Uroburos: the snake rootkit

deresz@gmail.com tecamac@gmail.com

March 12, 2014

Abstract

This is a technical analysis of a unique, very powerful and covert remote access backdoor used in targeted attacks. This rootkit has been designed and coded by very skilled and experienced programmers. The techniques used demonstrate their excellent knowledge of Windows kernel internals. The paper describes the use of undocumented and clever tricks performed on the running Windows kernel such as on-the-fly disassembling or PatchGuard bypassing. It details how the covert communication channels work (although this part requires a lot of further analysis) and how the virtual encrypted storage is implemented.

The document has been created to raise awareness - its goal is to enhance understanding of this (and potentially similar) threats as well as to provide means of protection against it. The indicators of compromise (IOCs) are provided to scan for the presence of this malware in computer systems.

Contents

Samples

```
f4f192004df1a4723cb9a8b4a9eb2fbf 32bit driver
ed785bbd156b61553aaf78b6f71fb37b 64bit driver
```

The 32bit and 64bit versions are very similar except some functionalities like the PatchGuard bypassing method that is not necessarily present on the 32bit version. As it is more convenient to reverse, the 32bit version will be presented and we would switch to the 64bit version when necessary.

Yara detection rules

```
rule snake_packed
{
```

```
md5 = "f4f192004df1a4723cb9a8b4a9eb2fbf"
strings:
/*
25 FF FF FE FF    and     eax, 0FFFEFFFFh
0F 22 C0          mov     cr0, eax
C0 E8 ?? ?? 00 00 call    sub_????
*/
$cr0 = { 25 FF FF FE FF 0F 22 C0 E8 ?? ?? 00 00}
condition:
any of them
}

rule snake
{
md5 = "40aa66d9600d82e6c814b5307c137be5"
strings:
$ModuleStart = { 00 4D 6F 64 75 6C 65 53 74 61 72 74 00 }
$ModuleStop = { 00 4D 6F 64 75 6C 65 53 74 6F 70 00}
$firefox = "firefox.exe"
condition:
all of them
}
```

1 Initialization

1.1 Injection

The driver hides its execution by injecting the code into the system process and exiting its initial process.

The execution is manually forked as a system thread in function *fork_and_hide (1D75Ah)*. Specifically the mechanism is similar to a *vfork* on Unix systems; the parent image is suspended until the child image releases the execution thread. It is composed of three steps. (i) The instantiation check verifies that no other rootkit instance is running. This is described in Section 1.2r. (ii) Section 1.3 details the image copy. It makes the code resident in non-paged memory so that it can run as a system thread. (iii) Finally, the control is transferred to the cloned image, in return the child image is hidden by zeroing its headers, see Section 1.4. As a result, the rootkit runs in a kernel thread with code in non-paged memory. Note that this thread has a non existent owning process. This can be used as an IOC; owning process of any thread usually does exist.

1.2 Instantiation check

The presence of another rootkit instance is verified in *reinfection_check (1D460h)*. This function checks against the presence of the of three events. If one is observed the error *STATUS_OBJECT_NAME_EXISTS* is returned.

- pr
- \BaseNamedObjects\{B93DFED5-9A3B-459b-A617-59FD9FAD693E} [1]

[1] This infection marker was observed to be created by a different version of this toolkit aka Agent.BTZ.

Table 1: Indicators of compromise

Location	Type	Value	Comment
Physical Memory	Event	`\BaseNamedObjects\{B93DFED5-9A3B-459b-A617-59FD9FAD693E}`	Infection marker
Physical Memory	Event	`\BaseNamedObjects\shell.{F21EDC09-85D3-4eb9-915F-1AFA2FF28153}`	Infection marker
Physical Memory	Event	`\BaseNamedObjects\wininet_activate`	Related to outbound network flow interception
Physical Memory	Kernel thread	Presence of a kernel thread with non-existent owning process. This is consequence of the kernel injection	
Physical Memory	Interrupt	`0x55`	Custom interrupt used by the hooking engine
Physical Memory	Interrupt	`0xC3`	Custom interrupt used by the hooking engine
Physical Memory	Device	`\Device\RawDisk1`	Device relate to the rootkit's virtual file system
Physical Memory	Device	`\Device\RawDisk2`	Device relate to a second virtual file system that only reside in physical memory and not flushed to file system
Registry	Key	`\Registry\Machine\System\CurrentControlSet\Services\Ultra3`	Rootkit autorun, this IOC can present a false positive; if present the executed file need to be checked
Registry	Key	`\Registry\Machine\System\CurrentControlSet\Services\~ROOT`	Rootkit autorun, this IOC can present a false positive; if present the executed file need to be checked
File system	Filename	`\%SystemRoot%\$NtuninstallQ817473$\hotfix.dat`	Virtual file system, must be checked querying mfc directly because file API can be hooked
File system	Filesize	`104857600`	The default size of the file backing the virtual file system, must be checked querying mfc directly because file API can be hooked
File system	Filename	`\%SystemRoot%\$NtuninstallQ817473$\virtdisk.bin`	Virtual file system, must be checked querying mfc directly because file API can be hooked
File system	Filename	`\%SystemRoot%\$NtuninstallQ817473$\fdisk.sys`	Rootkit, must be checked querying mfc directly because file API can be hooked
File system	Filename	`\%SystemRoot%\$NtuninstallQ817473$\rkng_inst.exe`	Rootkit, must be checked querying mfc directly because file API can be hooked.
File system	Volume	`\\.\Hd1\\`	Volume associated to the virtual file system
File system	Volume	`\\.\Hd2\\`	Volume associated to the virtual file system that only resides in memory
File system	Partition	The partition used as virtual file system by the rootkit has a peculiar setting to make the partition stealthy for most applications. Those settings can be used as IOC see Section 4.2.2	
Network	Named pipe	`\\.\pipe\isapi_dg`	Named pipe used for internal communications
Network	Named pipe	`\\.\pipe\isapi_dg{1-4}`	Named pipes used for internal communications
Network	Named pipe	`\\.\pipe\isapi_http`	Named pipe used for internal communications
Network	Named pipe	`\\.\pipe\isapi_http{1-3}`	Named pipes used for internal communications
Network	Named pipe	`\\.\pipe\services_control`	Named pipe used for internal communications
Network	Named pipe	`\\.\pipe\wininet_activate`	Named pipe used for internal communications
Network	Fingerprint	For SMTP: in addresses like username@domain with N the length of username, the (N-1)th char of the username has ASCII code 97+(sum/26) and the (N)th char ascii code is 122-(sum/26) where sum is the sum of the ASCII codes of the first 8 username chars	
Network	Fingerprint	For base64 HTTP headers: after base64 decoding, the 9th char of the username has ASCII code 97+(sum/26) and the 10th char ASCII code is 122-(sum/26) where sum is the sum of the ASCII codes of the first 8 chars	
Network	Fingerprint	For raw data: after base64 decoding, the 9th char of the username has ASCII code 97+(sum/26) and the 10th char ASCII code is 122-(sum/26) where sum is the sum of the ASCII codes of the first 8 chars	

- \BaseNamedObjects\shell.{F21EDC09-85D3-4eb9-915F-1AFA2FF28153}[2]

Only the later event is created in the case no others are observed. It is believed that the former ones are infection markers of previous versions of the rootkit. It is worth to mention that such infection markers constitute valuable IOCs.

- \BaseNamedObjects\shell.{F21EDC09-85D3-4eb9-915F-1AFA2FF28153}

If no infection marker is observed then the above event is created.

1.3 Image copy and relocations

A fork requires to copy the current driver image. For this, non-paged pool with the tag *NtFs* and a size that fits the rootkit image is allocated. The image is then copied over there. The tag *NtFs* cannot be considered as a relevant IOC as it is associated with Windows *StrucSup.c* which seems responsible for file management.

The main issue with such a copy is that the base address of the image changes and all static references are invalidated. This is cleanly handled via proper relocations defined in the *.reloc* section; the rootkit embeds a relocation routine described in the remainder of this section. Usually, malware handles this by writing to the injected memory, it is much more pragmatic and sustainable to do it via such an appropriate relocation. Indeed, relocations are automatically generated by compilers; reusing this for relocation prevents forgetting to remap newly created static values across development versions.

The function *PE_do_relocation* (5B270h) partially implements the relocation algorithm for portable executables[3]. It implements most of the relocation types but the MIPS and 48bit addresses relocations. This function is detailed in Section B.2

1.4 Hiding the child image

The control is passed to the newly created child image calling its entry point at *1D6D3h*. Upon success, the headers of the child image are zeroed out as on the following listing. This makes the carving of the rootkit's portable executable difficult.

```
loc_1D6EA
  mov     eax, [ebp+size_of_headers]
  add     eax, 1
  mov     [ebp+size_of_headers], eax
loc_1D6F3:
  mov     ecx, [ebp+my_image_copy] ; screw the headers
  mov     edx, [ecx+IMAGE_DOS_HEADER.e_lfanew]
  mov     eax, [ebp+my_image_copy]
  mov     ecx, [eax+edx+IMAGE_NT_HEADERS.OptionalHeader.SizeOfHeaders]
  shr     ecx, 2
  cmp     [ebp+size_of_headers], ecx
  jnb     short loc_1D717
  mov     edx, [ebp+size_of_headers]
  mov     eax, [ebp+my_image_copy]
  mov     dword ptr [eax+edx*4], 0
  jmp     short loc_1D6EA
```

[2]This value is stored encrypted, decryption at *2B550h* consists in a byte per byte xor of memory *8BB88h* and *8BBC8h*. Several other strings are stored the same way, that is xor split into two locations with a *40h* byte offset.

[3]See "Microsoft Portable Executable and Common Object File Format Specification" section 6.6 for a full description

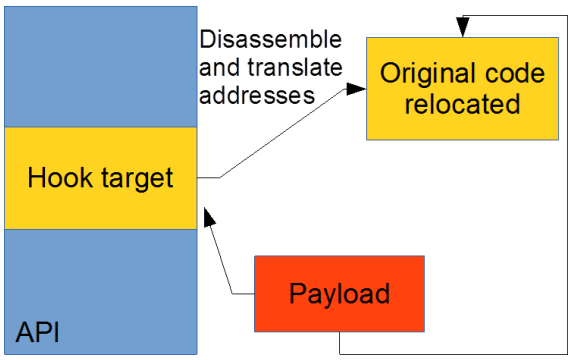

Figure 1: Snake advanced assembly manipulation when relocating the code.

