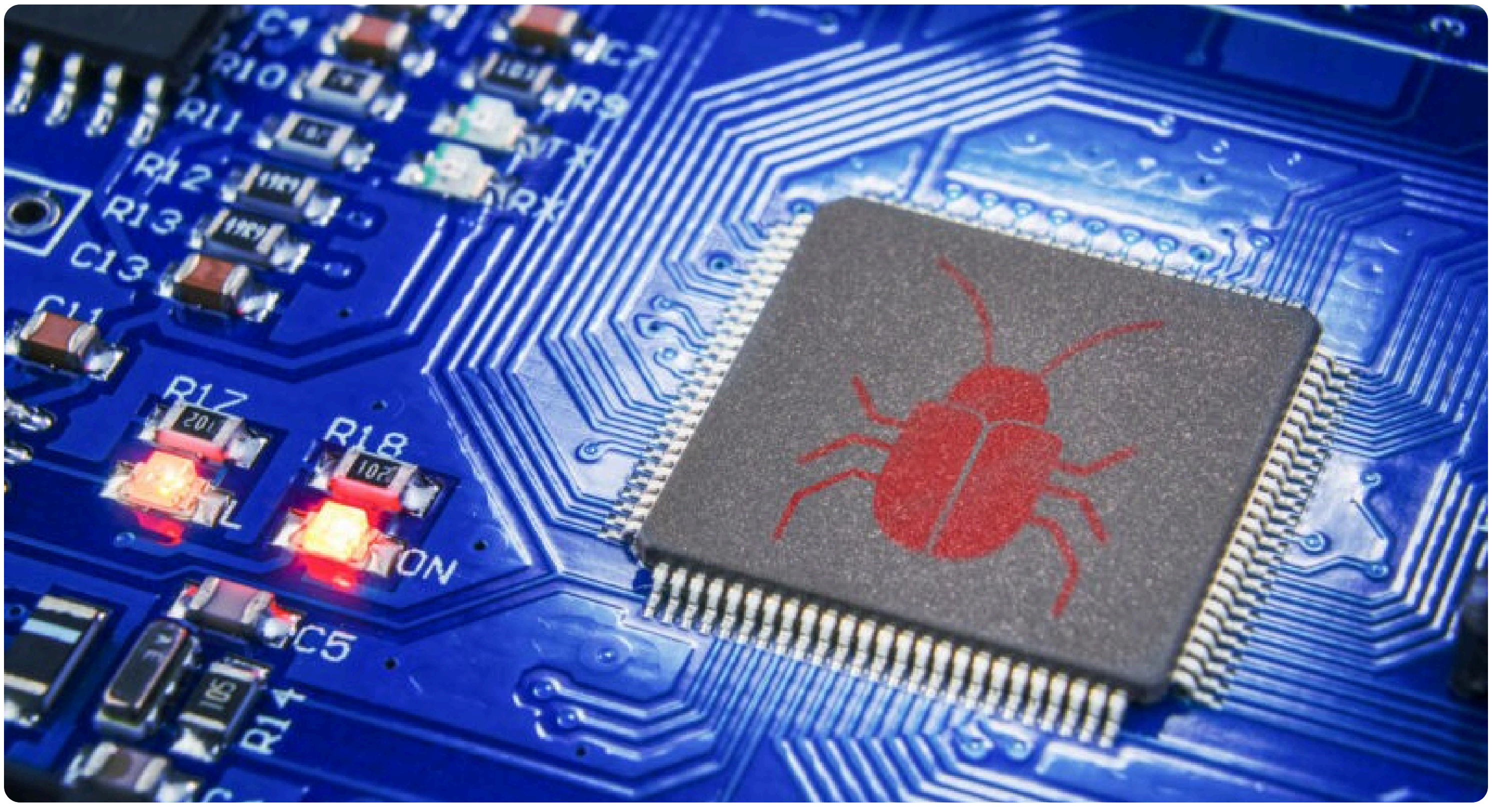

Guntior - the story of an advanced bootkit that doesn't rely on Windows disk drivers

Artem Baranov
Certified Cyberpunk Netrunner
Published Nov 14, 2023

+ Follow

Update 1: The MITRE ATT&CK matrix reflecting the malware TTPs was added.

I first stumbled upon this interesting malware sample about a decade ago, being a contributor to the kernelmodeinfo forum. Amid the rise of bootkits at that time, the dropper was captured in-the-wild and posted on one of malware trackers. The malware was called "Guntior", after the device object its authors had chosen for it (\Device\Guntior). The name also appears in AV detections.

At this time, most systems were x86, and thus didn't benefit from **Kernel Patch Protection (KPP)** or **Driver Signing Enforcement**. As a result, there was a lot of sophisticated malware loading unsigned drivers that used kernel mode hooks and direct disk access to hide malicious activity. Bootkits typically store their components in the disk sectors outside the normal file system and conceal their data from the rest of the operating system by returning zeroes or spoofed data in response to any requests for it

The analysis of any bootkit involves not only reverse engineering skills, but also forensic skills in order to extract the infected boot sector, MBR and malware modules from disk sectors.

Chapters:

The dropper

Okay, let's take a look at the dropper. The bootkit itself is encrypted and stored in the resources section of the malware dropper. The latter is wrapped in another dropper, which appears to have been downloaded by the user. Along with the bootkit, the resources section also incorporates other encrypted malware modules.

