EternalPetya – yet another stolen piece in the package?

blog.malwarebytes.com/threat-analysis/2017/06/eternalpetya-yet-another-stolen-piece-package/

Malwarebytes Labs

June 30, 2017

Since June 27th we have been investigating the outbreak of the new Petya-like malware armed with an infector similar to WannaCry. Since day one, various contradicting theories started popping up. Some believed that this malware is a rip-off of the original Petya, while others think that it is another step in Petya's evolution. However, those were just different opinions and none of them were backed up with enough evidence to hold solid. In this post, we will try to fill this gap by making step-by-step comparisons of the current kernel and the one on which it is based (Goldeneye Petya).

Analyzed sample:

Why is it important to know whether or not the code was recompiled?

Answering this question and collecting enough evidence is crucial for further discussions on attribution. The source code of the original Petya has never been leaked publicly, so in case it was recompiled it proves that the original Petya's author, Janus, is somehow linked to the current outbreak (either this is his work or he has sold the code to another actor).

In this analysis, we hope to identify if this malware could have been recompiled from the original code, or it's just a work of anyone with the appropriate skills to modify the ready-made binary. Doing so would not entirely disprove Janus as the creator, but his involvement becomes less likely.

Anyways, let’s take a look at the code.

Sectors

Looking at the sectors, we can find that the layout of EternalPetya is identical to Goldeneye. Full comparison:

Petya kernel:

- Petya Goldeneye: sector 1
- Petya Eternal: sector 1

Data sector:

- Petya Goldeneye: 32
- Petya Eternal: 32

Verification sector:

- Petya Goldeneye: 33
- Petya Eternal: 33

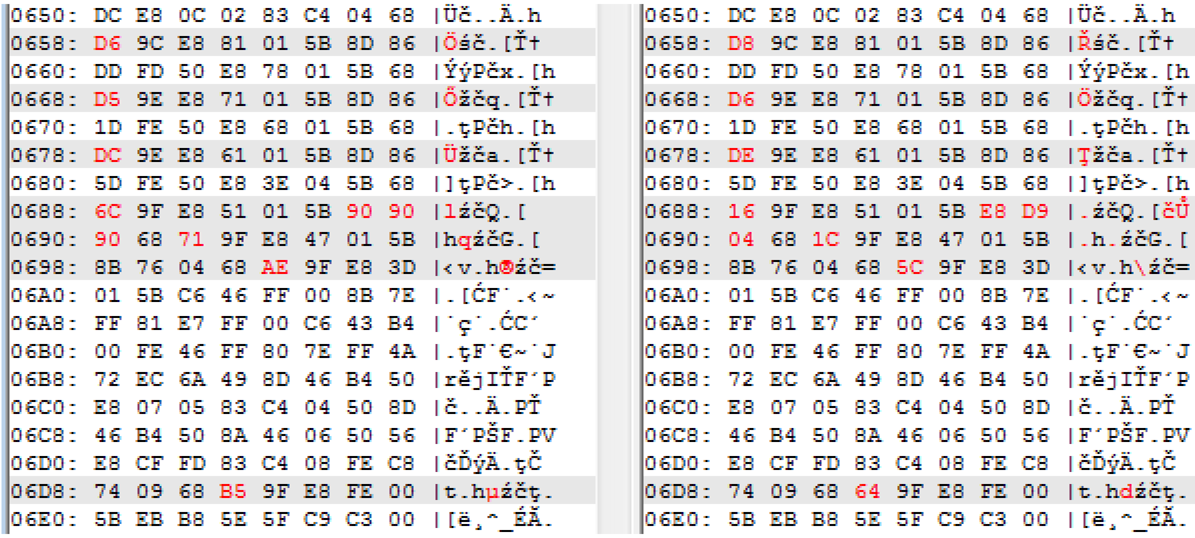
Original MBR (xored with 7)

- Petya Goldeneye: 34
- Petya Eternal: 34

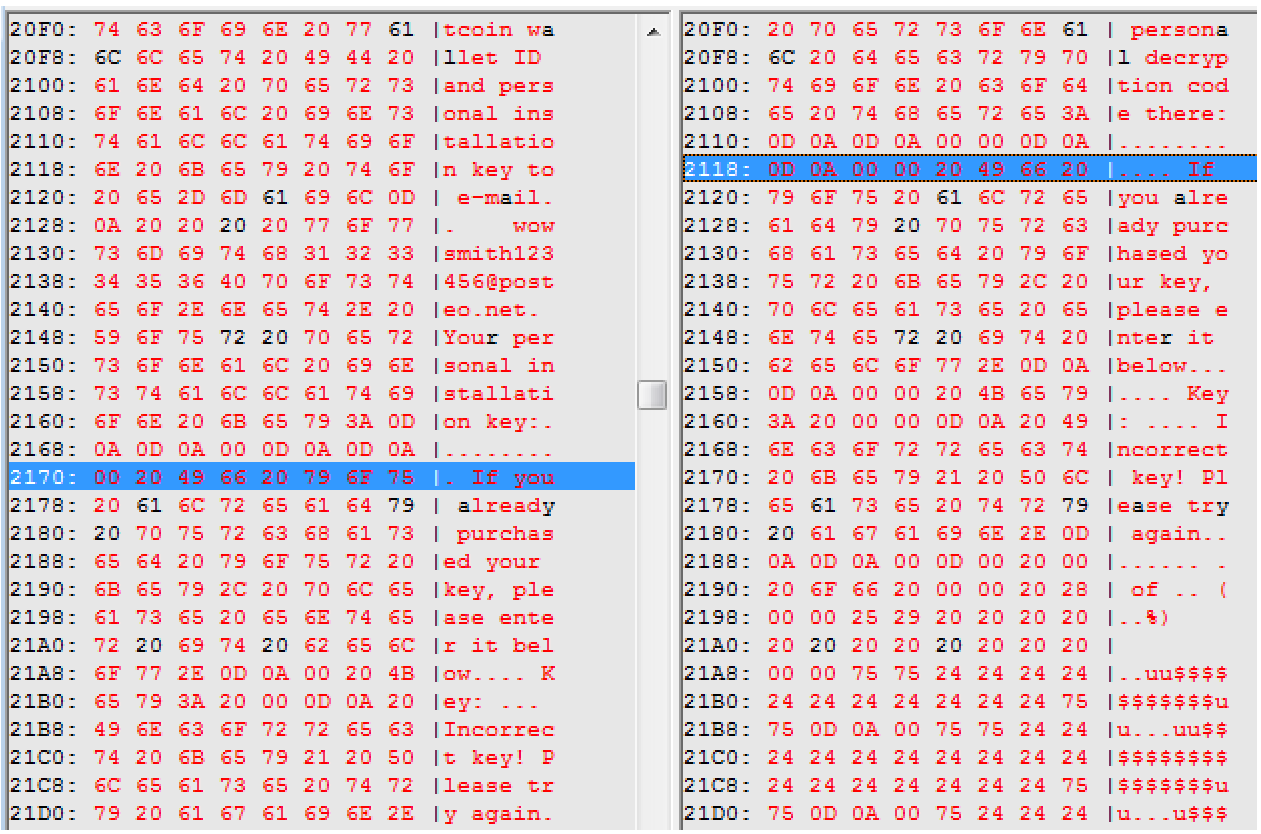
Hexadecimal comparison

Comparing both kernels at hexadecimal level, we can see tiny differences at various points. However, there are big portions of code that are identical in both.

The screenshots below show fragments of the (current) EternalPetya on the left, and Goldeneye on the right.



Its interesting that, at some point, the layout of the same strings in the memory was shifted:



As mentioned, the data sector starts in both cases at the same offset. This sector stores the random Salsa20 key and nonce, which are generated per victim, and this is identical in both cases. However, in Goldeneye the victim ID is much longer, which is not surprising taking into the account the fact that in the past it was supposed to be the encrypted backup of the Salsa key, and now it is just an arbitrary string, so it’s length doesn’t really matter.

```
3FF8: 00 00 00 00 00 00 00 00 |........
4000: 00 3D FE F2 0D 72 92 CC |.=ţň.r’Ě
4008: 5E 6F 01 15 78 93 07 0C |^o..x“..
4010: 3E 61 92 68 A8 EF 91 AD |>a’h¨ď‘-
4018: 10 7B CF 19 0A 7C C5 33 |.{Ď..|Ĺ3
4020: E0 E1 02 71 42 E4 09 F8 |ŕá.qBä.ř
4028: 05 31 4D 7A 37 31 35 33 |.1Mz7153
4030: 48 4D 75 78 58 54 75 52 |HMuxXTuR
4038: 32 52 31 74 37 38 6D 47 |2R1t78mG
4040: 53 64 7A 61 41 74 4E 62 |SdzaAtNb
4048: 42 57 58 00 00 00 00 00 |BWX.....
4050: 00 00 00 00 00 00 00 00 |........
4058: 00 00 00 00 00 00 00 00 |........
4060: 00 00 00 00 00 00 00 00 |........
4068: 00 00 00 00 00 00 00 00 |........
4070: 00 00 00 00 00 00 00 00 |........
4078: 00 00 00 00 00 00 00 00 |........
4080: 00 00 00 00 00 00 00 00 |........
4088: 00 00 00 00 00 00 00 00 |........
4090: 00 00 00 00 00 00 00 00 |........
4098: 00 00 00 00 00 00 00 00 |........
40A0: 00 00 00 00 00 00 00 00 |........
40A8: 00 71 56 62 6E 64 42 70 |.qVbndBp
40B0: 36 57 59 73 6B 52 4A 5A |6WYskRJZ
40B8: 4A 35 51 53 51 34 6E 41 |J5QSQ4nA
40C0: 51 53 38 6F 6D 51 79 4D |QS8omQyM
40C8: 33 7A 4A 4C 64 4D 48 58 |3zJLdMHX
40D0: 68 41 63 51 50 68 44 58 |hAcQPhDX
40D8: 55 76 51 70 53 58 34 5A |UvQpSX4Z
40E0: 33 52 66 67 77 00 00 00 |3Rfgw...
40E8: 00 00 00 00 00 00 00 00 |........
40F0: 00 00 00 00 00 00 00 00 |........
40F8: 00 00 00 00 00 00 00 00 |........
4100: 00 00 00 00 00 00 00 00 |........
4108: 00 00 00 00 00 00 00 00 |........
4110: 00 00 00 00 00 00 00 00 |........
```

```
3FF8: 00 00 00 00 00 00 00 00 |........
4000: 00 65 65 89 D1 CC 56 41 |.ee‰ŃĚVA
4008: 82 D6 20 74 C2 53 D0 09 |‚Ö tÂSĐ.
4010: 76 3B C9 A3 5E FE 55 DC |v;ÉŁ^ţUÜ
4018: 01 9C A7 19 0F 12 3E E3 |.ś§...>ă
4020: 32 8C E9 F6 EA 8B 29 B8 |2Śéöę‹),
4028: 93 68 74 74 70 3A 2F 2F |“http://
4030: 67 6F 6C 64 65 6E 68 6A |goldenhj
4038: 6E 71 76 63 32 6C 6C 64 |nqvc2lld
4040: 2E 6F 6E 69 6F 6E 2F 72 |.onion/r
4048: 7A 76 6A 34 33 66 57 00 |zvj43fW.
4050: 00 00 00 00 00 00 00 00 |........
4058: 00 00 00 00 00 00 00 00 |........
4060: 00 00 00 00 00 00 00 00 |........
4068: 00 68 74 74 70 3A 2F 2F |.http://
4070: 67 6F 6C 64 65 6E 32 75 |golden2u
4078: 71 70 69 71 63 73 36 6A |qpiqcs6j
4080: 2E 6F 6E 69 6F 6E 2F 72 |.onion/r
4088: 7A 76 6A 34 33 66 57 00 |zvj43fW.
4090: 00 00 00 00 00 00 00 00 |........
4098: 00 00 00 00 00 00 00 00 |........
40A0: 00 00 00 00 00 00 00 00 |........
40A8: 00 72 7A 76 6A 34 33 66 |.rzvj43f
40B0: 57 76 66 35 65 66 33 4E |Wvf5ef3N
40B8: 71 36 57 37 5A 34 5A 62 |q6W7Z4Zb
40C0: 7A 6F 65 68 6D 68 35 35 |zoehmh55
40C8: 37 6D 39 37 32 51 68 4D |7m972QhM
40D0: 65 64 56 48 51 76 4C 79 |edVHQvLy
40D8: 6A 75 33 6D 78 37 42 72 |ju3mx7Br
40E0: 52 62 35 58 70 6B 76 44 |Rb5XpkvD
40E8: 32 31 4A 55 56 77 31 4D |21JUVw1M
40F0: 39 43 73 37 34 41 76 5A |9Cs74AvZ
40F8: 70 7A 47 54 57 70 34 78 |pzGTWp4x
4100: 62 70 39 61 56 71 73 6F |bp9aVqso
4108: 4C 00 00 00 00 00 00 00 |L.......
4110: 00 00 00 00 00 00 00 00 |........
```

The Bootloader

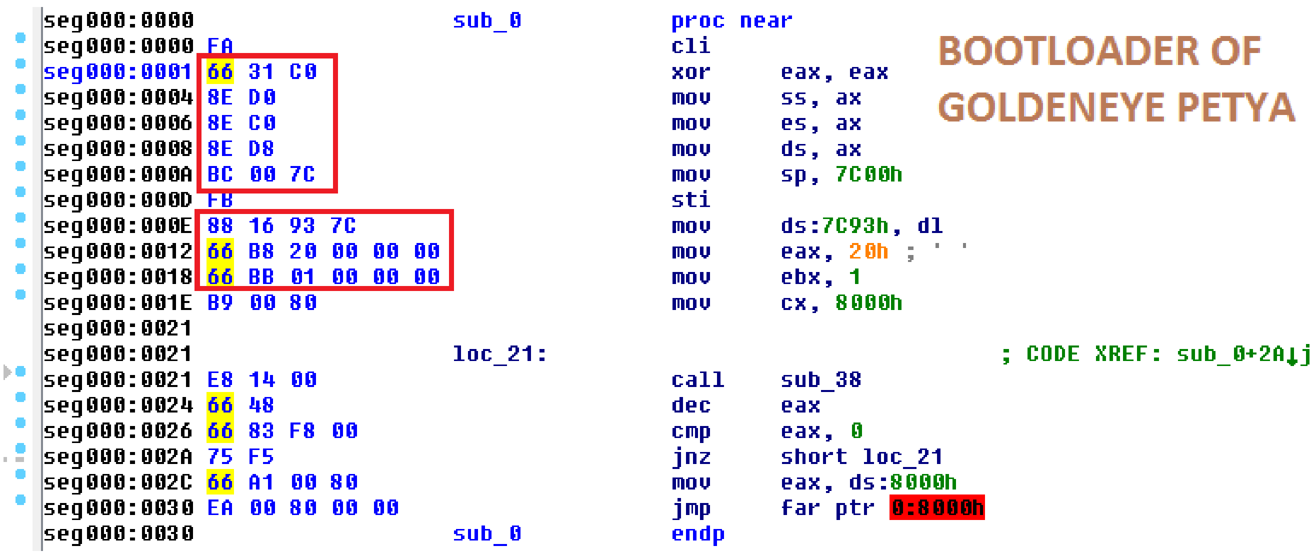
The first thing that struck me as different was the bootloader. Fragment of the hexdump (as before: EternalPetya on the left, and Goldeneye on the right.):

```
0000: FA 31 C0 8E D8 8E D0 8E |ú1ŔŽŘŽĐŽ
0008: C0 8D 26 00 7C FB 66 B8 |ŔŤ&.|űf¸
0010: 20 00 00 00 88 16 93 7C |  ....“|
0018: 66 BB 01 00 00 00 B9 00 |f»....ą.
0020: 80 E8 14 00 66 48 66 83 |€č..fHf
0028: F8 00 75 F5 66 A1 00 80 |ř.uőfˇ.€
0030: EA 00 80 00 00 F4 EB FD |ę.€..ôëý
0038: 66 50 66 31 C0 52 56 57 |fPf1ŔRVW
0040: 66 50 66 53 89 E7 66 50 |fPfS‰çfP
0048: 66 53 06 51 6A 01 6A 10 |fS.Qj.j.
0050: 89 E6 8A 16 93 7C B4 42 |‰ćŠ.“|´B
0058: CD 13 89 FC 66 5B 66 58 |Í.‰üf[fX
0060: 73 08 50 30 E4 CD 13 58 |s.P0äÍ.X
0068: EB D6 66 83 C3 01 66 83 |ëÖfĂ.f
0070: D0 00 81 C1 00 02 73 07 |Đ.Á..s.
0078: 8C C2 80 C6 10 8E C2 5F |ŚÂ€Ć.ŽÂ_
0080: 5E 5A 66 58 C3 60 B4 0E |^ZfXĂ`´.
0088: AC 3C 00 74 04 CD 10 EB |¬<.t.Í.ë
0090: F7 61 C3 00 00 00 00 00 |÷aĂ.....
```

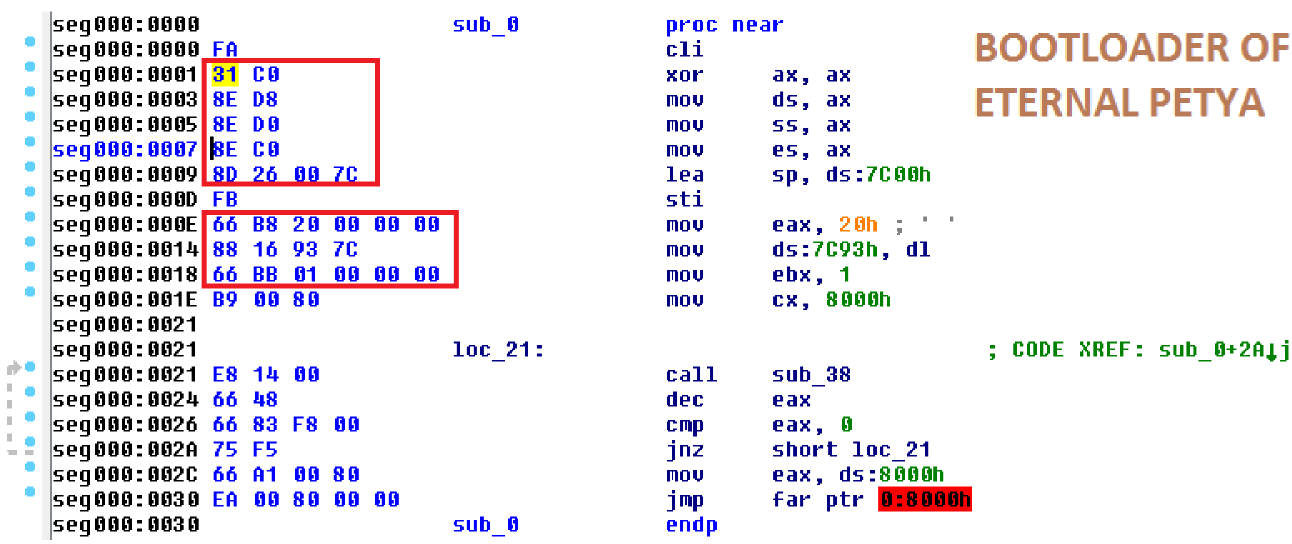
```
0000: FA 66 31 C0 8E D0 8E C0 |úf1ŔŽĐŽŔ
0008: 8E D8 BC 00 7C FB 88 16 |ŽŘĽ.|ű.
0010: 93 7C 66 B8 20 00 00 00 |“|f¸ ...
0018: 66 BB 01 00 00 00 B9 00 |f»....ą.
0020: 80 E8 14 00 66 48 66 83 |€č..fHf
0028: F8 00 75 F5 66 A1 00 80 |ř.uőfˇ.€
0030: EA 00 80 00 00 F4 EB FD |ę.€..ôëý
0038: 66 50 66 31 C0 52 56 57 |fPf1ŔRVW
0040: 66 50 66 53 89 E7 66 50 |fPfS‰çfP
0048: 66 53 06 51 6A 01 6A 10 |fS.Qj.j.
0050: 89 E6 8A 16 93 7C B4 42 |‰ćŠ.“|´B
0058: CD 13 89 FC 66 5B 66 58 |Í.‰üf[fX
0060: 73 08 50 30 E4 CD 13 58 |s.P0äÍ.X
0068: EB D6 66 83 C3 01 66 83 |ëÖfĂ.f
0070: D0 00 81 C1 00 02 73 07 |Đ.Á..s.
0078: 8C C2 80 C6 10 8E C2 5F |ŚÂ€Ć.ŽÂ_
0080: 5E 5A 66 58 C3 60 B4 0E |^ZfXĂ`´.
0088: AC 3C 00 74 04 CD 10 EB |¬<.t.Í.ë
0090: F7 61 C3 00 00 00 00 00 |÷aĂ.....
```

Functionality-wise, it is the same in both cases. It is supposed to read 32 (0x20) sectors from the disk, starting from sector 1, and load them into memory at the address 0x8000. However, the opcodes that are used in both cases to do the same operations are a bit different.

This is the old bootloader, used in Goldeneye:



And this is the bootloader used in the EternalPetya version:



My first impression upon seeing this was that the code was recompiled with different settings, however, another possibility also exists. The total length of the different fragments are the same – so, we cannot exclude the possibility that someone manually edited them inside the pre-compiled binary.

Optimizations – and why it matters

So far we've seen some interesting changes, but they were not enough to prove or disprove whether the code was recompiled. However, the breakthrough in the research may lie in the interesting observation made by David Buchanan.

> The Salsa20 Key expansion was modified using a hexeditor, NOT by modifying the source pic.twitter.com/Q06ZEle8k9
>
> — David Buchanan (@David3141593) June 29, 2017

His theory was based on compiler optimization, which ensures that the same character will not need to be loaded into memory twice. We can see this rule applied in examining the code responsible for storing a string in the memory. Inside of Goldeneye's key expansion function, we can find that this kind of optimization absolutely happens – every character is unique, no character is loaded twice:

```
seg000:96D4                 enter   16h, 0
seg000:96D8                 push    di
seg000:96D9                 push    si
seg000:96DA                 mov     [bp+var_11], 'x'
seg000:96DE                 mov     [bp+var_10], 'p'
seg000:96E2                 mov     [bp+var_F], 'a'
seg000:96E6                 mov     [bp+var_E], 'n'
seg000:96EA                 mov     [bp+var_D], 'd'
seg000:96EE                 mov     [bp+var_B], '3'
seg000:96F2                 mov     [bp+var_A], '2'
seg000:96F6                 mov     [bp+var_9], '-'
seg000:96FA                 mov     [bp+var_8], 'b'
seg000:96FE                 mov     [bp+var_7], 'y'
seg000:9702                 mov     [bp+var_6], 't'
seg000:9706                 mov     al, 'e'
seg000:9708                 mov     [bp+var_12], al
seg000:970B                 mov     [bp+var_5], al
seg000:970E                 mov     al, ' '
seg000:9710                 mov     [bp+var_C], al
seg000:9713                 mov     [bp+var_4], al
seg000:9716                 mov     [bp+var_3], 'k'
seg000:971A                 xor     di, di
```

But in the corresponding fragment of the current kernel, we can find that this rule is broken. The character 'd' repeats and optimization was not applied:

```
seg000:96D4                 enter   16h, 0
seg000:96D8                 push    di
seg000:96D9                 push    si
seg000:96DA                 mov     [bp+var_11], '1' ; -1nvald s3ct-id
seg000:96DE                 mov     [bp+var_10], 'n'
seg000:96E2                 mov     [bp+var_F], 'v'
seg000:96E6                 mov     [bp+var_E], 'a'
seg000:96EA                 mov     [bp+var_D], 'l'
seg000:96EE                 mov     [bp+var_B], 'd'
seg000:96F2                 mov     [bp+var_A], ' '
seg000:96F6                 mov     [bp+var_9], 's'
seg000:96FA                 mov     [bp+var_8], '3'
seg000:96FE                 mov     [bp+var_7], 'c'
seg000:9702                 mov     [bp+var_6], 't'
seg000:9706                 mov     al, '-'
seg000:9708                 mov     [bp+var_12], al
seg000:970B                 mov     [bp+var_5], al
seg000:970E                 mov     al, 'i'
seg000:9710                 mov     [bp+var_C], al
seg000:9713                 mov     [bp+var_4], al
seg000:9716                 mov     [bp+var_3], 'd'
seg000:971A                 xor     di, di
```

If the same code was generated by a compiler, this fragment would look identical to other repeated characters:

```
mov al, 'd'
mov [bp+var_B], al
mov [bp+var_3], al
```

This is a very strong argument against the theory of the code being recompiled. But anyway, let’s continue the analysis and see if we can find even more evidence.

Closer look at the changes

In a previous post I presented a fast comparison of the current kernel vs Goldeneye, done with the help of IDA plugin, BinDiff:

similarity	confide	change	EA primary	name primary	EA secondary
1.00	0.99	-------	000088C4	sub_88C4_13	000888C4
1.00	0.99	-------	00008972	sub_8972_19	00088972
1.00	0.99	-------	0000899A	sub_899A_20	0008899A
1.00	0.99	-------	000089B2	sub_89B2_21	000889B2
1.00	0.99	-------	000089CA	read_input	000889CA
1.00	0.99	-------	00008A64	sub_8A64_23	00088A64
1.00	0.99	-------	00008B9A	sub_8B9A_24	00088B9A
1.00	0.99	-------	00008BF2	sub_8BF2_25	00088BF2
1.00	0.99	-------	00008C98	enc_dec_disk	00088C98
1.00	0.99	-------	00009386	sub_9386_26	00089386
1.00	0.99	-------	00009652	s20_hash	00089652
1.00	0.99	-------	000096D4	s20_expand_key	000896D4
1.00	0.99	-------	00009798	s20_crypt	00089798
1.00	0.99	-------	0000998E	sub_998E_36	0008998E
1.00	0.99	-------	000099FC	sub_99FC_37	000899FC
1.00	0.99	-------	000082A2	sub_82A2_8	000882A2
1.00	0.99	-------	000098D6	sub_98D6_35	000898D6
1.00	0.99	-------	00008FA6	encrypt_mft	00088FA6
1.00	0.99	-------	00008DE2	find_and_encrypt_mft	00088DE2
1.00	0.99	-------	0000811A	fake_chkdsk	0008811A
1.00	0.99	-------	00008212	display_reboot_request	00088212
1.00	0.99	-------	000085CE	screen_output	000885CE
1.00	0.99	-------	00008726	sub_8726_12	00088726
1.00	0.99	-------	00008932	sub_8932_15	00088932
1.00	0.99	-------	00008A54	sub_8A54_22	00088A54
1.00	0.99	-------	00009462	sub_9462_27	00089462
1.00	0.99	-------	0000949A	sub_949A_28	0008949A
1.00	0.99	-------	000095D8	sub_95D8_31	000895D8
1.00	0.99	-------	000095EC	sub_95EC_32	000895EC
1.00	0.99	-------	00009628	s20_rev_little_endian	00089628
1.00	0.99	-------	00009878	sub_9878_33	00089878
1.00	0.99	-------	0000989C	sub_989C_34	0008989C
1.00	0.98	-------	00008684	display_strings	00088684
1.00	0.98	-------	0000891E	sub_891E_14	0008891E
1.00	0.98	-------	00008948	sub_8948_16	00088948
1.00	0.98	-------	00008950	sub_8950_17	00088950
1.00	0.98	-------	0000896A	sub_896A_18	0008896A
1.00	0.98	-------	00008C5A	disk_read_or_write	00088C5A
1.00	0.88	-------	00009518	sub_9518_29	00089518
1.00	0.88	-------	00009578	sub_9578_30	00089578
0.99	0.99	-I--E--	00008426	main_info_screen	00088426
0.16	0.38	GI--EL-	000086E0	sub_86E0_11	000886E0

We can see that significant modifications have been made only in the functions related to displaying the information screen. Let’s check how exactly these changes have been applied.

main_info_screen (offset 0x8426):

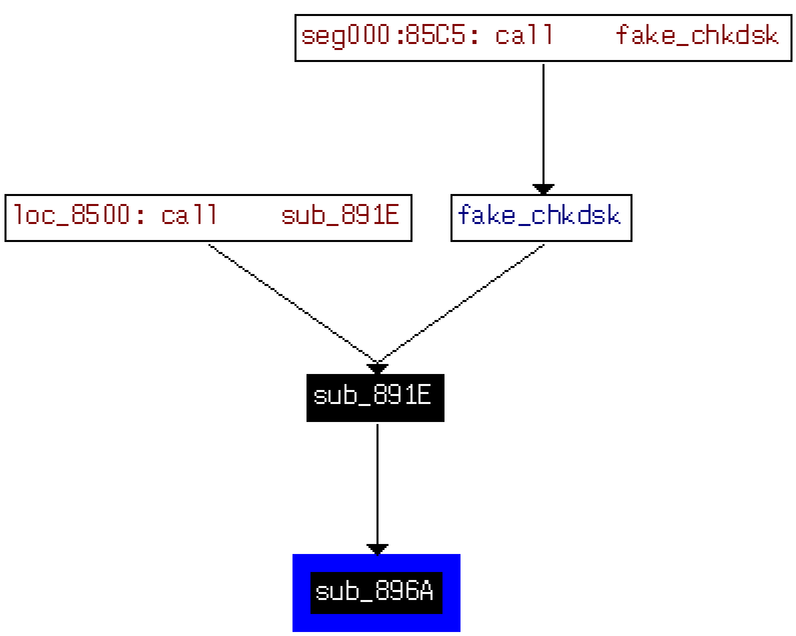
Changes of the main_info_screen pointed out by the BinDiff (left: current, right: Goldeneye):

As we can see, the call to a function at 0x008848E was replaced with NOPs (No Operation). This is a common practice used to remove an unwanted function in case of patching compiled binaries. Yet, sometimes it can be also introduced by #Ifdefs. The rest of the code matches the previous version, even using the same offsets. However, the addresses to the displayed strings are different in both binaries.

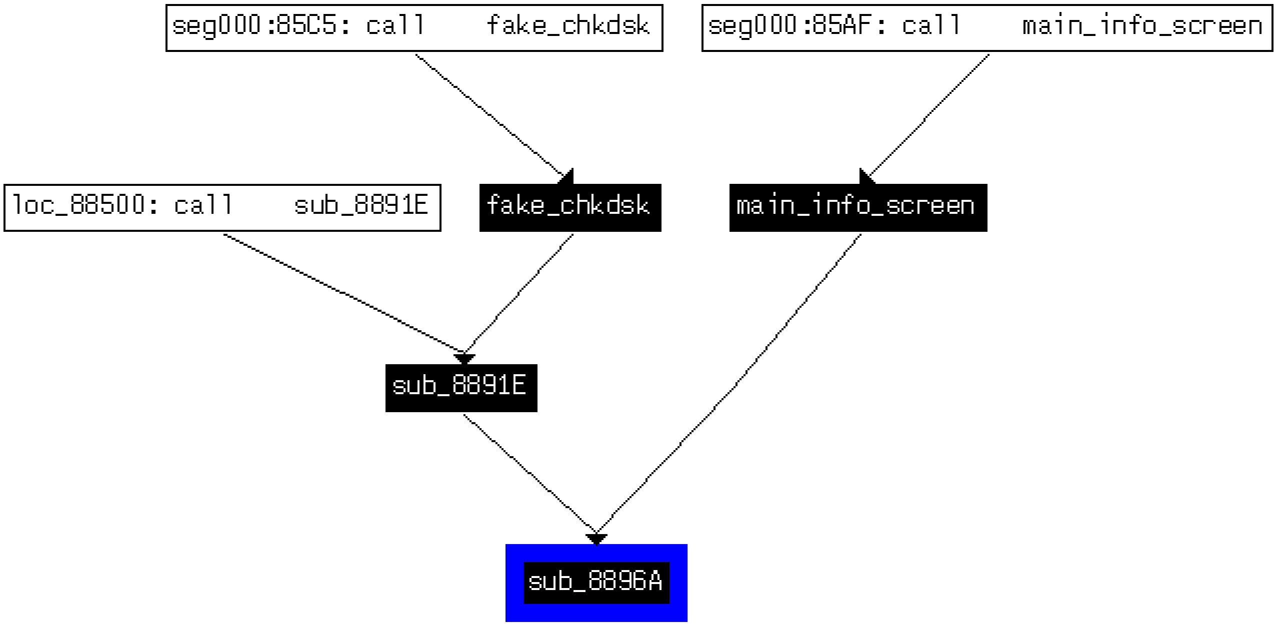
The unreferenced function is still present in the current binary:

```
seg000:896A
seg000:896A                 sub_896A        proc near
seg000:896A 6A 00                           push    0
seg000:896C E8 03 00                        call    read_key
seg000:896F 5B                              pop     bx
seg000:8970 C3                              retn
seg000:8970                 sub_896A        endp
```

…and called in some other places of code:

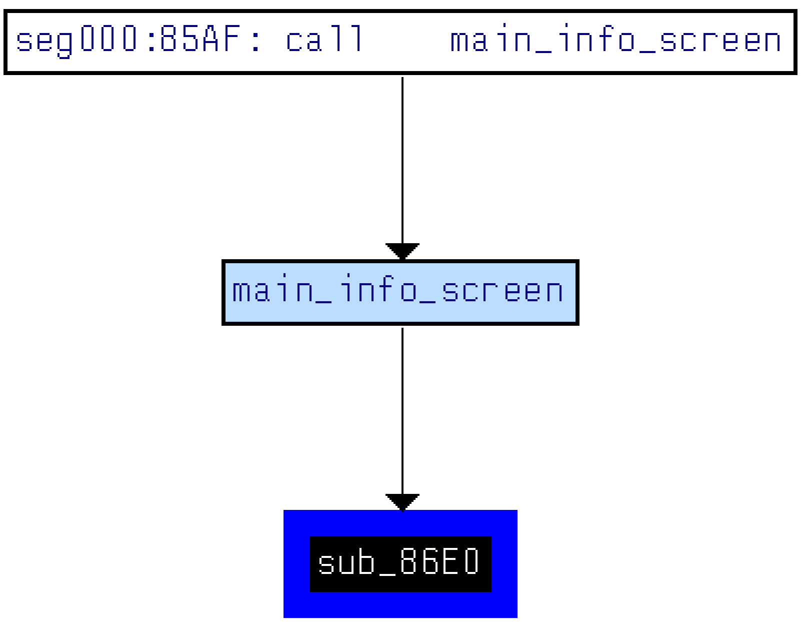


Comparison to the Goldeneye's call graph, it lacks one of the references, but the other ones are consistent:



sub_86E0 (offset 0x86E0):

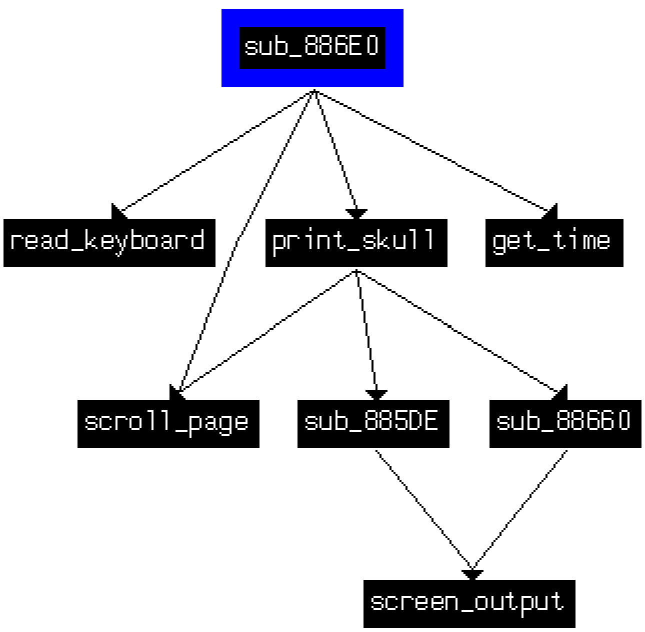
The second change is in another function, that is also a part of the information screen. It is not referenced from any other place in the code:



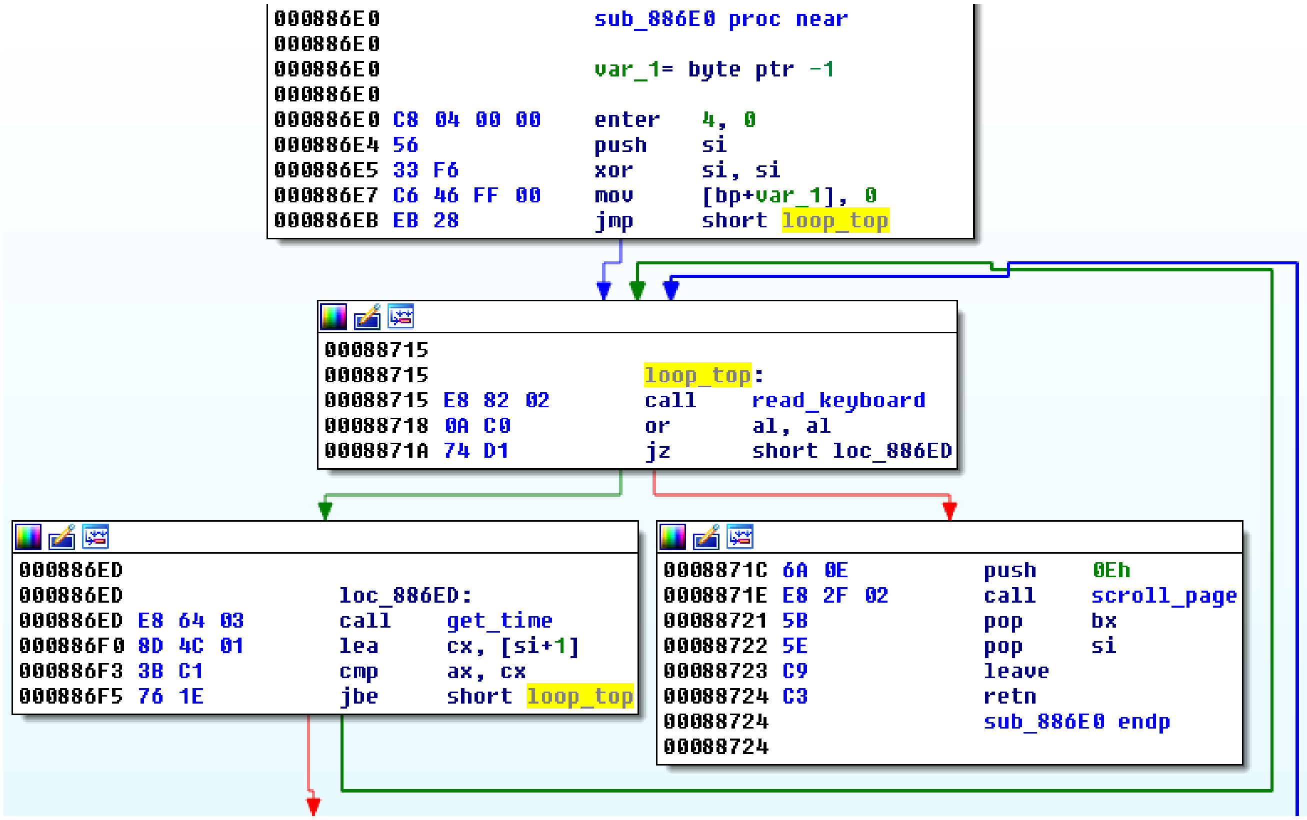
As we can see, it is called at the beginning of the previously discussed function:

```
seg000:8426             main_info_screen proc near
seg000:8426
seg000:8426             var_24C          = byte ptr -24Ch
seg000:8426             var_223          = byte ptr -223h
seg000:8426             var_1E3          = byte ptr -1E3h
seg000:8426             var_1A3          = byte ptr -1A3h
seg000:8426             var_4C           = byte ptr -4Ch
seg000:8426             var_1            = byte ptr -1
seg000:8426             arg_0            = word ptr  4
seg000:8426             arg_2            = byte ptr  6
seg000:8426
seg000:8426 C8 4C 02 00                  enter   24Ch, 0
seg000:842A 57                           push    di
seg000:842B 56                           push    si
seg000:842C E8 B1 02                     call    sub_86E0
```

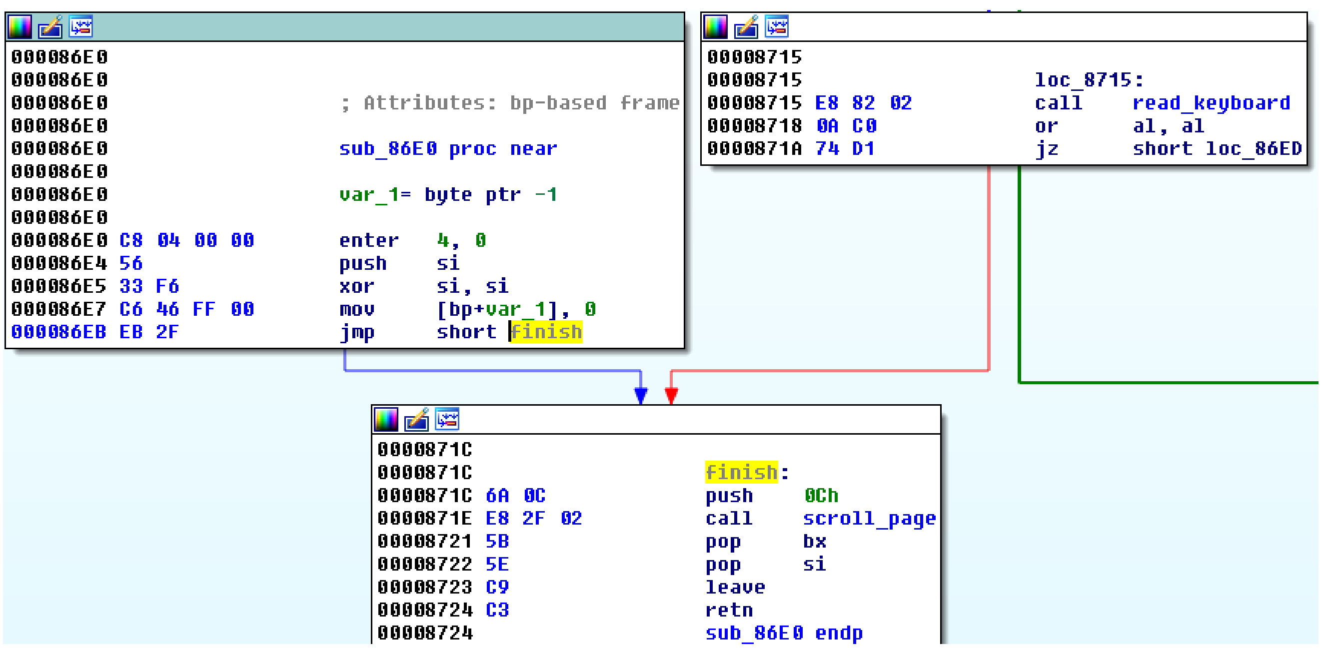
In the Goldeneye kernel, the corresponding function was the one responsible for printing the skull:



The first jump leads to the loop responsible for displaying the skull and waiting for the key to be pressed by the user. Fragment of the code:



Looking inside the EternalPetya code, we are almost sure that this function was patched post-compilation, rather than recompiled. The first jump, that was supposed to lead to the loop leads directly to the function end:



The original code is still in the binary, but it is never referenced (dead code).

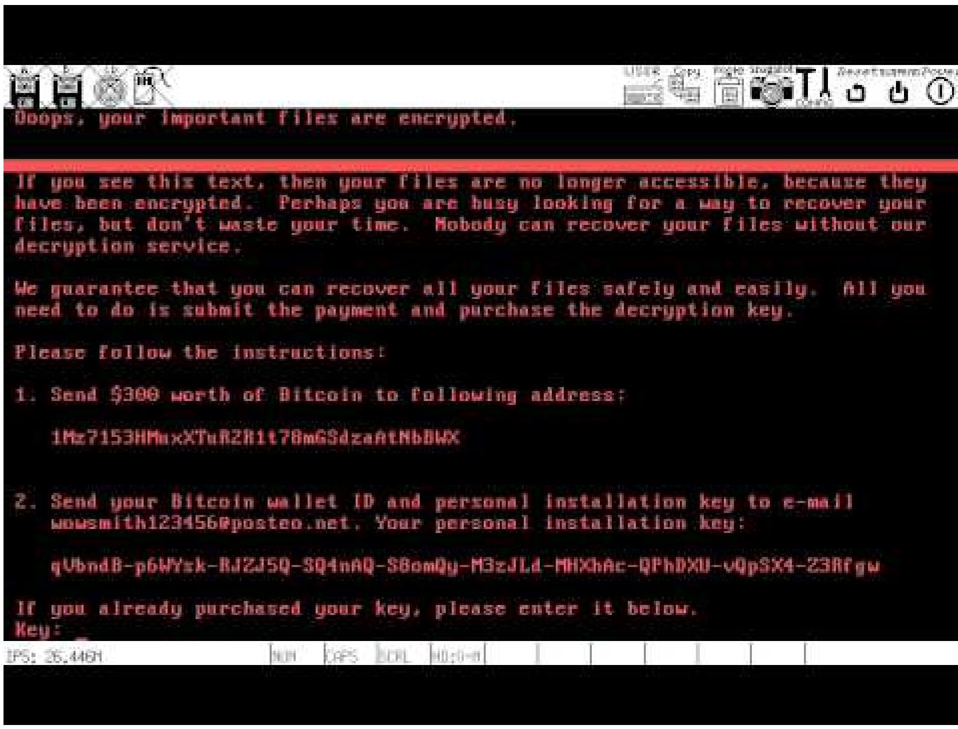
Are the patches reversible?

I thought as a finishing touch of this research it would be interesting to reverse the changes and bring the dead code back to life. As an input, I used the dumped code of:

> EternalPetya kernel + bootloader (f3471d609077479891218b0f93a77ceb).

My version (reverse patch): (7957520271edf003742db63fc250c231).

Indeed, after applying the patches, we are back to seeing the same blinking screen, only the skull is gone (the corresponding strings has been overwritten):



Watch Video At:

https://youtu.be/I3gK3C_0zUs

Conclusions

I think the presented evidence is enough to prove, that the code was not recompiled from the original source (in contrary to what I initially suspected). Thus, the involvement of the original Petya author, Janus, seems unlikely. It seems in this case he was just chosen as a scapegoat by some different actor.

The edits made in the code are well crafted – the person doing them was fluent in assembly and knew exactly what to change and why. Thus, it gave the first impression of very neat and clean modifications, that could possibly be a result of code recompilation. Yet, after doing a deeper analysis, we have identified numerous nuances that show otherwise.

EternalPetya seems to be a patchwork made of code stolen from various sources. In addition to the modified version of the GoldenEye Petya kernel, we can find the leaked NSA exploits from the “Eternal” series as well as legitimate applications, such as PsExec.

It is common practice among unsophisticated actors (script-kiddies) to steal and repurpose someone else's code. However, in this case, the composition was done well by a person or team with good technical knowledge and careful execution. A possible reason for using so many stolen elements, apart from saving actor's time, could have been to throw off any obvious signs of attribution.

There are still many mysteries to solve about this malware which creates many theories that, until proven true, are nothing more than speculation.

Appendix

Read also:

> EternalPetya and the lost Salsa20 key

This video cannot be displayed because your *Functional Cookies* are currently disabled. To enable them, please visit our *privacy policy* and search for the Cookies section. Select *"Click Here"* to open the Privacy Preference Center and select *"Functional Cookies"* in the menu. You can switch the tab back to *"Active"* or disable by moving the tab to *"Inactive."* Click *"Save Settings."*

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.wordp