Finally the main thread just returns causing the termination of the process. The rootkit runs in a kernel thread with code in non-paged memory. This method has a limitation: the owning process of this system thread does not exist anymore. This is an anomaly that can be used as a relevant IOC. Indeed, according to the authors experience the owning process of a legitimate system thread usually exists.

1.5 Setup persistence

TODO: describe setup persistence *rootkit_init (15CA0h)*

1.6 Force kernel mode

The rootkit will interact between user-mode and kernel-mode. Such interaction is delicate; in particular several system calls behave slightly differently when called from each mode. On such calls, the system traps the caller and uses the routine *ExGetPreviousMode* to determine whether the parameters are from a user-mode or kernel-mode source.

Managing the situation can become difficult in the current rootkit where user mode modules need to be orchestrated via a messaging service. The authors believe this is the reason why the rootkit authors wrote wrappers for several system calls forcing kernel-mode before their execution.

For example on Windows 2003, the code of *ExGetPreviousMode* is quite simple, it just gets and returns the PreviousMode value

```
.text:0044086C                  mov     eax, large fs:124h
.text:00440872                  mov     al, [eax+0D7h]
.text:00440878                  retn
```

From this, two directives are built which set the previous mode `[fs:124h+0D7h]` appropriately: *set_kernelmode (8E070h)* and *set_usermode (8E074h)*.

The tricky part is that the code may change across Windows versions. The current rootkit implements an easy solution for this issue: it disassembles *ExGetPreviousMode*, copies each instruction to the generated directive until the last return, and replaces `mov al, [eax+0D7h]` by `mov [eax+0D7h] MODE` where the D7 is dynamically computed according to the disassembly process and MODE is the requested mode (see Figure 1).

For example, on Windows 2003 the rootkit generates the following directives:

```
set_kernelmode:
 mov     eax, large fs:124h
 mov     [eax+0D7h] 0
```

```
 retn
set_usermode:
 mov     eax, large fs:124h
 mov     [eax+0D7h] 1
 retn
```

TODO: Describe the process for Windows 7 and 2008 server

The disassembly is achieved with the support of *Udis86*[4]

Before actually doing this there is a last tricky part. The instructions of *ExGetPreviousMode* are in pageable memory. This can cause issues while accessing the buffer, and we do not want to handle page faults. Windows uses MDL[5] to describe the physical page layout for a virtual memory buffer. This layout can be fixed by calling *MmProbeAndLockPages* which makes the pageable memory non-paged until the MDL is unlocked.

The directives are created in *ExGetPreviousMode_patch_setup (1B3B0h)*. The system module *ntoskrnl.exe*, where *ExGetPreviousMode* resides, was saved in the rootkit context at *1D328h* using the *look_module* function described in Section B.3. The system export for *ExGetPreviousMode* is obtained from the custom *getprocaddress* detailed in Section B.4 and its pages are locked in physical memory. The code is disassembled linearly, copying the code to the directive memory and saving second operand values of the last disassembled instruction. When a return instruction is encountered, the preceding instruction is replaced with the following opcodes where the last_op2 is a double word corresponding to the saved second operand value of the instruction proceeding the return instruction.

```
C6h 80h last_op2 00h C3h         // mov byte ptr [eax+last_op2], 0 ; ret
```

A dual code is created for user mode. In user mode, the resulting system call wrappers simply call *set_kernelmode* as prologue and *set_usermode* as epilogue.

2 Hooking

2.1 Hooking engine

The hooking engine is very pragmatic, it relies on a custom interrupt (C3h). When a location is to be hooked, the instructions that cover the first four bytes are relocated to a handler structure. A callback is also set in this structure and the structure is inserted in a table *handler_table (9A75C)* where it is associated to an ID.

The target location is hooked replacing the first four following bytes with `push h; int C3h` where `h` is a handler identifier.

2.1.1 Creation of the interrupt C3h

The routine which sets the interrupt payload, *gate_set (63BA0)*, has two functional modes.

Mode 0 : Set a gate to a routine, the gate and the routine are provided as argument.

Mode 1 : Reset gate to original value, the value is provided as argument.

[4]One of the authors was lucky on this part of the reverse engineering process, recognizing Udis86 by chance. The library is not really linked, but parts of the source code were included. Further information about *Udis86* can be obtain at http://udis86.sourceforge.net/.

[5]Memory Descriptor List

Saving IDT and GDT from all processors. As each processor has its own interrupt table, the main issue is to actually execute the code on the target processor. For this, the routine *launch_on_cpu (1A6A0h)* first locks the execution on the current CPU raising IRQL to DPC level and gets the CPU number via `movzx eax, large byte ptr fs:51h` launched as a DPC on the targeted CPU

In order to collect the IDT and the GDT on each CPU, the code simply uses the later function with `sidt` and `sgdt` based routines, iterating over all processors via a loop on the number of processor: *KeNumberofProcessors*.

Setting the interrupt Under mode 0, the selected gate (C3h) payload is registered, setting the appropriate flags for code execution:

```
mov     ecx, [ebp+routine]
shr     ecx, 10h
mov     [ebp+gate.offset_high], cx
mov     dx, word ptr [ebp+routine]
mov     [ebp+gate.offset_low], dx
mov     al, [ebp+gate.type]
or      al, P_IS_USED   ; set used
mov     [ebp+gate.type], al
mov     cl, [ebp+gate.type]
and     cl, not (DPL_RING_1 or DPL_RING_2) ; set RING 0
mov     [ebp+gate.type], cl
mov     [ebp+gate.segment], 8
mov     dl, [ebp+gate.type]
or      dl, IS_CODE     ; set code
mov     [ebp+gate.type], dl
mov     al, [ebp+gate.type]
and     al, not (ACCESSED or CODE_EXECUTABLE or CODE_CONFORMING)
or      al, CODE_EXECUTABLE or CODE_CONFORMING
mov     [ebp+gate.type], al
mov     [ebp+gate.type_word], 0
mov     cl, [ebp+gate.type]
and     cl, not (S)     ; set data/code
mov     [ebp+gate.type], cl
```

2.1.2 Self-hooking bypass

As presented in the previous section, the rootkit includes a system wide hooking primitive. Such a hooking can be troublesome for some APIs internally used by the rootkit. For example, if accessing registry keys are prevented, then the rootkit itself may have difficulties accessing these registry keys, too. This is the assumed reason why five hooking bypasses are implemented on the following APIs.

- ExAllocatePoolWithTag
- ExFreePoolWithTag
- ZwCreateKey
- ZwSetValueKey
- ZwOpenProcess
- ZwTerminateProcess

The bypass technique heavily relies on the relocation engine described in Section 1.6. The disassembly is not done on the real image of ntoskrnl but on a non paged copy. In the same manner a non paged copy of the KiServiceTable is obtained. One possible reason for this is stability and performance. A second reason is stealthiness, read access to ntosimage is unlikely to be legitimate

Copying ntoskrnl image. *PE_load_ntoskrnl (62220h)* obtains the file path and the base address of *ntoskrnl.exe* via the first module in the structure *SYSTEM_MODULE_INFORMATION* accessed querying *ZwQuerySystemInformation* on class *SystemModuleInformation (1Bh)*. This information is passed down to *PE_load_from_system_file (5C520h)* where the portable executable format is parsed and mapped into memory according to the process described in Section B.1 and relocated according to the algorithm described in Section B.2. Note that the IAT is not fixed.

The resulting copy is saved at *ntoskrnl_local_image (9A750h)*.

Copying the serice table. The function *KiServiceTable_copy 61D40h* creates a copy of the service table using the local copy of ntoskrnl. For this the Service table descriptor (SDT) is obtained via the system call *KeServiceDescriptorTable*, it has the following structure:

```
00000000 SDT             struc ; (sizeof=0x40)
00000000 ntoskrnl        SSDT ?
00000010 used_or_win32k  SSDT ?
00000020 iis_spud        SSDT ?
00000030 unused          SSDT ?
00000040 SDT             ends

00000000 SSDT            struc ; (sizeof=0x10)
00000000 service_table_baseadd dd ?
00000004 counter_table_baseadd dd ?
00000008 service_limit   dd ?
0000000C arg_table       dd ?
00000010 SSDT            ends
```

The first entry of the SDT provides the system service descriptor table (SSDT) for ntoskrnl, and the first field of the later structure provides the base address of the service table in the loaded image of ntoskrnl.

It appears that this method may fail to obtain an address out of the image bound. In this case the code implements fallback cribling the image of ntoskrnl for the following instruction.

```
mov ds:SDT offset // C7h 05h Offset
```

Each word of the image is compared with 5C7h. On a match, the next double word is taken as a KiServiceTable candidate. This is confirmed verifying that the last service is in the image bound.

```
mov     ecx, [ebp+SDT]
mov     edx, [ecx+SDT.ntoskrnl.service_limit]
mov     eax, [ebp+KiServiceTable]
lea     ecx, [eax+edx*4]
mov     edx, ntoskrnl_local_image
add     edx, [ebp+ntoskrnl_imagesize]
cmp     ecx, edx
jnb     short loc_61E37
```

When a sound service table pointer is identified, its counter-part in the local copy of the ntoskrnl image is saved at *KiSeviceTable_copy (9A74Ch)*.

Starting stub relocation. The idea behind the bypass of *ExAllocatePoolWithTag* and *ExFreePoolWithTag* consists of relocating the beginning of the function where the hooking should be applied if the function is hooked. Executing the relocation would then bypass the hook. The length of the hook is hardcoded to six bytes, then the relocation only applies to the six first bytes.

The relocation engine *relocate_code (625E0)* is also based on Udis86, it takes as input the start address and the length of the buffer to disassemble and the relocation offset to apply.

First, the buffer is disassembled linearly from the starting relocation address until the number of disassembled bytes is greater or equal to the requested relocation size. If a return instruction is encountered during this stage, the function exits with a failure error code.

This disassembled stub is relocated during a second stage. The mechanism transforms any relative branch into an absolute branch applying the offset. Concerning absolute branches, the offset is simply applied. In other terms, sequential instructions are just copied as it is, and the following rewriting rules are applied to branching instructions, where `o` is the relocation offset, `a` an offset, `A` an absolute address, `@` the address of the rewritten instruction, `B = @ + a + o` and `C = A + o`.

Unconditional relative jump

```
@:jmp a → push B; ret
```

Unconditional absolute jump

```
@:jmp A → jump C; ret
```

Unconditional absolute call

```
@:call A → call C; ret
```

Conditional jump

```
@:jcc a → jcc +2; push B; ret
```

Note: This translation seems buggy; the authors are not sure about the interpretation of this part of the code. The natural would rather be the following, as `jnc` is the negated conditional jump `jcc`

```
@:jcc a → jnc +6; push B; ret
```

Trap to debugger On Windows XP, the fist byte of the relocated code is set to 8bh. On Windows 2000, the first byte is set to 55h. The authors do not understand this relocation rule.

When the relocation is terminated, the resulting code is finalized with a `push D` where `D` is the is the end of the relocated buffer with the relocation offset applied.

This relocation algorithm is very limited and definitely error prone in some cases like backward jumps or jumps to register values. But considering that this is applied to small pieces of code, this algorithm should work most of the time.

This bypassing mechanism seems to be designed for pre-Windows XP systems. Considering modern PatchGuard, inline hooking may have been removed making this part of the code unnecessary.

Note: the x64 version is simpler, it only does relocation for conditional and unconditional relative jumps, substituting `jmp o` →`jmp ptr [rip + 6 + O]` where `a` is an offset, `o` is the relocation offset and `O = a + o`.

Service gate relocation. Almost the same technique is used for ZwCreateKey, ZwSetValueKey, ZwOpenProcess and ZwTerminateProcess. The main difference is that those API functions are redirections to the global service handler. Here the bypass consists of the relocation of the service routine, bypassing the service manager.

Indeed, those functions have the following form, where `service` is the requested service and `manager` - the global manager.

```
mov     eax, service
lea     edx, [esp+4]
pushf
push    8
call    manager
retn    1Ch
```

The code first parses the function to obtain the `service`. This is achieved in *get_service_id (61BD0h)* where Udis86 is used to search for the pattern `mov eax,`. On the first match, the second operand is decoded and saved as the requested service.

Secondly, the service routine is collected from the copy of the KiServiceTable and the API function address is localized in the copy of the ntoskrnl image.

```
loc_61868:
mov     eax, [ebp+service]
mov     ecx, KiSeviceTable_copy
mov     edx, [ecx+eax*4]
sub     edx, ntoskrnl_imagebase
add     edx, ntoskrnl_local_image
mov     [ebp+routine], edx
```

Finally, the starting stub of the routine is relocated using the same code as for *ExAllocatePoolWithTag* and *ExFreePoolWithTag*.

2.2 Bypass the PatchGuard

This section relies on the 64bit sample.

```
ed785bbd156b61553aaf78b6f71fb37b 64bit driver
```

The x64 version of PatchGuard prevents the installation of the hooking mechanism. As a result, the x64 version of the rootkit implements a PatchGuard bypass. The technique is based on a *KeBugCheckEx* hook similar to the one described in Uninformed[6]as by skape (mmiller@hick.org) and Skywing (Skywing@valhallalegends.com). The idea is to abuse the validation reporting mechanism based on a bug check reported with code 109h.

[6]http://uninformed.org/index.cgi?v=3&a=3&p=17

2.2.1 Cancel PatchGuard notification hooking KeBugCheck

The bypass is fully based on the hooking engine previously described with insertion of an int C3h and the relocation of the overwritten code. Two callbacks are placed, the first one in *RtlContextCapture*. This callback is actually used to hook *KeBugCheck*, of which the code is saved at PatchGuard initialization and overwritten when signaling an integrity issue. This is well described in the following article:

`http://www.codeproject.com/Articles/28318/Bypassing-PatchGuard-3`

The mechanism is simple: check if *RtlContextCapture* was called by PatchGuard notification, checking if the return address is in the first 64h bytes of *KeBugCheck* and that the bug code is 109h.

```
loc_163A1:
mov     rax, cs:KeBugcheckEx_address
cmp     rdx, rax
jb      loc_1648F
add     rax, 64h
cmp     rax, rdx
jb      loc_1648F
cmp     ecx, 109h       ; patchguard reporting code
jnz     loc_1648F
```

If the IRQL is passive then the code is in a worker and it is enough to restore the worker entry point context so that the worker goes back to processing other work items. This is enough to cancel the bug check. We do not have to worry about the context as all is stored in it statically.

Before resetting the worker a *do nothing* DPC is launched. This may be to ensure that the worker has something to process.

```
loc_1640D:
mov     rsi, [rax+r11]        ; worker start address
call    cs:IoGetInitialStack  ; get stack pointer
mov     cl, gs:52h            ; get current CPU
movzx   ebx, cl
mov     rcx, cs:DPC_cpu_table
lea     rdx, do nothing       ; do nothing
mov     rdi, rbx
xor     r8d, r8d              ; no context
mov     rbp, rax
shl     rdi, 6
add     rcx, rdi
call    cs:KeInitializeDpc
mov     r11, cs:DPC_cpu_table
mov     dl, bl
lea     rcx, [rdi+r11]        ; do not change CPU
call    cs:KeSetTargetProcessorDpc
mov     r11, cs:DPC_cpu_table
xor     r8d, r8d
lea     rcx, [rdi+r11]
xor     edx, edx
call    cs:KeInsertQueueDpc
call    cs:KeGetCurrentIrql
test    al, al
jnz     short loc_1648A       ; IRQL < 0 means not in worker
lea     rcx, [rbp-8]          ; shift stack to caller
xor     r8d, r8d
```

```
mov     rdx, rsi
call    reset_worker          ; mov     rsp, rcx
                              ; xchg    r8, rcx
                              ; jmp     rdx
```

2.2.2 Replay PatchGuard DPC hooking KxDispatchInterrupt

There is an "annoying" case, instead of launching *KeBugCheck* the PatchGuard seems to be able to do a direct call at dispatching level. In this case the worker reset is not an alternative.

In this case the bypass technique relies on a second hook whose target is computed by the following DPC. It looks for *gs:20*[7] value on the stack an gets the address above it.

```
void __stdcall sub_160A0(struct _KDPC *, PVOID, PVOID, PVOID)
sub_160A0       proc near
                push    rbx
                sub     rsp, 20h
                mov     rbx, rdx
                call    gs_20             ; get mov rax, gs:20
                mov     r11, rax
                lea     rax, [rsp+28h]
                jmp     short loc_160BC
loc_160B8:
                add     rax, 8            ; inspect the stack
loc_160BC:
                cmp     [rax], r11
                jnz     short loc_160B8
                mov     rax, [rax-8]      ; get the return address
                xor     r8d, r8d
                xor     edx, edx
                movsxd  rcx, dword ptr [rax-4]
                add     rcx, rax
                mov     cs:qword_787E0, rcx
                mov     rcx, rbx
                add     rsp, 20h
                pop     rbx
                jmp     cs:KeSetEvent
```

Being in a DPC, the previous context to inspect is *KiDispatchInterrupt* and indeed we see that gs:20 is saved on the stack by *nt!PsChargeProcessCpuCycles* called at *nt!KxDispatchInterrupt+0xd6*. We can guess that the return address of this call is the target of the hook. At this location the CPU context is captured just before dispatching the DPC.

```
nt!KxDispatchInterrupt:
fffff800`02a8b8cf 65488b1c2520000000 mov   rbx,qword ptr gs:[20h]
fffff800`02a8b8d8 488b7b08           mov     rdi,qword ptr [rbx+8]
fffff800`02a8b8dc 0fae9c24f0000000   stmxcsr dword ptr [rsp+0F0h]
fffff800`02a8b8e4 650fae142580010000 ldmxcsr dword ptr gs:[180h]
fffff800`02a8b8ed f0480fba6b3000     lock bts qword ptr [rbx+30h],0
fffff800`02a8b8f4 7334               jae     nt!KxDispatchInterrupt+0xba
[...]
nt!KxDispatchInterrupt+0xee:
fffff800`02a8b95e 488bd0          mov     rdx,rax
fffff800`02a8b961 488bcb          mov     rcx,rbx
fffff800`02a8b964 e887fc0a00      call    nt!PsChargeProcessCpuCycles
```

[7] The usage of the structure referenced by gs:20 is not well documented. In ntoskrnl.exe we can observe that DPC management seems to massively rely on gs:20 for CPU data and DPC queue management.

The callback of this hook saves the CPU context in a table with one entry per CPU. In other terms, this table then contains, for each CPU, the context of the last DPC.

Let's come back to PatchGuard and *KeBugCheck*. If the IRQL is at dispatch level, the callback restores the context saved in the latter table and returns. This sets the CPU in the exact same context as in the last DPC, that is PatchGuard DPC. At this stage the PatchGuard will again randomly select a notification method and this will loop until a notification via a worker is selected where the above callback cancels the notification.

2.3 Hooking payload

2.3.1 Inline hooking

TODO: describe hook_inline (631a0h)

2.3.2 Device hooking: IofCallDriver

IofCallDriver is hooked with payload *hook_IofCallDriver (164E0h)*. When this routine is called a list of second level payload is traversed, we name this list *HookedDevicesList (82920h)*. This list manipulates IO requests including the rootkit internal requests, e.g. to the rootkit virtual file system described in Section 4.

This second level handler list is manipulated by *hookedDevice_insertHandler (16090h)* and *hookedDevice_removeHandler (16110h)* which respectively, insert into and remove, members from the list. The members have the following structure

```
00000000 hooked_device   struc ; (sizeof=0x10)
00000000 next            dd ?
00000004 deviceObject    dd ?  ; DEVICE_OBJECT*
00000008 handler         dd ?  ; int *(handler)(*_DEVICE_OBJECT, *IRP, int, int;);
0000000C final?          dd ?  ; stub to execute after the hook payload
00000010 hooked_device   ends
```

The payload of *hook_IofCallDriver (164E0h)* has several different behaviors according to the device name.