The bootkit dropper has several anti-debug and anti-analysis tricks to confuse a malware researcher. Its Original Entry Point (OEP) is called via a Structured Exception Handling (SEH) handler upon throwing an exception. To confuse a potential researcher, the dropper creates its copy on disk and converts this executable into a DLL. This trick is used to bypass HIPS sensors and explained later. After performing these anti-analysis tricks, it proceeds with execution.

HIPS evasion

The malware performs an interesting trick to inject its DLL into Explorer, which works on older versions of Microsoft Windows. This technique allows the malware to bypass HIPS checks and behavioral protection designed to detect malicious activity. In a nutshell, the malware itself doesn't inject the DLL into the target process, but causes the OS to do so.

The trick is based on using Windows Input Method Editor (IME) keyboard layout feature, which allows an attacker to load a DLL into the process after sending the WM_INPUTLANGCHANGEREQUEST message. Before doing this, the client needs to register the DLL in a special registry key that is assumed to implement this keyboard layout. This registry key should store a value named "Ime File" with a DLL path, but writing any file path in the registry would trigger an alert of any behavioral protection. To avoid this, the malware intercepts the *NtQueryValueKey* API to provide the necessary DLL path. There's another important API in the event chain - *ImmLoadLayout*. It loads the malicious DLL In the context of the Explorer process, after receiving the aforementioned message.

Before doing these actions, the malware copies its executable to the system directory with a random name.tmp and patches its PE characteristics by setting the corresponding DLL flag, effectively converting it into a DLL.. As it's not difficult to guess, this DLL is intended to be injected into Explorer as explained above.

Let's sum up this injection technique:

- The malware copies its executable to the temp directory as a DLL.
- Registers a new keyboard layout in the registry.
- Intercepts *ZwQueryValueKey* in its own process to supply "Ime File" registry value with the malware DLL path to the OS.
- Gets a handle to the Explorer window with *FindWindow*.
- Send the WM_INPUTLANGCHANGEREQUEST message to that window.
- The OS calls the *ZwQueryValueKey* hook to query the "Ime File" value of the newly registered keyboard layout and gets the DLL path.
- Windows caches information about this keyboard layout.
- Send the WM_INPUTLANGCHANGEREQUEST message again.
- *ImmLoadLayout* loads the DLL into the Explorer process using the cached data.

Created keyboard layout data.

The following diagram explains this trick.

Disabling security software

Upon its successful execution, the DLL code sets an event to signal the dropper. This injected DLL is responsible for loading the bootkit driver, but before doing this, it tries to disable the following tools by sending the appropriate IOCTLs to their drivers or killing their processes.

- 360tray.exe, 360 Total Security by Qihoo 360
- HintClient.exe by Shanghai Hintsoft Co., LTD, IOCTL code 0x00403A3B.

- DrvMon tool by Fyyre and EP_X0FF, IOCTL code 0x00403B0A.
- HardwareInfo.exe, part of the NetTools, IOCTL code 0x00220008.
- CfgClt.exe
- AVP.exe, Kaspersky security products
- KSafeTray.exe, PC Doctor Flow Monitor (PC Doctor) by Kingsoft
- RavMonD.exe, Rising AntiVirus by Beijing Rising Information Technology

For example, below you can see the code to disable DrvMon.

The bootkit is particularly interested in disabling ESET security products. It creates a separate thread in which it tries to terminate nod32krn.exe service in an infinite loop with a timeout of two seconds. To ensure that the process is killed, the driver is used.

Driver installation

Another interesting characteristic of this malware is how it installs and loads its driver. This tricky method is based on hijacking the Microsoft Trusted Audio Drivers service named drmkaud (drmkaud.sys) and utilizing PnP Manager to load it. To install the driver, the malware performs the following actions.

1. Selects a random name for its driver to be dropped into C:\Windows\System32.
2. Opens the registry key of Trusted Audio Drivers in HKLM\SYSTEM\CurrentControlSet\Enum\SW belonging to Device Manager with the full path HKLM\SYSTEM\CurrentControlSet\Enum\SW\GUID\GUID\{eec12db6-ad9c-4168-8658-b03daef417fe}\{ABD61E00-9350-47e2-A632-4438B90C6641}. These GUIDs are stored in encrypted form.
3. Modifies the security descriptor allowing Everyone all access to those keys.
4. Sets ConfigFlags value to zero, replaces the original name of the service drmkaud with the malware's.

5. Creates the driver service key in HKLM\SYSTEM\CurrentControlSet\Services.

6. Extracts the driver from the resources section, decrypts it and drops it onto disk.

Hijacked service

The Trusted Auto Driver Service in the Enum key is hijacked so that its GUID can be used to load the malicious driver via the PnP Manager. The malware repeatedly tries to open any device named \Device\000000NN, each time incrementing the N values until it succeeds. For each device the code sends a special IOCTL code supplying the aforementioned GUID, which is stored in encrypted form.

If this trick fails, the malware loads the driver as usual, manually creating its service. Once it's loaded, the malware starts infecting the disk.

Payload DLL

The dropper uses a similar trick to register the payload DLL in the system. Instead of simply dropping the DLL and registering it in autorun, it hijacks one of Windows standard services by rewriting its executable with the malware DLL. To defend against malware analysis, the malware stores a list of these DLL names in encrypted form and decrypts them only briefly to the stack, making it unlikely that an analyst will find these names in a memory dump.

Trying to hijack at least one of them, the malware targets the following Windows services:

- AppMgmt - Software Installation Service
- BITS - Background Intelligent Transfer Service
- FastUserSwitchingCompatibility - Fast User Switching Compatibility service
- WmdmPmSN - