Goldeneye Ransomware - the Petya/Mischa combo rebranded | Malwarebytes Labs

By Malwarebytes Labs

Published: 2016-12-14 · Archived: 2026-04-05 17:16:42 UTC

From March 2016 we've observed the evolution of an interesting low-level ransomware, Petya – you can read about it here. The second version (green) Petya comes combined with another ransomware, packed in the same dropper – Mischa. The latter one was deployed as an alternative payload: in case if the dropper was run without administrator privileges and the low-level attack was impossible. This combo is slowly reaching its maturity – the authors fixed bugs that allowed for decryption of the two earliest versions. Now, we are facing an outbreak of the fourth version – this time under a new name – Goldeneye, and, appropriately, a new, golden theme.

In this post we will take a look inside, in order to answer the question of whether or not any internal changes followed the external alterations.

Analyzed sample

- e068ee33b5e9cb317c1af7cecc1bacb5 – original sample (packed)
  - 08b079609c2a3a4deb4f11cf373f9278 – core.dll (dropper) // UPDATE: replaced with a fixed dump
    - 0cd94baa2dccc0e7c2008b7948cebfe3 – elevate_x86.dll
    - 54fb6dbad73eee5d8638c0869c35ed8f – elevate_x64.dll
    - e5a2cc00d1ad8d409576bc6d24a346bd – Petya Golden (dump from the disk)
- 435076f9c8900cbdfc48a15713b1c431 – Goldeneye Decrypter (original)

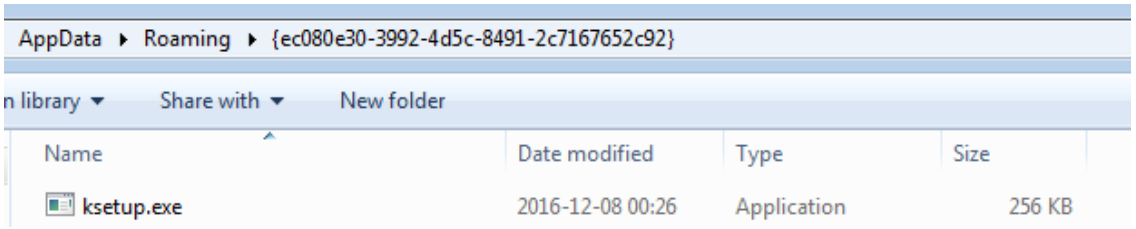
// special thanks to @procrash

Distribution

Currently Goldeneye is distributed by phishing e-mails, in campaigns targeting Germany. The same pattern of distribution was observed in first editions of Petya ransomware. Germany seems to be an environment familiar to this ransomware author (who is probably a German native speaker) and his testing campaigns are always released in this country. However, the threat will probably go global again, as the affiliate program for other criminals is going to be released soon.

Behavioural analysis

After being run, the malware installs its copy in the %APPDATA% directory, under the name of a random application found in the system:

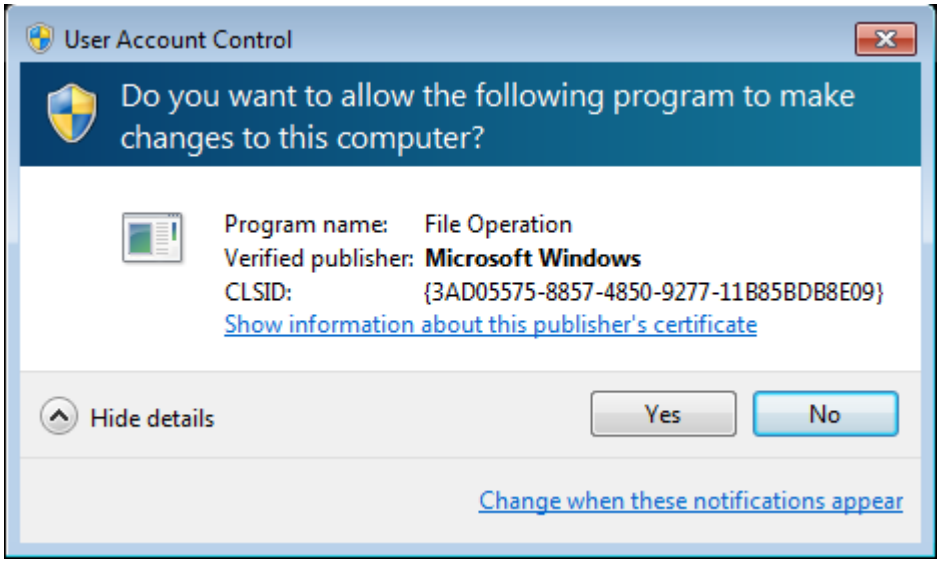


The installed copy is automatically executed and proceeds with malicious actions.

In the past, the dropper of Petya/Mischa used to trigger a UAC popup window. If the user had agreed to run the sample as the Administrator, he/she was attacked by the low-level payload: Petya. Otherwise, the high-level Mischa was deployed.

In the current case the model of the attack is different and looks more like a case of Satana ransomware.