Null, Beep, tcpip and Nsiproxy. The routine *rootkit_start (1ce40h)* is called. It initiates the rootkit functionalities setting up the messaging system, *init_commchan (15a50h)*, and installs the following hooks *system_hooking (1e5f0h)*. Most of them are related to the hiding of rootkit components. Access control lists seem to be maintained; authorized process IDs are stored at *pid_list (8c99ch)*, a second table (8c998h) is used in the access control but its usage is not fully understood. Finally, a third list *process_name_list (8d94c)* maintains process names.

***ZwQueryKey*, *ZwEnumerateKey*, *ZwCreateKey* and *ZwSaveKey*.** Hide registry keys named Ultra3 or LEGACY_Ultra3. Note Ultra3 is the rootkit service name, LEGACY_Ultra3 may be related to a former version of the rootkit.

***ZwReadFile*.** Hide the content of `\%SystemRoot%\$NtuninstallQ817473$`

***ZwQuerySystemInformation*.** Hide rootkit handles.

***ZwQueryInformationProcess*.** Lie on Data Execution Prevention status.

***ZwClose*.** Launch a rootkit module. The actual module is unknown.

***ZwTerminateProcess*.** If the system is shutting down, cleanly shutdown the rootkit, stopping modules and communication channels.

***ObOpenObjectByName*.** Hide the rootkit's virtual file systems.

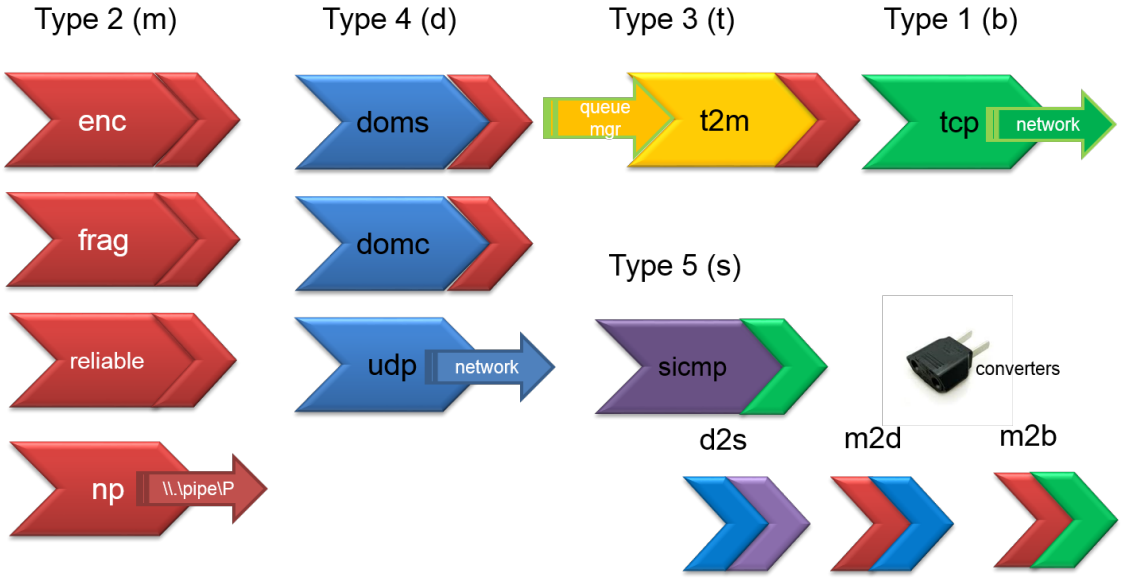

Figure 2: Snake communication objects resembling the building blocks.

netbt and afd. Install Transfer Device Interface hooking on the TCP device. This functionality is a communication dispatcher or a command listener.

TODO describe hook sub_174c0 on device `Filesystem\sr` (driver `\Filesystem\Ntfs`)

3 Communication framework

The communication framework is an essential part of this rootkit - almost every other component is relying on it. It was designed to provide robust remote control over infected machine by providing communication channels between user mode components of the Snake package and the remote operators, with the use of the VFS as a (permanent and volatile) storage mechanism.

3.1 Communication objects

The communication framework relies on ‘communication objects‘ created from a table of ‘object descriptor‘ structures, each composed of a name, type, and several handlers (see Figure 2)

```
00000000 commobj_descriptor        struc ; (sizeof=0x0C)
00000000 name                      char *
00000004 type                      dd
00000008 comm_handlers             (pointers to functions)
```

There are 11 communication objects divided into four types (names are used as such in the rootkit body):

- Type 1: tcp
- Type 2: enc, np, reliable, frag, m2b, m2d
- Type 3: t2m,

- Type 4: udp, doms, domc

The number and meaning of object handlers depend on their type, but they usually implement the following handler table (names of the handlers added by the authors):

```
00000000 comm_handlers   struc ; (sizeof=0x3C)
00000000 length          dd ?
00000004 return_zero     dd ?
00000008 return          dd ?
0000000C constructor     dd ?
00000010 destructor      dd ?
00000014 cleanup         dd ?
00000018 activation      dd ?                    ; init communication
0000001C unknown         dd ?
00000020 start           dd ?                    ; start communication
00000024 parse           dd ?                    ; parse commands
00000028 write           dd ?
0000002C read            dd ?
00000030 print_logs      dd ?
00000034 null            dd ?
00000038 ntstatus        dd ?
0000003C comm_handlers   ends
```

Base objects can be linked together to form a ‘piped structure‘ that we have called ‘communication channel‘ (see Figure 3). Objects are piped by the use of ’.’ (dot), for example: ‘domc.np‘ or ‘enc.reliable.doms.np‘. A given communication object can only support objects of certain type to be linked to, for example ’domc’ and ’doms’ objects can only link to an object of type 2.

The semantics of the different objects are not fully understood at the time of this writing. Here are the ones which have been fully or partially analyzed:

- tcp - TCP stream communication provider
- np - named pipe provider
- domc - client mode interface over type 2 object (for example named pipe client or encrypted named pipe client)
- doms - server mode interface over type 2 object (for example named pipe server or encrypted named pipe server)
- udp - UDP datagram communication provider
- enc - encryption layer over type 2 object
- reliable - implement reliable channel over type 2 object
- frag - fragments reassembling (?) over type 2 object
- m2b - bridge between type 2 and type 1 objects
- m2d - bridge between type 2 and type 4 objects
- t2m - bridge between type 3 and type 2 objects

Each object handles a specific task, for example: the first layer will handle decryption and scheduling and the second object handles the data layer; reads, writes to named pipes for example. ’domc’ and ’doms’ are objects that provide respectively a client and a server interface for bi-directional primitive objects such as ’np’ or ’enc.np’.

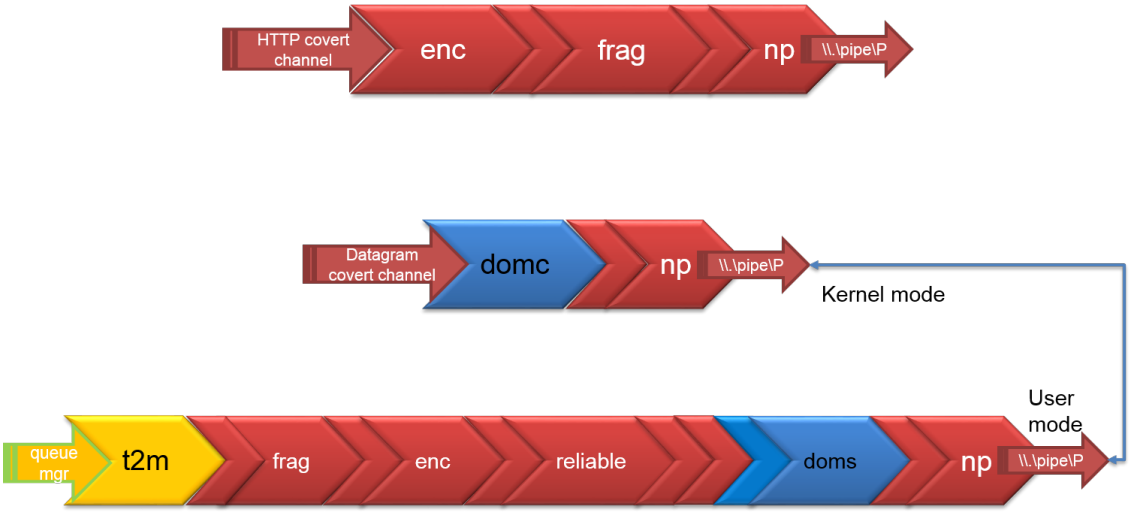

Figure 3: Snake communication channels are built from communication object used as building blocks.

3.2 Communication channels

To use the communication objects, communication channel structures are being used. They are created when the objects are being instantiated, and they contain all the data necessary to use them such as sub-channels, addresses, UNC names, encryption keys among others. This structure is of variable size depending on the object type and has a pointer to a communication object instance structure (comm_obj_inst_t2) at offset 0x0, and parameters of the channel at higher offsets. All the members except the pointer to a communication object instance are object dependent. The below describes an example of how a hardcoded channel 'domc.np', which implements a simple named pipe client, is designed. The leftmost object for this channel is 'domc', here is the corresponding channel structure:

```
struct comm_channel_domc //size=0x30
{
     comm_obj_inst *pcomm_obj;
     comm_channel_np *psub_channel_s;
     comm_channel_np *psub_channel_c;
     _DWORD dwordc
     _DWORD dword10;
     PRKMUTEX prkmutex14;
     PRKMUTEX prkmutex18;
     char *sub_obj_name;
     _DWORD dword20;
     _DWORD dword24;
     _DWORD dword28;
     _DWORD dword2C;
};
```

In this case at offset 0x8 we see the pointer to another comm_channel structure[8], which is a sub channel to this comm_channel. In case of 'domc.np' object, the link would lead to a sub-channel created from the 'np' object (comm_channel_np structure at offset 0x8):

```
struct comm_channel_np //size=0x3c
{
  comm_obj_inst_t2 *pcomm_obj;
  UNICODE_STRING chan_parameters;
  UNICODE_STRING unc_name;
  int client_np_handle;
  int field_18;
  int server_np_handle;
  PRKMUTEX server_np_handle_mutex;
  int status;
  int field_28;
  int field_2C;
  int field_30;
  int field_34;
  int field_38;
};
```

This time at offset 0xC we can see the string representing the UNC name of the named pipe used by this communication channel, for example `\Device\NamedPipe\isapi_dg`, and at offset 0x14 a handle to an open named pipe object. Another example: enc object contains an encryption key at offset 0xc of its communication channel structure (to be included in the next versions of the document).

comm_obj_inst structure contains, as a first member, a pointer to the handler table described above, and a bunch of synchronization primitives, possibly for controlling access to the instance by multiple threads. This structure represents an instance of a single communication object of type 2:

```
struct comm_obj_inst_t2
{
  comm_handlers *phandlers;
  void *field_4;
  int field_8;
  int connection_string;
  synchro *psynchro;
  int unknown;
  PRKMUTEX pmutex1;
  PRKMUTEX pmutex2;
};

struct synchro
{
  _KSEMAPHORE semaphore_1;
  _KSEMAPHORE semaphore_2;
  _KMUTANT mutex_1;
};
```

Each object type has its own initialization function that calls a constructor stored in the third handler table field. For example, type 1 objects use the initialization function sub_73870. All the created communication channels are inserted into a 64-slot hash table

[8] Actually at offset 0x4 there is a second communication sub-channel created when the subsequent type 2 object implements a server - accepts connections from multiple sources.

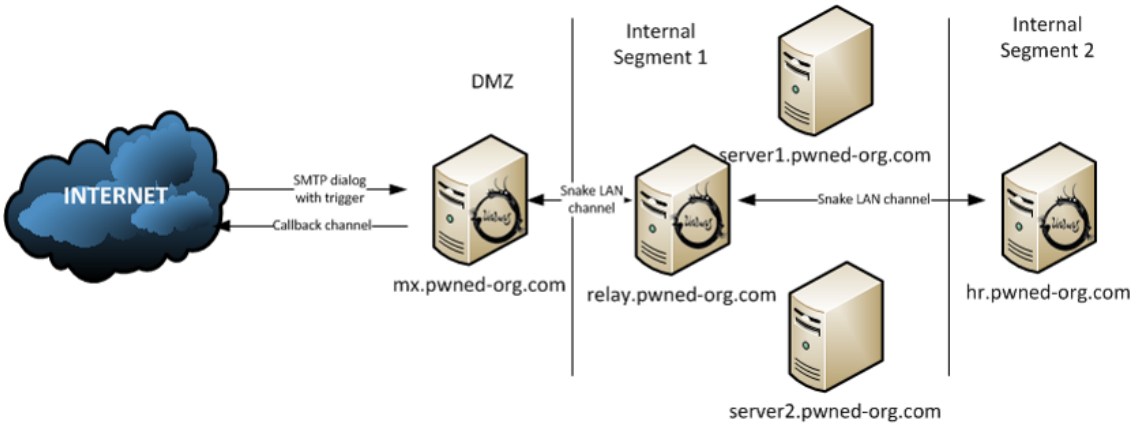

Figure 4: The example architecture of a compromised network. One type of covert channels are being used for LAN trafic between different Snake instances whereas different ones are used for Internet traffic. Note: the authors based this diagram solely on reverse engineering and had no insight into a compromised network.

referenced at 0x8d7f8h via the routine sub_6cd00. Each inserted channel is indexed by a hash computed on the object address and the type of its leftmost object, possibly for performance reasons. If a collision occurs, conflicting channels are inserted on the list with the same index.

3.3 Hardcoded channels

Five hardcoded communication channels - named pipe clients, all using object 'domc.np' (string hardcoded at 0x8ad08), are initialized upon rootkit start. The named pipe UNC names used for the 'np' sub-objects of this objects are the following:

```
\Device\NamedPipe\isapi_dg
\Device\NamedPipe\isapi_dg1
\Device\NamedPipe\isapi_dg2
\Device\NamedPipe\isapi_dg3
\Device\NamedPipe\isapi_dg4
```

The pointers to five instances of 'domc.np' objects are located in a table at 0x8d724. These objects are initialized upon driver startup at sub_4ee40 and the I/O communication loop sub_4fd80 is launched. The loop will periodically try to initiate communication by calling appropriate object handlers on named pipe objects. This loop seems to be implementing a tunnel between the named pipe to which a user mode process, that implements the server, is connected, and a remote end via a low level NDIS traffic that can be of type ICMP, UDP or TCP, which is implemented in sub_4fea0. These mechanisms are probably used for local LAN communication between different instances of the rootkit on the compromised network (see Figure 4)

Two levels of processing are implemented in two separate threads using two linked lists: one at 0x8d748 and another at 0x8d738. When the NDIS traffic is arrived, provided the appropriate trigger is satisfied, the tainted traffic will be routed to one of the isapi_dg pipes, where the userland component will be waiting to capture it (TODO: userland component analysis needed). Other listeners are also initialized: SMTP triggered (routed to userland via isapi_log pipe), and HTTP triggered (routed to userland via isapi_http pipe). See also 3.8 for the description of these triggers.

3.4 Channels initialized from the queue

Other communication channels are initialized from the bootstrap queue: several instances of 'frag.np' and 'enc.reliable.doms.np'. See the bootstrap queue analysis for more information (TODO)

3.5 Low level communications(TODO)

The rootkit implements a low-level communication channel using the ether type 0x7ff. This ether type is not referenced by RFC. The messages from this channel are handled differently via the worker sub_4fea0. The buffer queues are skipped in this case.

3.6 Command listener via TDI hooking

The Transfert Device Interface, a.k.a. TDI, is hooked on device `\Device\Tcp`. The hooking mainly consist in intercepting the traffic tainted by the malware on packet reception. Thus this code can be considered as a listener for command and control.

For this context, the malware implements inline hooking, inserting its handlers in the TDI processing of IRP. The major IRPs are handled in *hook_tcp (52ce0)*. This payload is inserted via *hookedDevice_insertHandler (16090h)* presented in Section 2.3.2 on the device hooking mechanism.

IRP_MJ_CREATE. Insert the pid to the list of monitored processes and create a packet queue.

IRP_MJ_DEVICE_CONTROL. Convert into IRP_MJ_INTERNAL_DEVICE_CONTROL and transfer the control over the corresponding branch.

IRP_MJ_INTERNAL_DEVICE_CONTROL. Look for the IRP minor function in the handled list of minor functions *hook_tcp_tdi_function (8ac88h)*. On match the corresponding handler in *hook_tcp_tdi_handler (8ac8ch)* is called on the IRP, the IRP's stack location and a reference to a handler that will be called at completion. This list of handlers is described below.

IRP_MJ_CLOSE. Close the packet queue.

IRP_MJ_DEFAULT. Nothing particular occured.

All major functions handlers are terminated by *hook_tcp_finalize (16140h)*, which completes the requests.

The following list describes the handlers for each minor function, i.e. the TDI events, processed by the rootkit.

TDI_ASSOCIATE_ADDRESS (52fa0h). Get the reference associated to the object in the IRP parameters and populate the reference in an internal list located at IRP_device_list (8d788h). In other terms, this handle maintains a list of file references corresponding to the different TCP connections.

TDI_DISSOCIATE_ADDRESS (54050h). Remove the entry from IRP_device_list (8d788h), this is clean-up.

TDI_CONNECT (540b0h). Reassemble the message and set post processing hook_tcp_process_message (542a0). The post processing shows 5 message types: :RESET, :CANCEL, :TIMEOUT, :UNREACH, :SUCCESS and an unknown type where a value is printed. Those messages are logged via the function *log_message (54600h)* and then committed to a queue located at *tcp_hook_message_queue (8d7d0h)*

TDI_DISCONNECT (546e0h) Push IRP AuxiliaryBuffer in a chained list for post processing.

TDI_SEND (548b0h) If the current process name is a browser, the following event is created `\\BaseNamedObjects\\wininet_activate`. This event is then pulsed and closed. There are no other reference to this event in the malware code. Possibilities include handling via a module or obsolete code.

TDI_RECEIVE (43d50h) Set the tainted traffic interception routine, *tainted_traffic_interception_0 (43de0h)*, as a final handler called at IRP completion. The authors assume that this final handler will be called when TDI terminates the handling of the current tcp packet.

TDI_SET_EVENT_HANDLER (54770h) Look for the event code in the list *hook_tcp_tdi_event (8acc8h)* then register the associated handler from *hook_tcp_tdi_handler (8accch)* in *IO_STACK_LOCATION.Parameters*

The following list describes the TDI event handlers defined by the malware.

ClientEventConnect (55210h) Log the connection via the routines *logger_2 (54600h)* and *logger_commit (56eb0h)*, retrieve the internal *tcp_hook_stuct* corresponding to the connection and set the transport address in this entry.

ClientEventReceive (554e0h)* and *ClientEvenChainedReceive (55700h) Call the traffic interception routine, *tainted_traffic_interception (55a00h)*, on the received data.

ClientEventReceiveExpedited (556b0h)* and *ClientEventChainedReceiveExpedited (55700h) Call the handler *TDI_REQUEST_KERNEL_SET_EVENT.EventHandler* on the received data.

ClientEventChainedReceiveExpedited (55440h) Release the internal *tpc_hook_struct* and call the *TDI_REQUEST_KERNEL_SET_EVENT.EventHandler* on the disconnection data.

3.7 Tainted traffic processing

The rootkit is designed to watch for door knocking on a server where Internet-facing services are being used. The trigger can be delivered hidden in different places of different network protocols (see 3.8) depending on rootkit's configuration and victim's server type (HTTP - 3 different possibilities, SMTP, TCP, IP, ICMP, DNS, Ethernet), but they all have one thing in common - the trigger mechanism. There are actually two trigger mechanisms: the DEADBEAF-style check and the Checksum. The DEADBEAF-style also contains the checksum calculation as one of its calculation stages.

3.7.1 The Checksum

The checksum is being calculated on 8 characters at a given offset and is compared with the following 2 characters, so the trigger signature is 10 characters long in total. Characters must be ASCII-printable (>32 and <128). The ord of the 8 characters are summed up (sum) and are being compared as follows:

```
signature[9] == sum / 26 + 65
signature[10] == 122 - sum % 26
```

If the match occurs, the traffic is processed as tainted. Example usage: user part of an e-mail address, see 3.8: "Mail server".

