keys stored in the binary the bot encrypts a copy of itself into a random named folder in `C:\ProgramData\` and sets up persistence in registry with a random named run key. The location of the binary will either be under the `C:\Windows\` directory as a random named executable or will be in `C:\ProgramData\` as a random named executable, the location depends on the privilege level of the malware when it runs. From the samples analyzed it appears the encrypted file 00000000 contains some stats such as how many times the malware has ran, file 01000000 is the encrypted exe and the encrypted file 02000000 contains the location of the non encrypted binary on disk. I've seen sandbox reports and portions of the malware code that indicate this file count can go up to 5 but I've yet to analyze a live sample with a working C2 so most of my research has been purely static analysis.

3 Obfuscation

Teerac employs some basic string obfuscation using a xor key and a lookup table of string addresses and sizes(Figure 3). This is easy to get around using a basic ida python script once we've located the lookup table(Figure 1). Running the script shows us some interesting strings(Figure 12). Some other obfuscation techniques used will be a routine that looks up a corresponding function by hash(Figure 2) and then calls it using a jump table and the occasional dead code.

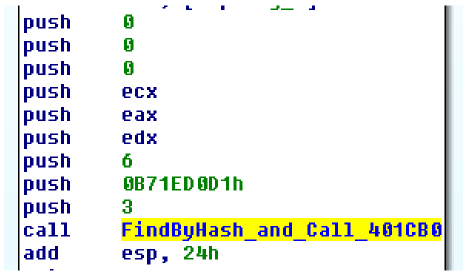
```
#ida python script to decode the strings in the following sample
#teerac_unpacked.exe
#sha256: a92f8b3647fe466ac32d7eee35580888418a47009037e99f9d6374d345f3ceff

xorkeyaddr = 0x413b20
stringTableAddr = 0x413d48

def decode_string(addr, slen, xor_key):
        out = ""
        for i in range(0,slen):
                out += chr((Byte(xor_key) ^ Byte(addr)) & 0xFF)
                addr += 1
                xor_key += 1
        return out

ptr = stringTableAddr
stringLoc = Dword(ptr)
ptr += 4
slen = Dword(ptr)
ptr += 4
while slen < 255:
        print(decode_string(stringLoc, slen, xorkeyaddr))
        stringLoc = Dword(ptr)
        ptr += 4
        slen = Dword(ptr)
        ptr += 4
```

Figure 1: Teerac String Decoding



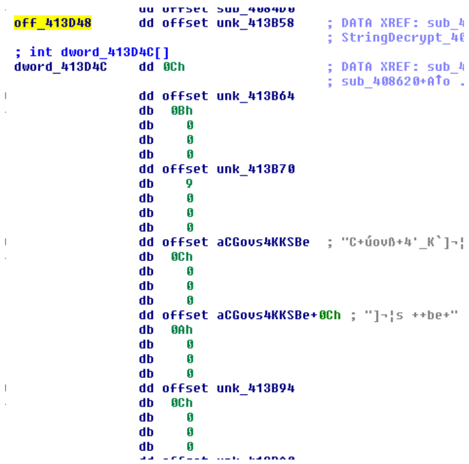
```
push    0
push    0
push    0
push    ecx
push    eax
push    edx
push    6
push    0B71ED0D1h
push    3
call    FindByHash_and_Call_401CB0
add     esp, 24h
```

Figure 2: Find By Hash

4 Interesting Routines

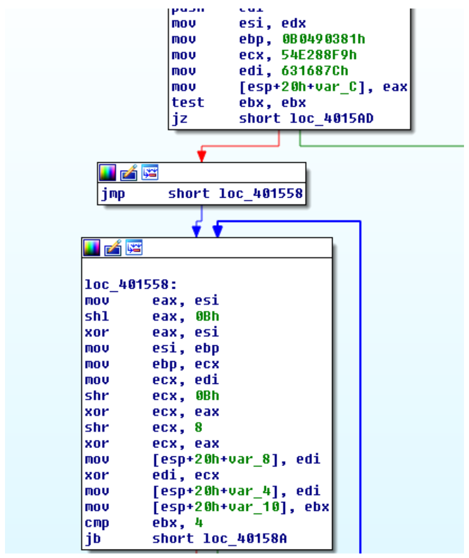
4.1 Pseudo Random 1

Teerac uses two variations of the same routine for generating random strings, while the routine isn't very complex one of the variations that is only used once



```
off_413D48      dd offset unk_413B58    ; DATA XREF: sub_4
                                        ; StringDecrypt_4(
; int dword_413D4C[]
dword_413D4C    dd 0Ch                  ; DATA XREF: sub_4
                                        ; sub_408620+A↑o .
                dd offset unk_413B64
                db  0Bh
                db    0
                db    0
                db    0
                dd offset unk_413B70
                db    9
                db    0
                db    0
                db    0
                dd offset aCGovs4KKSBe  ; "C+úovß+4'_K`]¬¦
                db  0Ch
                db    0
                db    0
                db    0
                dd offset aCGovs4KKSBe+0Ch ; "]¬¦s ++be+"
                db  0Ah
                db    0
                db    0
                db    0
                dd offset unk_413B94
                db  0Ch
                db    0
                db    0
                db    0
```

Figure 3: Encoded String Lookup Table



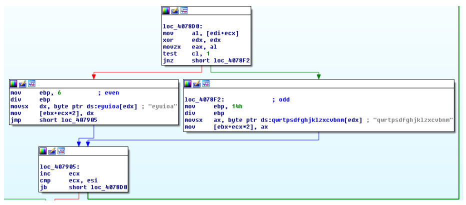
```
mov     esi, edx
mov     ebp, 0B0490381h
mov     ecx, 54E288F9h
mov     edi, 631687Ch
mov     [esp+20h+var_C], eax
test    ebx, ebx
jz      short loc_4015AD

jmp     short loc_401558

loc_401558:
mov     eax, esi
shl     eax, 0Bh
xor     eax, esi
mov     esi, ebp
mov     ebp, ecx
mov     ecx, edi
shr     ecx, 0Bh
xor     ecx, eax
shr     ecx, 8
xor     ecx, eax
mov     [esp+20h+var_8], edi
xor     edi, ecx
mov     [esp+20h+var_4], edi
mov     [esp+20h+var_10], ebx
cmp     ebx, 4
jb      short loc_40158A
```

Figure 4: Teerac Rand Bytes

caught my attention because it led me to a similar routine that was used by HesperBot. The routine from Teerac(Figure 4) matches with a routine that was reversed and published by Vinnu on garage4hackers[3]. This routine is used in a similar fashion as it was in HesperBot where it generates a long string of bytes which then feed into another routine, like a stream of salt values, striking a resemblance to the same report, this routine uses the byte stream to generate a string of characters in a seemingly random fashion. One major difference I see in Teerac is a separation of consonants and the vowels where the it will pull a



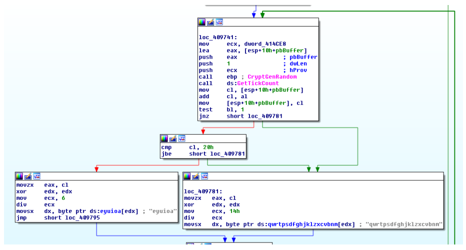
```
loc_4078D0:
mov     al, [edi+ecx]
xor     edx, edx
movzx   eax, al
test    cl, 1
jnz     short loc_4078F2
```

```
mov     ebp, 6          ; even
div     ebp
movsx   dx, byte ptr ds:eyuioa[edx] ; "eyuioa"
mov     [ebx+ecx*2], dx
jmp     short loc_407905
```

```
loc_4078F2:             ; odd
mov     ebp, 14h
div     ebp
movsx   ax, byte ptr ds:qwrtpsdfghjklzxcvbnm[edx] ; "qwrtpsdfghjklzxcvbnm"
mov     [ebx+ecx*2], ax
```

```
loc_407905:
inc     ecx
cmp     ecx, esi
jb      short loc_4078D0
```

Figure 5: Teerac Random String Gen 1



```
loc_409741:
mov     ecx, dword_414CE8
lea     eax, [esp+10h+pbBuffer]
push    eax             ; pbBuffer
push    1               ; dwLen
push    ecx             ; hProv
call    ebp ; CryptGenRandom
call    ds:GetTickCount
mov     cl, [esp+10h+pbBuffer]
add     cl, al
mov     [esp+10h+pbBuffer], cl
test    bl, 1
jnz     short loc_409781
```

```
cmp     cl, 20h
jbe     short loc_409781
```

```
movzx   eax, cl
xor     edx, edx
mov     ecx, 6
div     ecx
movsx   dx, byte ptr ds:eyuioa[edx] ; "eyuioa"
jmp     short loc_409795
```

```
loc_409781:
movzx   eax, cl
xor     edx, edx
mov     ecx, 14h
div     ecx
movsx   dx, byte ptr ds:qwrtpsdfghjklzxcvbnm[edx] ; "qwrtpsdfghjklzxcvbnm"
```

Figure 6: Teerac Random String Gen 2

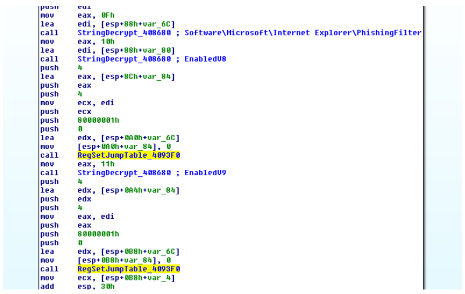
vowel every even iteration of the loop and vice-versa(Figure 5). In the samples I looked at this routine was only used to generate the string for the mutexs, folder names and part of the bot id.

4.2 Psuedo Random 2

The second random string generator is used for generating everything else from filenames, subdomain names and exe names. The biggest difference between the two is that instead of using the seed routine detailed earlier it uses CryptGenRandom and GetTickCount to create bytes that will be used to index into the character arrays in order to build the random strings along with an extra check almost making it look like a generation algorithm with a slight bias(Figure 6).

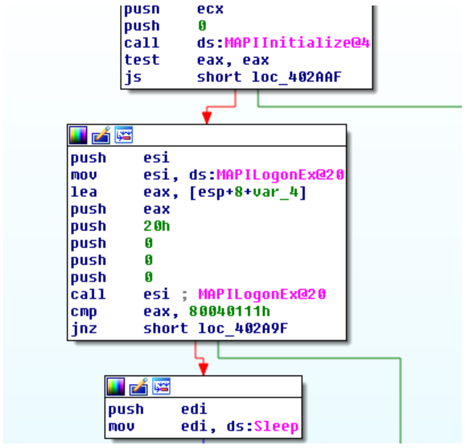
4.3 AES Key Generation

As previously mentioned one of the routines matches the DGA routine that was previously used by HesperBot but as it turns out it's not the only piece of code that shares a striking resemblance to anaysis reports on HesperBot. The first half of the AES key the bot generates is based on various information about the machine, specifically it uses the InstallDate value from `HKLM\SOFTWARE\Mirosoft\WindowsNT\CurrentVersion` as the first dword of the key The second is a hardcoded value depending on if the version is x86 x64 or IA64. The third is the adler32 hash of the value of MachineGuid from `HKLM\SOFTWARE\Microsoft\Cryptography`. The fourth is the adler32 hash of the value of DigitalProductId from `HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion`. This was the same information was used to build an encryption key used by HesperBot



```
push     edi
mov      eax, 0Fh
lea      edi, [esp+88h+var_6C]
call     StringDecrypt_408680 ; Software\Microsoft\Internet Explorer\PhishingFilter
mov      eax, 10h
lea      edi, [esp+88h+var_80]
call     StringDecrypt_408680 ; EnabledV8
push     4
lea      eax, [esp+8Ch+var_84]
push     eax
push     4
mov      ecx, edi
push     ecx
push     80000001h
push     0
lea      edx, [esp+0A0h+var_6C]
mov      [esp+0A0h+var_84], 0
call     RegSetJumpTable_4093F0
mov      eax, 11h
call     StringDecrypt_408680 ; EnabledV9
push     4
lea      edx, [esp+0A4h+var_84]
push     edx
push     4
mov      eax, edi
push     eax
push     80000001h
push     0
lea      edx, [esp+0B8h+var_6C]
mov      [esp+0B8h+var_84], 0
call     RegSetJumpTable_4093F0
mov      ecx, [esp+0B8h+var_4]
add      esp, 30h
```

Figure 7: Teerac Turning Off SmartScreen

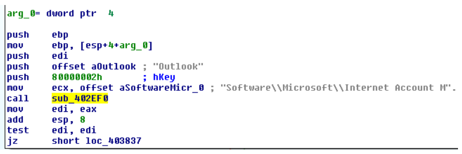


```
push     ecx
push     0
call     ds:MAPIInitialize@4
test     eax, eax
js       short loc_402AAF
```

```
push     esi
mov      esi, ds:MAPILogonEx@20
lea      eax, [esp+8+var_4]
push     eax
push     20h
push     0
push     0
push     0
call     esi ; MAPILogonEx@20
cmp      eax, 80040111h
jnz      short loc_402A9F
```

```
push     edi
mov      edi, ds:Sleep
```