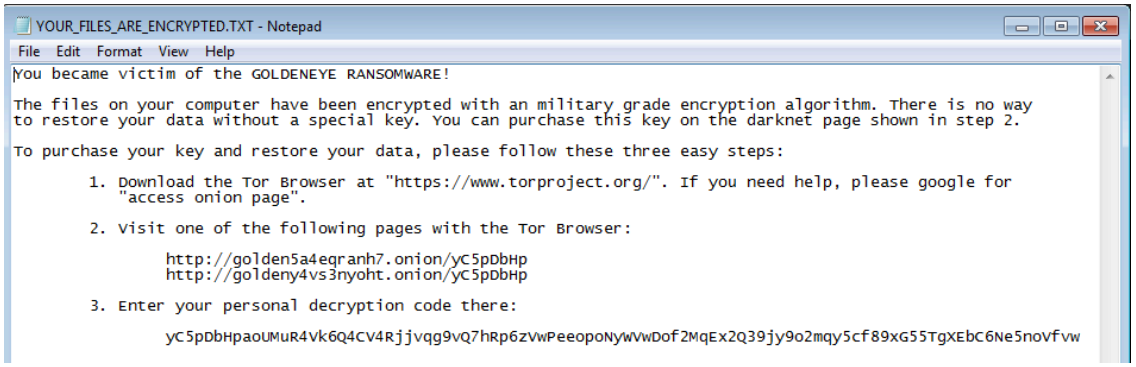
First, the high-level attack is deployed and the files are encrypted one by one. Then, the malware tries to bypass UAC and elevate its privileges by its own, in order to make the second attack, this time at low-level: installing Petya at the beginning of the disk. The bypass works silently if the UAC is set to default or lower. In cases where the UAC is set to max, the following window pops up repeatedly, till the user accepts the elevation:



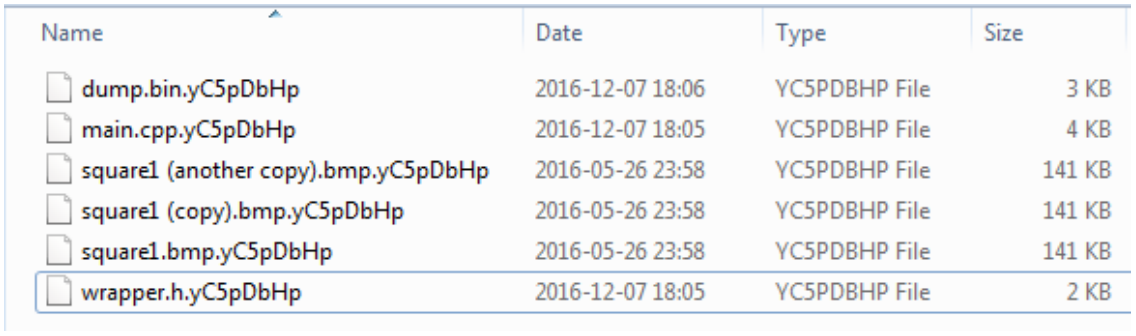
The used bypass techniques works on both – 32-bit and 64-bit – versions of Windows, up to Windows 8.1. On Windows 10, even if the UAC is set to default a popup is displayed – but not revealing the real name of the infecting program, i.e.

The high-level part (former Mischa)

On the first stage of the attack, files are being encrypted one by one. The malware drops the following note in TXT format:

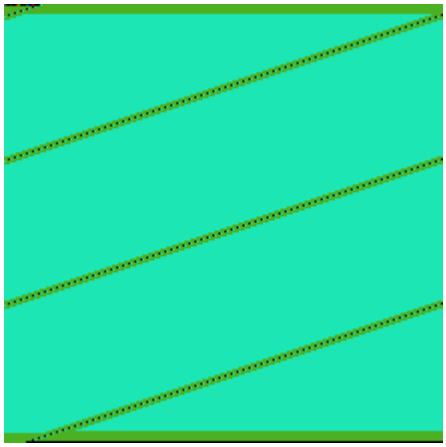


Files that are encrypted are added random extensions:



If we have two files with the same plaintext they turn into two different cipher-texts – that indicates that each file is encrypted with a new key or an initialization vector. The high entropy suggests AES in CBC mode.

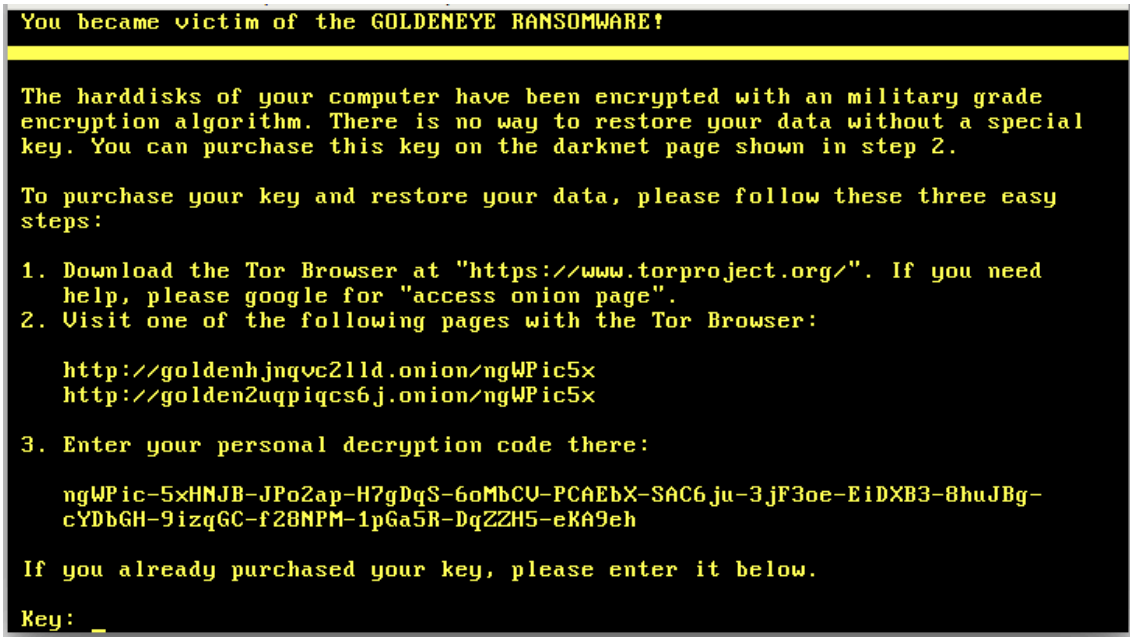
Visualization – original file vs encrypted one:



The low-level part (former Petya)

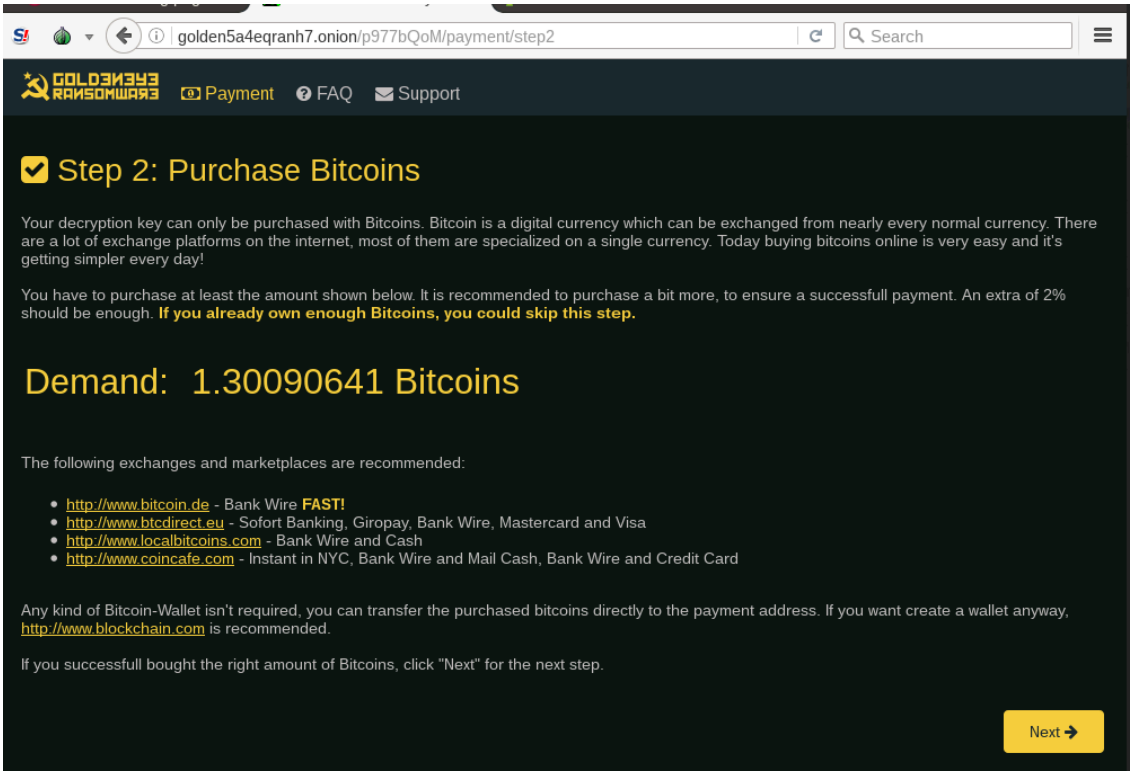
The second stage of infection is deployed after encrypting the files. The behavior of second payload is no different than in the previous versions of Petya. After the malware is deployed, system crashes and starts with a fake CHKDSK. It pretends to be checking the disk for errors, but in reality it performs Master File Table encryption, using Salsa20. After it is completed, we are facing a familiar blinking skull – this time in yellow/golden color:”>

After pressing a key, we can see the screen with the ransom note:

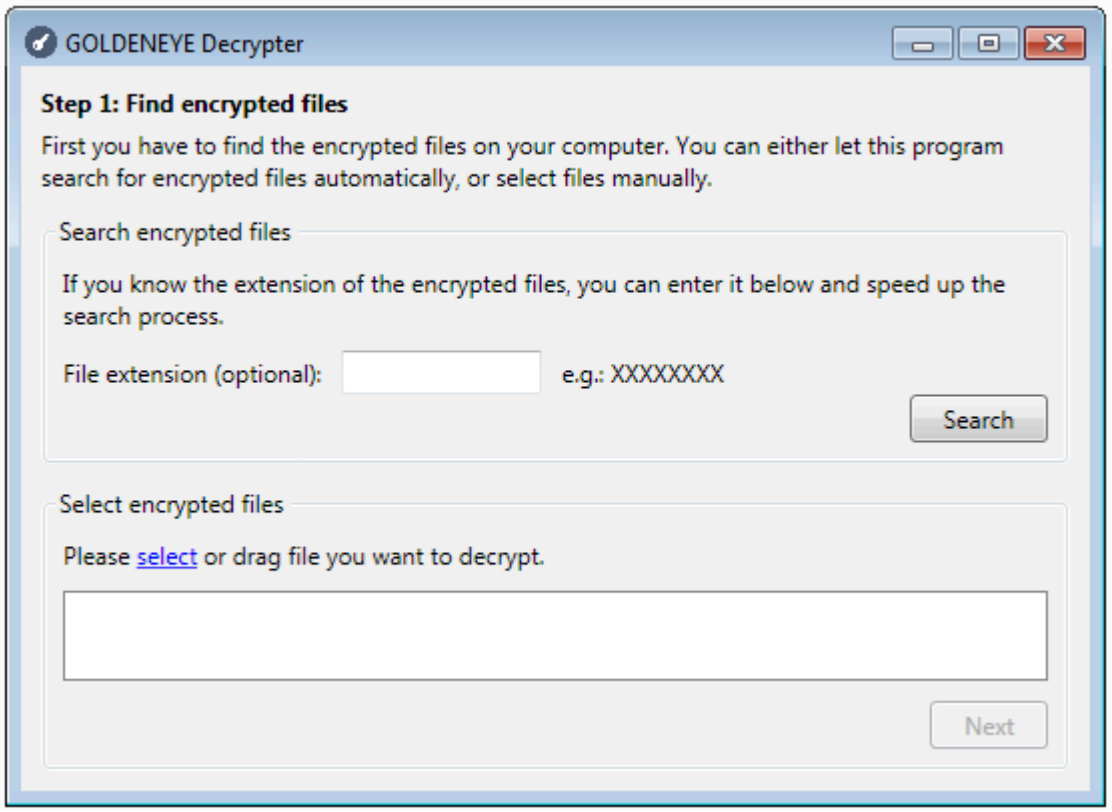


Page for the victim

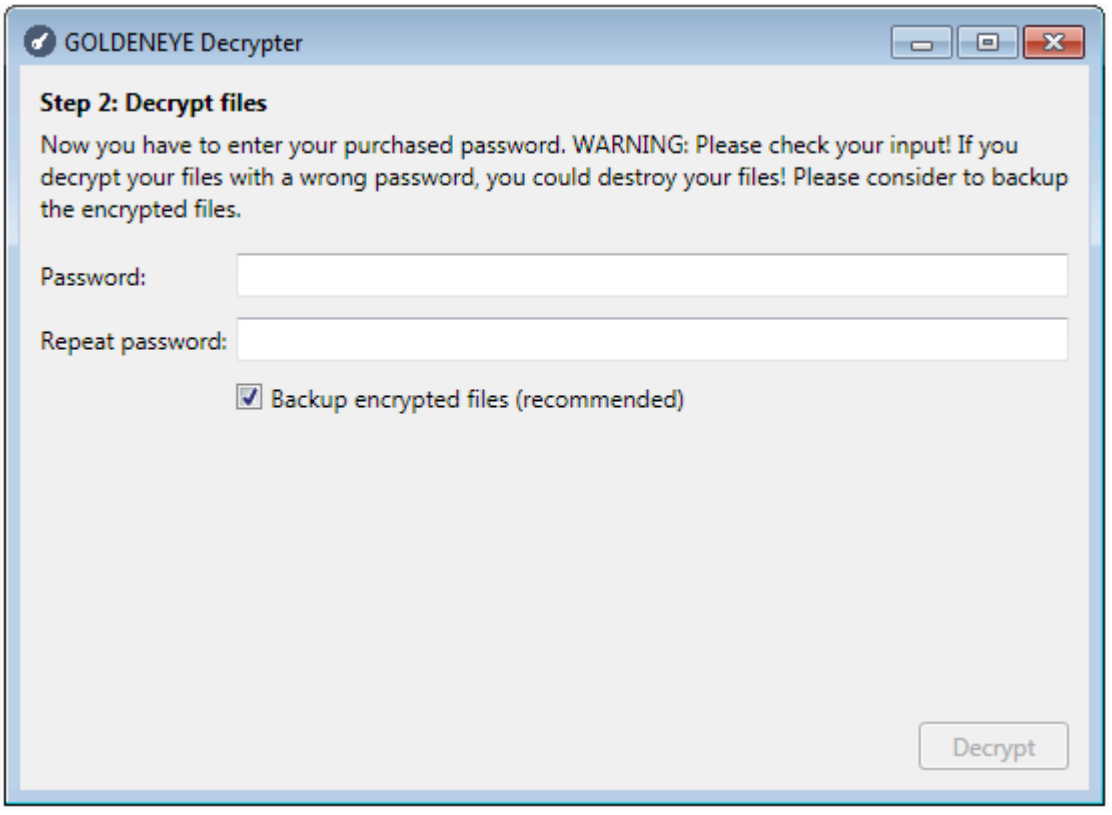
On every edition all the pieces of the ransomware had a consistent theme. This time is no different. The page for the victim, that is hosted on a Tor-based site comes in very similar theme like the ransomware itself:



After paying the ransom, the victim is provided with a key to decrypt the first (bootlocker) stage and a decrypter to recover the files:

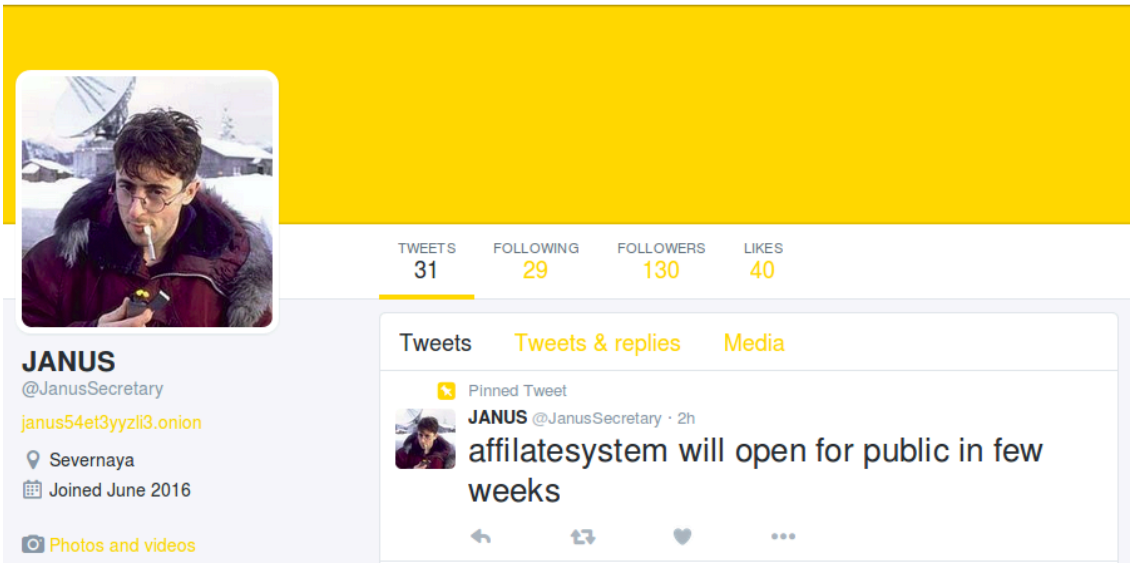


The decrypter requires having a proper key in order to work:



Affiliate program

In the past, Petya/Mischa combo was available as RaaS (Ransomware as a Service). Following the changes in the layout, the Twitter account associated with the criminal(s) behind the malware, also changed the theme of the profile, and updated the information about the affiliate program status:



It confirms that the actor behind Goldeneye as well as the methods of redistributing it didn't change.

Inside

This ransomware is very complex, having multiple pieces that have already been described in our previous articles. That's why, in this one we will focus only on the differences comparing to the previous editions. Let's start from the *core.dll*, that is the PE file that we get after unpacking the first layer.

The core.dll

Just like in the previous versions, the main application is a DLL (*core.dll*), packed by various crypters and loaded by a technique known as *Reflective Loader*.

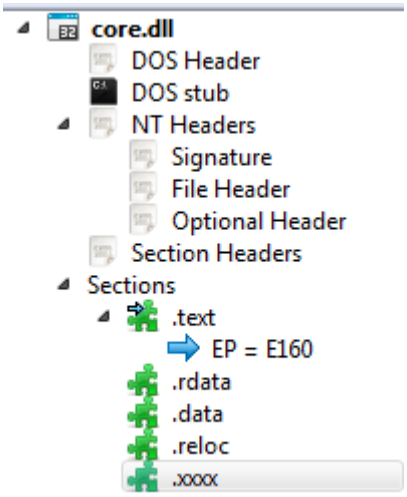
Offset	Name	Value	Meaning
EB90	Characteristics	0	
EB94	TimeDateStamp	5845AF3A	
EB98	MajorVersion	0	
EB9A	MinorVersion	0	
EB9C	Name	F7C2	core.dll
EBA0	Base	1	
EBA4	NumberOfFunctions	1	
EBA8	NumberOfNames	1	
EBAC	AddressOfFunctions	F7B8	
EBB0	AddressOfNames	F7BC	
EBB4	AddressOfNameOrdinals	F7C0	

Details

Offset	Ordinal	Function RVA	Name RVA	Name
EBB8	1	CB95	F7CB	_ReflectiveLoader@4

In the past Petya and Mischa were two separate modules delivered by this DLL. The dropper was deciding which one of them to deploy, by making an attempt to run the sample with Administrator privileges – no UAC bypass was used, only social engineering. Now, however, it comes with two DLLs that perform UAC bypass – one for 32 bit and another for 64 bit variant of Windows. It decides which one to deploy, basing on the detected architecture.

The internal logic of this module changed a bit. There is no *Mischa.dll* separated. Instead, the *core.dll* covers the functionality of encrypting files as well as of installing disk locker afterwards. The payloads are XOR encrypted and stored in the last section of the PE file (*.xxxx*):



Section *.xxxx* contains:

- the low level part (former Petya)
- 32 bit DLL (elevate_x86.dll)
- 64 bit DLL (elevate_x64.dll)

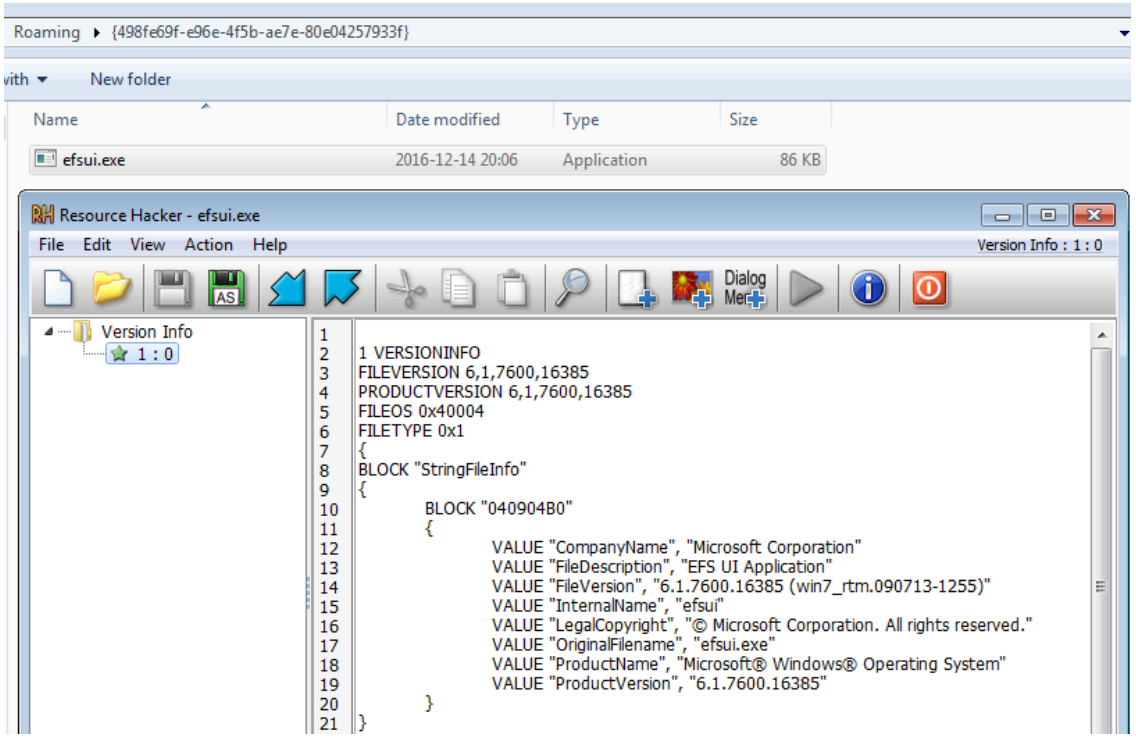
(The two DLLs used to UAC bypass are based on the technique similar to the one described here.)