3.7.2 DEADBEAF-style

Similar to Checksum but some more processing is involved, as this encoding can also carry a data message in addition to being only a signature.

The signature contains the 10 bytes of checksum but it is being followed by a base64-encoded content. First, the Checksum signature is being calculated as described above, then the base64-decoded content is being base64-decoded. The base64-decoded content is then being XOR-ed with the raw buffer starting from offset 0x0, so at the first byte of the signature. First DWORD of such a decrypted buffer is being compared with the value located at 0x7945c (hardcoded to 0xDEADBEAF but was observed to being changed to an arbitrary value during the boostrap queue loading). If there is a match, the traffic is considered tainted. For some covert channels the payload can also carry a data message. A data portion of the message is located at offset 0xA, directly after the 10-byte signature (after de-XORing). Example usage of this trigger: the value of an HTTP header, see 3.8: "Web server or web proxy".

The checks for tainted traffic are perform on raw TCP stream (sub_49620), HTTP and SMTP (see also 3.8). From this handling a queue of work item is built via sub_49F80. TODO: describe tainted_traffic_interception_0 (43de0h).

3.8 Network footprint

It is likely that a communication mechanism is in fact a sort of proxy to pass traffic from the external world to named pipes where user land component can be attached. Also, named pipes can be potentially used to route internal ('lateral') traffic, e.g. across the segments on the infected LAN. Most of these channels are created in a way that IDS signatures would be difficult to construct, for others a prototype is proposed below. Traffic inspection is split into two groups. First group is hooked via TCP traffic signature matching performed at sub_49580 and obtained by hooking `\Device\Tcp`.

WARNING: The below signatures are PROTOTYPES - and HAVE NOT BEEN TESTED

1. TCP. Can be installed on a server with any kind of service. Signature matching: DEADBEAF-style at offset 0x0. Encrypted and encoded signature is 22-byte long consisting of alphanumeric characters 0x2b-0x7a. Plain snort signature would be difficult to create. Traffic is passed to 'frag.np' at isapi_http named pipe (frag object cuts data into smaller pieces before sending it to the named pipe, currently the reason is unknown).

2. Web server - data passed in HTTP GET request URL parameters Can be installed on a web server. Signature check type: Checksum. It is contained between "?" and first "&" in the URL. It is 10 bytes, 8 arbitrary bytes and 2 bytes of checksum (this checks are also a part of DEADBEAF-style signature). IDS signature is difficult to create in this case. Traffic is passed to 'np' at isapi_log named pipe.

3. Web server or web proxy - data passed in HTTP headers. Can be installed on a proxy server. Signature is a DEADBEAF-style signature contained after ":" in the first HTTP-header (just after the request line). Connection is logged to 'np' at isapi_log named pipe by sending the following string: $IP_address + newline (including the dollar sign, IP_address replaced by textual form of IP address of the remote end), then the rest of the data is getting decoded using the DEADBEAF-style decoding. The decoded data is being sent to 'frag.np' or 'enc.frag.np' at isapi_http[n] where n is encoded in a first DWORD in the decoded buffer (see sub_6c000). Plain snort signature would be difficult to create. The example traffic with trigger could be the following:

```
GET / HTTP/1.1
SomeHeader: YtrEgHjkLoYmFzZTY0ZW5jb2RlZCBzdHJpbmcKYmFz
```

4. Web server special - data passed in HTTP request local file part of URL. This is triggered by comparing the local file part of the URL with a string defined at 0x8d680 (initialized by the rootkit but zeroed on my memory dump). Pipe name where the traffic is passed to, is also computed at runtime (type 'np'). It is possible that the rootkit with specific configurations writes an arbitrary value here.

5. Mail server. Signature check type: Checksum. The signature is located after RCPT TO: and ends by @ (so this is the user part of a recipient e-mail address). The example traffic with trigger could be the following:

```
HELO whatever.com
250 Hello whatever.com, I am glad to meet you
MAIL FROM: <you.bet@you.are.not>
250 OK
RCPT TO: <trueburger@pwned-org.com>
250 OK
354 End data with <CR><LF>.<CR><LF>
<commands>
```

   Building snort signature would be questionable. The whole traffic is passed to 'np' at isapi_log. First the IP of the remote end is logged as $IP_address + newline, then all lines of the SMTP message body are passed to the pipe. The user land component will then receive this transmission and interpret it. There are few commands encoded by a first character of the sent buffer that are parsed in sub_200081e2 in the main userland module:

   - & - read and send a slot from a local queue
   - ! - adjust privileges of a current process
   - : - execute a file (file name follows the command)
   - - re-parse the local queue

- no_command - read and send a file (file name is the buffer)

Second group is packed-based matching obtained by registering an NDIS protocol (mechanism described in previous subsections), checks performed at sub_4F1B0. These can be processed by 'domc.np' object and send to isapi_dg (depending on the configuration state it could possibly be different).

1. Raw Ethernet. Traffic is identified by a custom ethertype 0x7FF (whereas the IP protocol is identified by 0x800)
2. Raw ICMP. DEADBEAF signature is checked at offset 0x8 of ICMP packet.

```
alert icmp any any -> any any (msg:"Snake ICMP"; \
       offset: 0x8 ; pcre:"/^[+-z]{22}/";)
```

3. Raw TCP. DEADBEAF signature at offset 0x0 of the TCP stream.
4. Raw UDP. Must be port 53, DEADBEAF signature at offset 0x0 of UDP data, so this can be handy to be installed on a DNS server:

```
alert udp any any -> any 53 (msg:"Snake UDP 53"; \
       pcre:"/^[+-z]{22}/";)
```

5. RAW IP. DEADBEAF signature at offset 0x0 of IP packet data:

```
alert ip any any -> any any (msg:"Snake IP"; \
       pcre:"/^[+-z]{22}/";)
```

Interesting typo is contained in one of the SMTP messages. "Transmittion" is a mistake that an English native speaker would rather not do.

4 Virtual file systems

The rootkit uses two virtual file systems. One is persistent over reboot and NTFS formatted. The other one volatile and FAT formated; its content is never flushed to a real file system, as a result its content is only accessible on a live infected machine. Both are encrypted with a CAST like algorithm. The filesystem clusters are decrypted on access via cache management. As a result the file systems do no appear in clear text even in the physical memory (see Figure 5).

4.1 Description

The virtual file system is set-up in *vfs_init (32230h)*. As initialization, a chained list *vfs_IO (8e3a0h)* is initialized and associated to a lock *vfs_IO_lock (8e384)*. This list is handled via *ExfInterlockedInsertTailList* and *ExfInterlockedRemoveHeadList* so that the code can run at any IRQL. Then an event *vfs_up (8e3a8h)* is create to indicate the status of the virtual file system service.

Two files system are setup, the first one is persistent over reboot and backed by a file on the filesystem. The second one is volatile and only resides in memory, it is lost on rootkit restart or reboot.

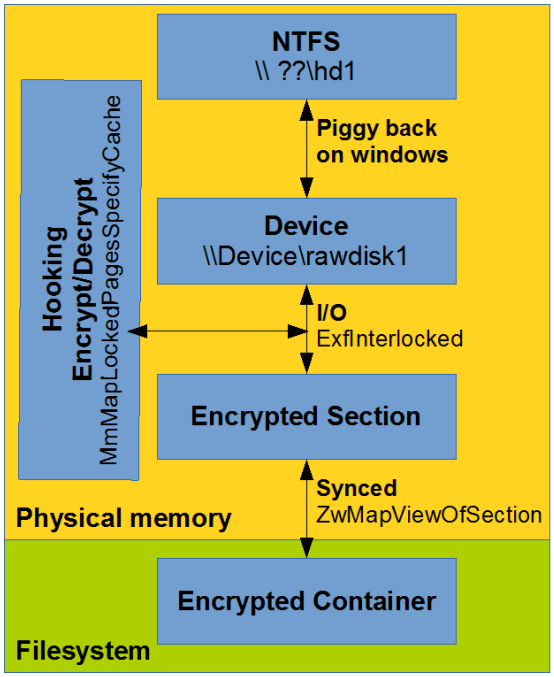

Figure 5: Snake Virtual File System architecture

4.2 Persistent file system

The persistent file system is setup in two steps: (i) setup a memory section backed by a file, (ii) provide access to the section via a device hooked with encryption/decryption primitive on access, (iii) mount the device as an NTFS filesystem. This operation is detailed below.

4.2.1 Setup a section backed by a file.

The first called routine *vfs_map_raw (339e0h)* map in memory the file that will back the virtual file system. This file is located at the following path.

```
\\SystemRoot\\$NtuninstallQ817473$\\hotfix.dat
```

The file is created with the following options meaning that the file is not a directory, it is not cached, all operation are performed synchronously and no read ahead can be achieved. In other terms the file is accessed cluster per cluster and all changes are committed immediately with no caching mechanism. This configuration has limited performance but it is very reliable.

```
FILE_NO_INTERMEDIATE_BUFFERING
FILE_SYNCHRONOUS_IO_NONALERT
FILE_NON_DIRECTORY_FILE
FILE_RANDOM_ACCESS
```

The second noticeable parameter is the allocation size of 640000h (104857600d) which can be considered as an indicator of compromise.

Then this file is mapped to a memory section. The section has read/write permission and it is *SEC_COMMIT* and *SEC_NOCACHE*, meaning that the section is backed by the previously created file via the operating system paging mechanism with no caching. In other terms all modification to the section are directly mirrored to the backing file. Finally this section is mapped to virtual memory via *ZwMapViewOfSection*.

4.2.2 Bind the section with a device.

A device named *\Device\RawDisk1* is created and hooked via the generic device hooking engine described in Section 2.3.2. The handler is located at *327c0h* and we name it *vfs_resident_handler*.

The handler behave differently according to the IRP function.

IRP_MJ_CREATE, IRP_MJ_CLOSE: The IO status of the IRP is set to 1. Then the request is completed.

IRP_MJ_READ, IRP_MJ_WRITE: The IRP *AuxiliaryBuffer* is inserted in the VFS IO list: *vfs_IO (8e3a0h)*.