Figure 8: Teerac MAPI via COM



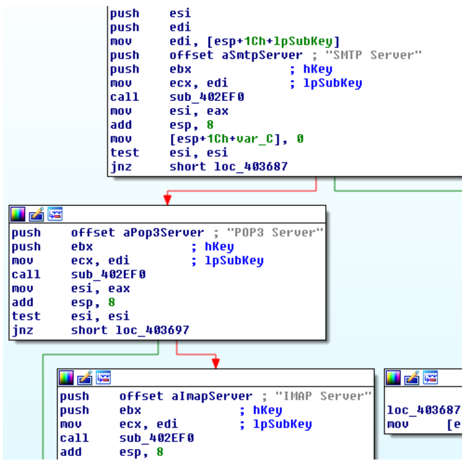
```
arg_0= dword ptr  4

push     ebp
mov      ebp, [esp+4+arg_0]
push     edi
push     offset aOutlook ; "Outlook"
push     80000002h       ; hKey
mov      ecx, offset aSoftwareMicr_0 ; "Software\\Microsoft\\Internet Account M".
call     sub_402EF0
mov      edi, eax
add      esp, 8
test     edi, edi
jz       short loc_403837
```

Figure 9: Teerac Outlook Harvest

to store it's configuration data and plugin modules according to a report by welivesecurity.com[4].

The other two AES keys used came hardcoded in the binary, one was used for encrypting the files the malware used in the random folder in `ProgramData` and the other key was used for C2 communication. As previously mentioned all keys are used in CBC Mode with an IV of 16 NULL byte.



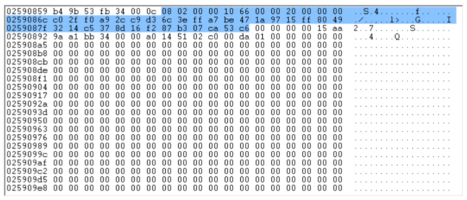
```
push    esi
push    edi
mov     edi, [esp+1Ch+lpSubKey]
push    offset aSmtpServer ; "SMTP Server"
push    ebx             ; hKey
mov     ecx, edi        ; lpSubKey
call    sub_402EF0
mov     esi, eax
add     esp, 8
mov     [esp+1Ch+var_C], 0
test    esi, esi
jnz     short loc_403687
```

```
push    offset aPop3Server ; "POP3 Server"
push    ebx             ; hKey
mov     ecx, edi        ; lpSubKey
call    sub_402EF0
mov     esi, eax
add     esp, 8
test    esi, esi
jnz     short loc_403697
```

```
push    offset aImapServer ; "IMAP Server"
push    ebx             ; hKey
mov     ecx, edi        ; lpSubKey
call    sub_402EF0
add     esp, 8
```

```
loc_403687
mov     [e
```

Figure 10: Teerac Email Server Info



```
02590859 b4 9b 53 fb 34 00 0c 08 02 00 00 10 66 00 00 20 00 00 00   ..S.4.......f.. ...
0259086c c0 2f f0 a9 2c c9 d3 6c 3e ff a7 be 47 1a 97 15 ff 80 49   ./..,..l>...G.....I
0259087f 32 14 c5 37 8d 16 f2 87 b3 07 ca 53 c6 00 00 00 00 15 aa   2..7........S......
02590892 9a a1 bb 34 00 00 a0 14 51 02 c0 00 da 01 00 00 00 00 00   ...4....Q..........
025908a5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025908b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025908cb 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025908de 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025908f1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590904 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590917 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
0259092a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
0259093d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590950 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590963 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590976 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
02590989 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
0259099c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025909af 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025909c2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025909d5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
025909e8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...................
```

Figure 11: Hardcoded C2 AES Key

5 Worker Threads

Right before it begins trying to find a live C2 the bot spins up 5 threads. One of these threads will delete shadow copies using vssadmin.exe. Another will disable some security features in the registry, phishing filter and SmartScreen(Figure 7). Another simply returns immediately. The other two will utilize COM through CoInitialize, one thread enumerates the registry trying to find various outlook(Figure 9) and account settings(Figure 10) while the other uses MAPI(Figure 8) in an attempt to harvest your contacts.

6 C2 Communications

C2 subdomains are constructed by using a hardcoded domain and then generating a random subdomain to add to the beginning of it. From what I've seen when you have a live C2 usually any subdomain will resolve to the same IP of the hardcoded domain. First checkin to the C2 the bot will take the botid which is generated based on the computer name and some hex data converted to ascii which is comprised of random generated data that's been encrypted and

a static string 'main-4' which I've also seen 'main-5' so this appears to be a campaign designator. This data will be encrypted using one of the hardcoded AES keys previously mentioned(Figure 11) and then POSTed up to the C2. The information that follows is purely static for reasons I previously mentioned but I can at least confirm a few things about what the bot expects as response from the C2.
1. The data from the C2 is encrypted with the same key the bot used to encrypt the data that is sent to the C2.
2. The decrypted response data will be parsed in the following structured format $< c2DATA >$ %%1%% $< c2DATA >$ %%2%% $< c2DATA >$. 3. The bot performs validation on the data and will not continue execution until it has gotten a proper response.

7 Conclusions

Sample SHA256: f5f7cb83a8f229b96a39f2be7a686fdecd717f2519ffe5b62bc98ff439b6f583

```
Other Samples:
89edb283b3a3c892cb8ed7fa893aff5f36982fc3f4657c3b0723351212ded3e6
c4928426873726e4eeb341aaea33d07f41cef58193eb1655bfe1ee6a97afd4c8
2c6b46b60b4ddb5e75a45a9ba2e57a60a1d95bd798bac6b3036ecde237dddb74
56cbf1281a50e0082a1db873bec0097b61c6074152d40598f73c094d37674ea6
6ef7c2cd280b17ea104f7c9c75711992176bb2b854424b779e6da7becda8d998
43d0b93f825a60c676eeab175cc11eea07f1b598bee08bf57d99c64f41a9b8c6
580c61c84d588f32b0cb6b4203cf5918a0c63a15b1529d5ea0ba105b59ab4373
7db8759c7260b71866d896c9a381f47b8d7e452aea3d1d8aab41e38085ccfb70
fe17addfb458cf66f4a922f342baf4337ec33e9e1aa3b715ec94e676ca74417b
c9e9f81c9438ea7a062b41bbb1c121f88b6a372c4eb15030c50a3f16b714b62d
3c38e1e5956c2a9f6fe4f33d52d5c1ddbdc2e43abeda25b16f7ae4aa7eaa610f
545f991909341b92702a0aa2aa18c4ccceefad207af2180aeed24f5c1b346037

C2 domains observed:
megezawone.net
vodleklina.org
pyjtoxoyr.org
ioytoxpaire.net
kdiertyjoxeg.com
vjivebilan.org
jgiwoxoqlwez.com
rygzatyee.com
asoijaisojais.net
nemexcikx.net
kheoyostowe.net
lderktdfphje.net
```

```
explorer.exe
svchost.exe
ntdll.dll
kernel32.dll
user32.dll
advapi32.dll
shlwapi.dll
wininet.dll
%ProgramFiles%
%ProgramW6432%
.encrypted
CryptoLocker
.exe
.html
.txt
Software\Microsoft\Internet Explorer\PhishingFilter
EnabledV8
EnabledV9
vssadmin.exe Delete Shadows /All /Quiet
UNKNOWN
Common Files\System\wab32.dll
Thunderbird\Profiles\
abook.mab
history.mab
DisplayName
PrimaryEmail
// <!-- <mdb:mork:z v="1.4"/> -->
@$${
{@
@$$}
}@
(^%X^
^%x)
(%X=
about:blank
```

Figure 12: Teerac Decoded Strings

```
C2 IPs observed:
31.170.104.60
188.225.34.221
80.78.253.130
91.214.114.122
```

References

[1] Hex-Rays Decompiler, http://www.hex-rays.com/products/decompiler/index.shtml.

[2] Python, https://www.python.org/

[3] Hesperbot DGA, http://garage4hackers.com/entry.php?b=3091

[4] Hesperbot Technical analysis part 1/2, http://www.welivesecurity.com/2013/09/06/hesperbot-technical-analysis-part-12/

[5] Emerging Threats, http://www.emergingthreats.net/

[6] Update on the Torrentlocker ransomware, http://blog.fox-it.com/2014/10/21/update-on-the-torrentlocker-ransomware/