At first run, the core module makes its own copy into %APPDATA% and applies some tricks to blend into the environment:

- Choosing the application name at random, out of various applications in System folder
- Changing own timestamp to the timestamp of Kernel32.dll (the so called “timestomping” technique).
- Adding to its resources the resource of the genuine Microsoft application, under which name it is installed:

```
0FDDC4FE  .   PUSH 0x1
0FDDC500  .   PUSH EDI
0FDDC501  .   CALL DWORD PTR DS:[EAX+0x298]      kernel32.FindResourceA
0FDDC507  .   MOV EBP,EAX
0FDDC509  .   TEST EBP,EBP
0FDDC50B  .v  JNZ SHORT core1.0FDDC514
0FDDC50D  >   XOR EAX,EAX
0FDDC50F  .v  JMP core1.0FDDC59B
0FDDC514  >   MOV EAX,DWORD PTR DS:[0xFDE28C4]
0FDDC519  .   PUSH EBP
0FDDC51A  .   PUSH EDI
0FDDC51B  .   CALL DWORD PTR DS:[EAX+0x29C]      kernel32.LoadResource
0FDDC521  .   TEST EAX,EAX
0FDDC523  .^  JE SHORT core1.0FDDC50D
0FDDC525  .   PUSH EAX
0FDDC526  .   MOV EAX,DWORD PTR DS:[0xFDE28C4]
0FDDC52B  .   CALL DWORD PTR DS:[EAX+0x2A0]      kernel32.LockResource
0FDDC531  .   MOV DWORD PTR SS:[ESP+0xC],EAX
0FDDC535  .   TEST EAX,EAX
0FDDC537  .^  JE SHORT core1.0FDDC50D
0FDDC539  .   MOV EAX,DWORD PTR DS:[0xFDE28C4]
0FDDC53E  .   PUSH EBX
0FDDC53F  .   PUSH 0x0
0FDDC541  .   PUSH ESI
0FDDC542  .   CALL DWORD PTR DS:[EAX+0x2A4]      kernel32.BeginUpdateResourceA
0FDDC548  .   MOV EBX,EAX
```

Result:



Some of those tricks remind us of Cerber ransomware and they were probably inspired by it.

Then, the dropper deploys the installed copy and proceeds with encryption.

The file cryptor (former Mischa)

The file cryptor feature is now implemented inside the *core.dll*.

It behaves similarly to the former Mischa ransomware – the only difference is that now it is employed before the low-level attack, rather than being an alternative.

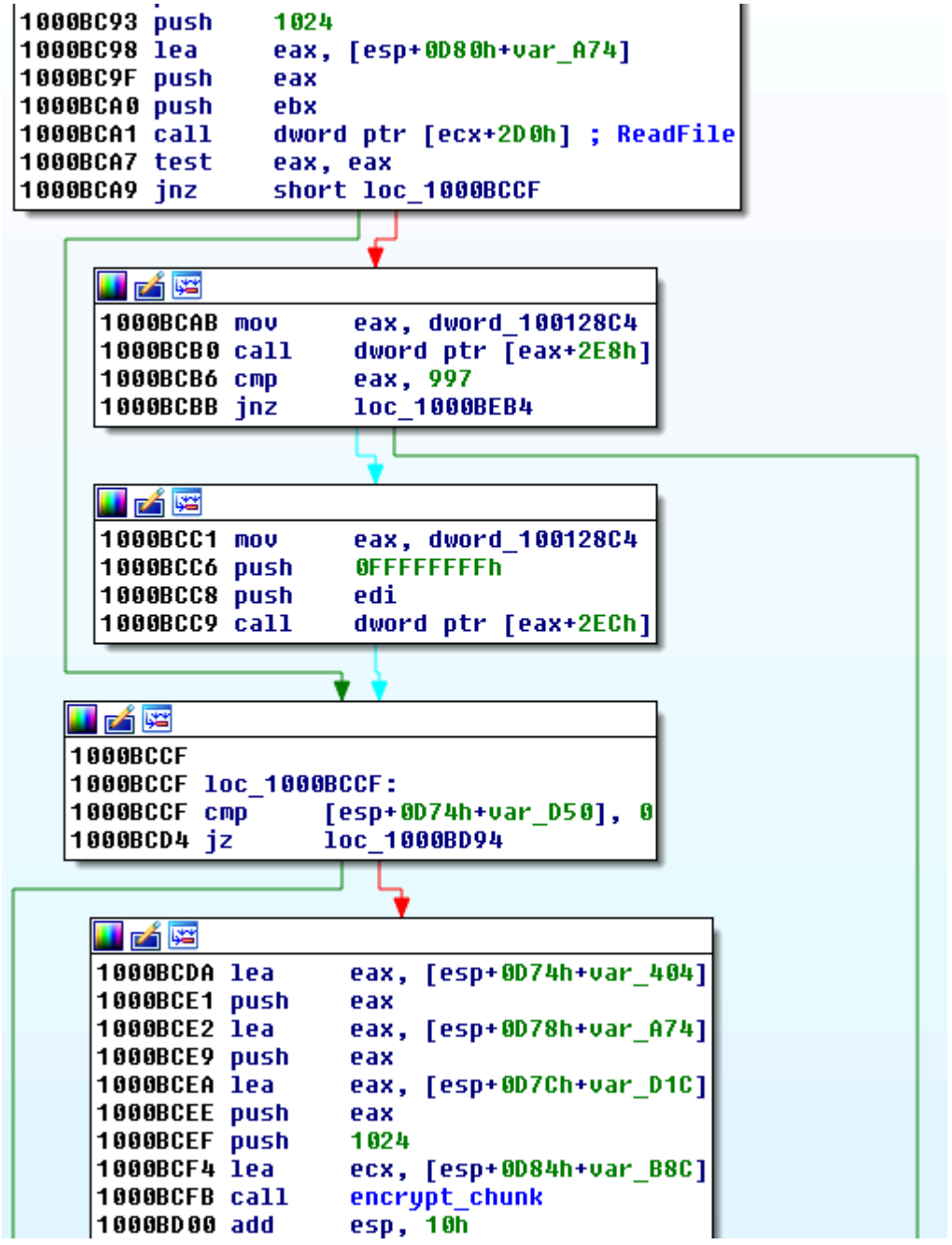
Attacked targets

Files are attacked with the following extensions:

```
doc docx docm odt ods odp odf odc odm odb xlsm xlsb xlk xls xlsx  pps ppt pptm pptx pub epub pdf  jp
```

Encryption

Files are read in chunks, each is 1024 bytes long. Then, they are processed by the built-in implementation of AES.



The easiest way to analyze the encryption algorithm used, is by reversing the original decrypter, provided by the ransomware author to victims that paid the ransom. The decrypter is written in .NET and not obfuscated.

Looking at the decrypter code we can confirm that each file is encrypted using AES in CBC mode. The AES key is 32 byte long, and it is the taken from the beginning of SHA512 hash of the password.

```
this.DecryptPassword1.Text = this.DecryptPassword1.Text.Replace("-", "").Replace(" ", "");
this.DecryptPassword2.Text = this.DecryptPassword1.Text;
byte[] hash = new SHA512Managed().ComputeHash(Step2.StringToByteArray(this.DecryptPassword1.Text)).Take(32).ToArray<byte>();
```

The initialisation vector is random for every file and it is stored in its content:

```
binaryReader.BaseStream.Seek(-1L, SeekOrigin.End);
byte b = binaryReader.ReadByte();
if (binaryReader.BaseStream.Length < (long)(b + 1))
{
    goto IL_2FF;
}
binaryReader.BaseStream.Seek((long)((int)(21 + b + 1) * -1), SeekOrigin.End);
bool flag = binaryReader.ReadByte() == 1;
int num = binaryReader.ReadInt32();
byte[] array = new byte[16];
binaryReader.Read(array, 0, 16);
if (num == 0)
{
    throw new ArgumentException("File is damaged!");
}
binaryReader.BaseStream.Seek(0L, SeekOrigin.Begin);
using (RijndaelManaged rijndaelManaged = new RijndaelManaged
{
    Key = hash,
    IV = array,
    Mode = CipherMode.CBC,
    Padding = PaddingMode.Zeros
})
{
    using (CryptoStream cryptoStream = new CryptoStream(binaryReader.BaseStream, rijndaelManaged.CreateDecryptor(hash, array),
      CryptoStreamMode.Read))
    {
```

The disk locker (former Petya)

This part of the Goldeneye ransomware is written at the disk beginning and is independent from the operating system. It is made up of a bootloader and a tiny, 16-bit kernel. At the very first sight we can suspect, that it is nothing more than a refactored Petya. That's why, for the simplicity I will refer this part as Petya Goldeneye.

Indeed, comparing the current edition with Petya 3 (described here) we can see, that the encryption algorithm and the codebase hasn't changed. Yet, we can spot some differences.

Encryption

All versions of Petya use Salsa20 to encrypt MFT. In the current edition, the implementation of Salsa20 is identical like in the former version.

See the *BinDiff* screenshot below – Petya Goldeneye vs Petya 3:

similarity	confidence	change	EA primary	name primary	EA secondary	name secondary
1.00	0.98	-------	00009462	rotl	000096C4	sub_96C4_70
1.00	0.88	-------	00009578	s20_columnround	000097DA	sub_97DA_73
1.00	0.99	-------	00009798	s20_crypt	000099FA	sub_99FA_79
1.00	0.99	-------	000095D8	s20_doubleround	0000983A	sub_983A_74
1.00	0.99	-------	000096D4	s20_expand32	00009936	sub_9936_78
1.00	0.99	-------	00009652	s20_hash	000098B4	sub_98B4_77
1.00	0.99	-------	000095EC	s20_littleendian	0000984E	sub_984E_75
1.00	0.99	-------	0000949A	s20_quarterround	000096FC	sub_96FC_71
1.00	0.98	-------	00009628	s20_rev_littleendian	0000988A	sub_988A_76
1.00	0.88	-------	00009518	s20_rowround	0000977A	sub_977A_72

We can safely assume, that just like in the previous case the Salsa20 has been implemented correctly – means, this edition of Petya is not decryptable by external tools.

What has changed in the code?

Although the main parts of the code didn't change, still we can notice that some refactoring has taken place:

0.99	0.99	-I-----	00008C98	deploy_crypt	00008E52	sub_8E52_61
0.99	0.99	-I-----	0000811A	fake_chkdsk	0000811A	sub_811A_34
0.93	0.98	GI-----	00008DE2	deploy_salsa	00008FA0	sub_8FA0_62
0.93	0.98	GI-----	00008FA6	mft_salsa_crypt	00009146	sub_9146_63
0.72	0.98	GI---L-	00009386	xor_crypt	0000951E	sub_951E_64
0.69	0.95	-I--E--	0000891E	reboot_disk	00008ADA	sub_8ADA_47
0.27	0.65	GI--ELC	00008220	decrypt	00008228	sub_8228_36

The most important changes are about the way in which the encryption/decryption is applied. The author added more checks and simplified the decryption function. Yet, the changes are rather about improving the code quality rather than introducing some new ideas.

Layout

Just like in the previous cases, Petya's code is written at the beginning of the disk – however, now the layout is more compact. The code of Petya's kernel starts just after MBR, without any padding. Due to this, other important sectors are also shifted. For example, the data sector, where the random salsa key is saved*, is now placed in sector 32:

```
00003FF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004000  00 65 65 89 D1 CC 56 41 82 D6 20 74 C2 53 D0 09  .ee‰ÑÌVA‚Ö tÂSÐ.    Sector 32
00004010  76 3B C9 A3 5E FE 55 DC 01 9C A7 19 0F 12 3E E3  v;É£^þUÜ.œ§...>ã
00004020  32 8C E9 F6 EA 8B 29 B8 93 68 74 74 70 3A 2F 2F  2Œéöê‹)¸"http://
00004030  67 6F 6C 64 65 6E 68 6A 6E 71 76 63 32 6C 6C 64  goldenhjnqvc2lld
00004040  2E 6F 6E 69 6F 6E 2F 72 7A 76 6A 34 33 66 57 00  .onion/rzvj43fW.
00004050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00004060  00 00 00 00 00 00 00 00 00 68 74 74 70 3A 2F 2F  .........http://
00004070  67 6F 6C 64 65 6E 32 75 71 70 69 71 63 73 36 6A  golden2uqpiqcs6j
00004080  2E 6F 6E 69 6F 6E 2F 72 7A 76 6A 34 33 66 57 00  .onion/rzvj43fW.
```

** just like in all previous editions, this key is erased after use. Read more about the full procedure here.*

Summing up, all the sectors are shifted towards the beginning of the disk.

Data sector:

- Petya3: 54
- Petya Goldeneye: 32

Verification sector:

- Petya3: 55
- Petya Goldeneye: 33

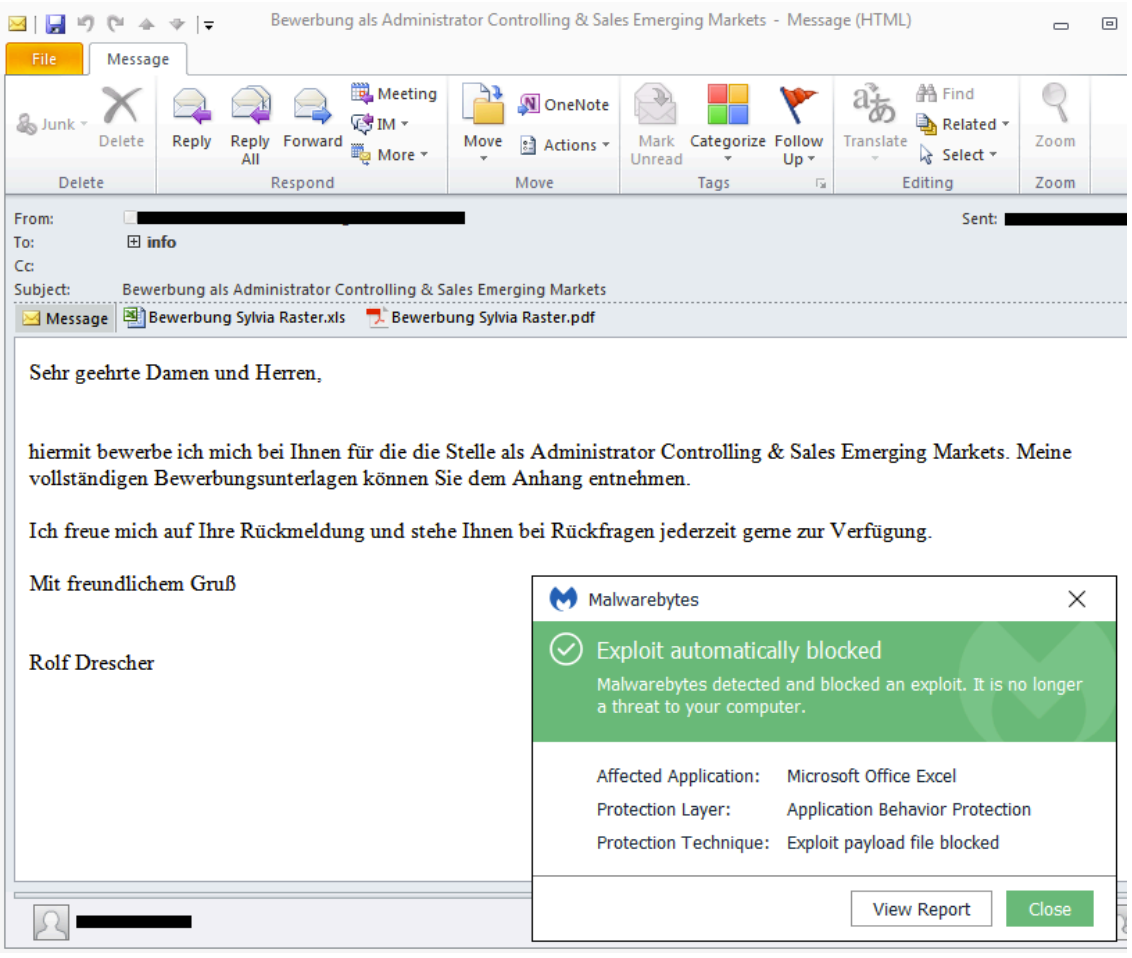
Original MBR (xored with 7)

- Petya3: 56
- Petya Goldeneye: 34

Conclusion

Goldeneye ransomware is yet another step in the development of the Petya/Mischa bundle. The redesigned dropper coupled both elements together in a new way, that makes it even more dangerous. At the current stage the product doesn't seem decryptable by external tools. We strongly advise to be very vigilant about opening e-mail attachments, because this is still the main way of distribution of this ransomware.

During the tests, Malwarebytes has proven to protect against the malicious payloads attached to Goldeneye phishing e-mails:



This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.