IRP_MJ_DEVICE_CONTROL: Several behaviors are observed according to the control code.

IOCTL_DISK_GET_DRIVE_GEOMETRY 70000h: The following structure is filled.

```
typedef struct _DISK_GEOMETRY {
  LARGE_INTEGER Cylinders;
  MEDIA_TYPE    MediaType;
  DWORD         TracksPerCylinder;
  DWORD         SectorsPerTrack;
  DWORD         BytesPerSector;
} DISK_GEOMETRY;
```

The number of cylinders is computed dividing the disk size accordingly; with 2 tracks per cylinder, 20h sector per track and 200h byte per cluster. The media type is FixedMedia (0ch) i.e. a fixed hard disk media.

IOCTL_DISK_VERIFY 700014h: Set the input extend length as IOStatus.

IOCTL_DISK_GET_PARTITION_INFO_EX 70048: The following structure is returned.

```
typedef struct _PARTITION_INFORMATION_EX {
  PARTITION_STYLE PartitionStyle;
  LARGE_INTEGER   StartingOffset;
  LARGE_INTEGER   PartitionLength;
  ULONG           PartitionNumber;
  BOOLEAN         RewritePartition;
  union {
    PARTITION_INFORMATION_MBR Mbr;
    PARTITION_INFORMATION_GPT Gpt;
  };
} PARTITION_INFORMATION_EX;
```

It indicates that the partition starts at offset 0x200 and the size is stored at 8e388h for the first persistent partition and at 8e390h for the volatile partition. The partition number is 0, it is not modifiable and the type is MBR. This is in contrary to the standard partition setup - the partition number should be 1-based and partitions are usually modifiable. The MBR type is 0 meaning unused, it is not bootable and not recognized. Finally a GPT attribute is set: GPT_ATTRIBUTE_PLATFORM_REQUIRED. Typically this flag is set for OEM partitions and it prevents DiskPart.exe to perform any operations.

This structure setting is extremely unusual. The configuration makes the partition invisible for most of applications. On the other hand, this can definitely be used as an indicator of compromise.

IOCTL_DISK_GET_PARTITION_INFO 74004h: The partition is kept stealth; the returned information is coherent with the one returned for control code 70048h. The the output structure is the following.

```
typedef struct _PARTITION_INFORMATION {
  LARGE_INTEGER StartingOffset;
  LARGE_INTEGER PartitionLength;
  DWORD         HiddenSectors;
  DWORD         PartitionNumber;
  BYTE          PartitionType;
  BOOLEAN       BootIndicator;
  BOOLEAN       RecognizedPartition;
  BOOLEAN       RewritePartition;
} PARTITION_INFORMATION;
```

IOCTL_DISK_GET_LENGTH_INFO 7405ch: Return the partition length; stored at 8e388h for the first persistent partition and at 8e390h for the volatile partition.

IOCTL_MOUNTDEV_QUERY_DEVICE_NAME 4D0008h: Return the device name accordingly: the handler is also used for the volatile VFS \Device\RawDisk1 or \Device\RawDisk2.

IOCTL_DISK_IS_WRITABLE 70024h

IOCTL_DISK_MEDIA_REMOVAL 74804h

IOCTL_DISK_CHECK_VERIFY 74800h

IOCTL_DISK_SET_PARTITION_INFO 7c008h

IOCTL_STORAGE_CHECK_VERIFY2 2d0800h

IOCTL_STORAGE_CHECK_VERIFY 2d4800h

IOCTL_STORAGE_MEDIA_REMOVAL 2d4804h

IOCTL_MOUNTDEV_QUERY_DEVICE_NAME 4d0008h: The request is simply completed, keeping the VFS stealth.

Other IRP major functions: The request is simply completed.

4.3 Volatile file system

4.3.1 Bind the the device to the backing section

4.3.2 Initialize FAT file system.

4.4 Encrypted IO.

4.5 Extraction

4.5.1 Live extraction

Attempt to extract the filesystem via a dd.exe utility on the device failed at first time because raw access to the physical device is blocked after the file system is mounted. However, after a small modification in the driver file that lets the driver install the encrypted volume but prevents mounting it, we could successfully dump the virtual file system. Modified the execution of the rooktit by pausing it just after the call to setup virtual storage at 0x15AAB. Then it was possible to dump the encrypted disk by using the following command:

```
dd=\\?\Device\RawDisk1 of=disk.img
```

There is also a way to access files on the VFS the same way Snake operators are doing it; the VFS is used by the user mode components of Snake and by any other tool including cmd.exe. To access it from user mode it is sufficient to operate a volume name in the 'dot' namespace, for example:

```
C:\Documents and Settings\Administrator>dir \\.\Hd1\\
 Volume in drive \\.\Hd1 has no label.
 Volume Serial Number is 0000-1D1E

 Directory of \\.\Hd1

[output redacted]
10/18/2011  06:23 AM                 0 dump
05/04/2013  06:10 AM             8,334 klog
10/18/2011  03:50 AM           294,912 pscp.exe
10/21/2011  08:17 AM         1,089,536 queue
10/06/2011  07:33 AM         1,089,536 queue.sav
10/20/2011  05:29 AM           275,968 rar.exe
              16 File(s)      3,475,618 bytes
               0 Dir(s)               0 bytes free

C:\Documents and Settings\Administrator>
```

Volume Hd1 is a permanent virtual storage, on the contrary, Hd2 is a temporary volatile virtual storage so there is no files on there unless during the same session on the infected machine:

```
C:\Documents and Settings\Administrator>dir \\.\Hd2\\
 Volume in drive \\.\Hd2 has no label.
 Volume Serial Number is BA9B-99E8

 Directory of \\.\Hd2

File Not Found
```

4.5.2 Offline extraction

The input and output to the VFS are handled by the routine located at 0x33500 which encrypt and decrypt data on the fly per chunks of 0x200 bytes. This routine uses MmMapLockedPagesSpecifyCache to serve the decrypted data. The hook for encrypted IO is installed via hooking IofCallDriver API (see 2.3.2). Each time there is an operation to be performed on the encrypted disk, the hook will divert it to the decryption routine first. The operation is performed at low level and is completely transparent to other kernel components and the user processes.

Cryptography The cryptographic algorithm used to access the VFS is a cipher derived from CAST composed of a key scheduler located at 0x31300 and a cipher primitive located at 0x37E0. The 128bit cryptographic key used to access the VFS is located at 0x8BB40.

```
0008bb40: A1 D2 10 B7 6D 5E DA 0F A1 65 AF EF 79 C3 66 FA
```

RawDisk1 The VFS RawDisk1 mapped from hotfix.dat is believed to be mounted as \\Hd1. When decrypted, hotfix.dat appears as an NTFS volume containing binaries, configuration files and encrypted files but also queue and queue.sav.

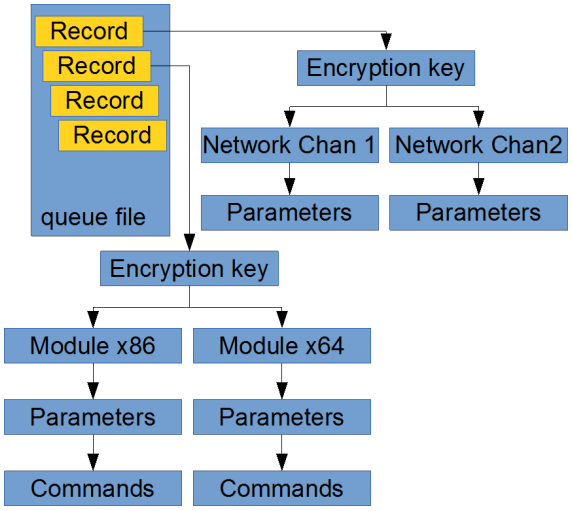

Figure 6: Snake messaging framework

RawDisk2 The VFS RawDisk2 is mounted as \\Hd2. The header is filled with the following buffer which is a file system header of a FAT partition.

```
00000000  eb 00 00 00 00 00 00 00  00 00 00 00 02 03 02 00  |................|
00000010  02 00 02 f8 00 00 0a 01  20 00 02 00 01 00 00 00  |........ .......|
00000020  ff 1f c7 00 80 29 e8 99  9b ba 4e 4f 20 4e 41 4d  |.....)....NO NAM|
00000030  45 20 20 20 46 41 54 31  36 20 20 20 00 00 00 00  |E   FAT16   ....|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

5 Bootstrap: the queue file

After the initialization the rootkit opens and reads its configuration resources. This is contained in the ‘queue‘ file located on Hd1 (see the fragment of the directory listing command at 4.5.1). This queue file is divided into slots called messages (see Figure 6). A message consists of 64-byte header (TODO - describe the header) and the body. Most of the messages are encrypted and must be decrypted by the routine located at sub_20410. Each message is identified by the message ID. Some message IDs are precisely defined, others are defined by range which describes their meaning (for example, msg 0x229 contains an encryption key to decrypt subsequent messages, messages 0x65-0x6F contain communication objects to be created). Queue file is also being accessed by the userland modules (TODO). Messages can be of type integer or string. There are many message ID ranges that are being read from the bootstrap queue by Snake kernel, so far the following have been identified: in the kernel part:

- msg 0x229 (not encrypted): contains a key to decrypt other messages. Read in at sub_29570
- msg 0x8: contains a 32-bit value to replace the DEADBEAF trigger cryptochecksum replacement. It is deliver as a DWORD in textual form (0x41414141) and is then converted to integer. Read in at sub_49960.
- msg 0x9: another integer is read similarly that affects processing of the HTTP trigger

- msg 0x2F3 - 0x2FB: integers, probably some network parameters are read in and stored in a table at 0x8d5c8 by sub_37c60
- msg 0x65 - 0x6F: contains the communication objects to be instantiated with the name of the pipe. it will create a communication object 'enc.frag.np' with the named pipe isapi_http1. In an example queue the following objects are created:
  - 'enc.frag.np' bound to isapi_http1 pipe
  - 'frag.enc.reliable.doms.np' bound to isapi_dg1 pipe
- msg 0x70 - 0x79: contain the parameters for the instantiated communication objects, e.g. 0x70 contains parameters for object created from 0x65, 0x71 for 0x66 etc. Each of the base object accepts some configiration parameters, for example np objects accepts 'allow=*everyone', enc objects accepts 'key=some_key'
- msg 0x1: unknown integer (0x332)
- msg 0x28B - 0x2BD: configure which processes are to be monitored in order to inject userland DLLs when they appear. Usual targets are: services.exe, explorer.exe, browser processes
- variable ID - injection payloads. DLL userland modules which will be injected to defined earlier processes. Injection is operated from sub_19fc0 (snake_module_start)

6 Userland components

Snake is a modular framework and it is difficult to analyze it without its userland counterparts that get injected from the queue during rootkit initialization. The authors have captured three different user mode components, each of them compiled for the two different architectures: 32 and 64-bit. It is possible that more components exist in the wild.

All user mode components are extracted from the queue file (see 5) and injected into the services.exe process by the kernel PE injector. The main malicious thread is created by the kernel code and its priority is being bumped to 31 which corresponds to the highest of 'real-time' priorities in Windows. It means that this thread is not being pre-empted if it continues to stay in the running state. It also makes the thread unstoppable by the debugger. The kernel is waiting for this thread with the timeout set to 15 seconds, probably to prevent the freeze of the system in case of the thread entering a long loop. The thread must spawn other threads before this timeout, otherwise the kenrel will terminate it. 32-bit versions of the modules will be described but the same functionality is implemented in their 64-bit counterparts.

6.1 inj_snake_Win32.dll - main module

This is the main userland module of Snake rootkit. This module is very large (884 functions) and performs many tasks. Currently identified tasks are the following:

- Opening the queue file and reading/writing messages destined for user space
- Connecting and operating the other endpoints of named pipe communication channels. It includes decrypting/encrypting and interpreting the traffic that is sent by/to the kernel components (see 3.8)

- Providing callback functions for all other injected modules (`code_result_tbl()`, `snake_modules_command()` etc.)

TODO

6.2 inj_services_Win32.dll

TODO

6.3 rkctl_Win32.dll - rootkit control module

This module seems to provide functions to control the rootkit behavior. TODO

A List of int C3h handlers (TODO)

B Supporting functions

B.1 PE image mapping (TODO)

B.2 Relocation algorithm

The PE relocation algoritm implemented by the rootkit follows the lines of the specification. It uses the fix-up table pointed by the *BaseReloc.VirtualAddress* field in the PE optional header. This table is usually located in the *.reloc* section. It is broken into blocks which define the fixups, each block begins with a 32bit address pointing to the location where the fix-ups will be applied followed by the size of the block expressed as a 32bit unsigned integer. This 8 byte header is followed by the fix-ups expressed as a bit field where the 4 high bits express the relocation type and the 12 low bit express the relocation offset. Programmatically the relocation table is a table of structures defined as follows.

```
00000000 RELOCATION_BLOCK struc
00000000 fixup_target    dd
00000004 block_size      dd
00000008 fixup[N]        dd ; where N = (block_size - 8)/4
???????? RELOCATION_BLOCK ends
```

The algorithm embedded in the rootkit processes the following fixup types.

IMAGE_REL_BASED_ABSOLUTE value 0 Do nothing, this is used to pad the relocation table

```
locZ_5B397:              ; IMAGE_REL_BASED_ABSOLUTE
jmp     loc_5B42B
```

IMAGE_REL_BASED_HIGH value 1 Add the 16 bits of the fix-up offset to the high word of the relocation target as a 32-bit address.

```
loc_5B39C:              ; IMAGE_REL_BASED_HIGH
mov     eax, [ebp+imagebase]
add     eax, [ebp+reloc_target_offset]
mov     [ebp+reloc_target_address], eax
mov     ecx, [ebp+reloc_target_address]
movzx   edx, word ptr [ecx]
mov     eax, [ebp+PE_imagebase]
shr     eax, 10h
movzx   ecx, ax
```

```
add     edx, ecx
mov     eax, [ebp+reloc_target_address]
mov     [eax], dx
jmp     short loc_5B42B
```

IMAGE_REL_BASED_LOW value 2 Add the 16 bits of the fix-up offset to the low word of the relocation target as a 32-bit address.

```
loc_5B3BE:              ; IMAGE_REL_BASED_LOW
mov     ecx, [ebp+imagebase]
add     ecx, [ebp+reloc_target_offset]
mov     [ebp+reloc_target_address], ecx
mov     edx, [ebp+reloc_target_address]
movzx   eax, word ptr [edx]
mov     ecx, [ebp+PE_imagebase]
and     ecx, 0FFFFh
movzx   edx, cx
add     eax, edx
mov     ecx, [ebp+reloc_target_address]
mov     [ecx], ax
jmp     short loc_5B42B
```

IMAGE_REL_BASED_HIGHLOW value 3 Add the the fix-up offset to the low word of the relocation target as a 32-bit address.

```
loc_5B3E3:              ; IMAGE_REL_BASED_HIGHLOW
mov     edx, [ebp+imagebase]
add     edx, [ebp+reloc_target_offset]
mov     [ebp+reloc_target_address32], edx
mov     eax, [ebp+reloc_target_address32]
mov     ecx, [eax]
add     ecx, [ebp+PE_imagebase]
mov     edx, [ebp+reloc_target_address32]
mov     [edx], ecx
jmp     short loc_5B42B
```

IMAGE_REL_BASED_DIR64 value 0Ah Add the the fix-up offset to the low word of the relocation target as a 64-bit address.

```
loc_5B402:              ; IMAGE_REL_BASED_DIR64
mov     eax, [ebp+imagebase]
add     eax, [ebp+reloc_target_offset]
mov     [ebp+reloc_address64], eax
mov     ecx, [ebp+PE_imagebase]
xor     edx, edx
mov     eax, [ebp+reloc_address64]
add     ecx, [eax]
mov     eax, [eax+4]
adc     eax, edx
mov     edx, [ebp+reloc_address64]
mov     [edx], ecx
mov     [edx+4], eax
jmp     short loc_5B42B
```

The following relocation types are not implemented.

IMAGE_REL_BASED_HIGHADJ value 4 Apply the relocation to relocation with a 32bit offset. It occupies two more bytes in the fix-up table.

IMAGE_REL_BASED_MIPS_JMPADDR value 5 Apply the relocation to a MIPS[9] jump.

IMAGE_REL_BASED_SECTION value6 Reserved.

IMAGE_REL_BASED_REL32 value 7 Reserved.

IMAGE_REL_BASED_MIPS_JMPADDR16 value 9 Apply the relocation to a MIPS16 jump.

IMAGE_REL_BASED_HIGH3ADJ value 0Bh This is a 48bit relocation, it occupies 4 more bytes in the relocation table.

B.3 Get module reference

The function *lookup_module (438F0h)* gets the image references of the module name specified as the first argument. Upon success it returns the image base address and the image size of the located module. The values are then respectively stored in *main_struct.ntoskrnl_baseaddress (8CB80h)* and *main_struct.ntoskrnl_imagesize (8CB84h)*.

The target module is searched in the module list obtained via a system call *ZwQuerySystemInformation* on *SystemModuleInformation (16h)*. This technique is documented at the following address *http://alter.org.ua/docs/nt_kernel/procaddr*. The selection routine implements four cases according to the target module name.

ntoskrnl.exe The first module in the list is selected.

```
.text:00034997                cmp     [ebp+idx], 0
.text:0003499B                jnz     short loc_349B3
.text:0003499D                mov     ecx, [ebp+module_name_len]
.text:000349A0                push    ecx
.text:000349A1                push    offset aNtoskrnl_exe ; ‘‘ntoskrnl.exe’’
.text:000349A6                mov     edx, [ebp+module_name]
.text:000349A9                push    edx
.text:000349AA                call    stricmp         ; case insensitive
.text:000349AF                test    eax, eax
```

HAL.dll The second module in the list is selected.

```
.text:000349B3                cmp     [ebp+idx], 1
.text:000349B7                jnz     short loc_349CF
.text:000349B9                mov     eax, [ebp+module_name_len]
.text:000349BC                push    eax
.text:000349BD                push    offset aHal_dll ; ‘‘HAL.dll’’
.text:000349C2                mov     ecx, [ebp+module_name]
.text:000349C5                push    ecx
.text:000349C6                call    stricmp         ; case insensitive
.text:000349CB                test    eax, eax
```

ntdll.dll The first module with a base address in userland is selected, with a baseaddress lower or equal than 0x80000000.

```
.text:00034A02                mov     ecx, [ebp+module_name_len]
.text:00034A05                push    ecx
.text:00034A06                push    offset aNtdll_dll ; ‘‘ntdll.dll’’
.text:00034A0B                mov     edx, [ebp+module_name]
.text:00034A0E                push    edx
```

[9]Specific machine code: Microprocessor without Interlocked Pipeline Stages.

```
.text:00034A0F                call    stricmp         ; case insensitive
.text:00034A14                test    eax, eax
.text:00034A16                jnz     short loc_34A33
.text:00034A18                mov     eax, [ebp+idx]
.text:00034A1B                imul    eax, size SYSTEM_MODULE
.text:00034A21                mov     ecx, [ebp+module_table]
.text:00034A24                cmp     [ecx+eax+SYSTEM_MODULE.ImageBaseAddress], 80000000h
.text:00034A2C                jbe     short loc_34A33
```

Other name The first module matching the name is selected.

```
.text:000349CF                mov     edx, [ebp+module_name_len]
.text:000349D2                push    edx
.text:000349D3                mov     eax, [ebp+module_name]
.text:000349D6                push    eax
.text:000349D7                mov     ecx, [ebp+idx]
.text:000349DA                imul    ecx, size SYSTEM_MODULE
.text:000349E0                add     ecx, [ebp+module_table]
.text:000349E3                mov     edx, [ebp+idx]
.text:000349E6                imul    edx, size SYSTEM_MODULE
.text:000349EC                mov     eax, [ebp+module_table]
.text:000349EF                movzx   edx, [eax+edx+SYSTEM_MODULE.NameOffset]
.text:000349F4                lea     eax, [ecx+edx+SYSTEM_MODULE.Name]
.text:000349F8                push    eax
.text:000349F9                call    stricmp         ; case insensitive
.text:000349FE                test    eax, eax
```

B.4 Get procedure address (TODO)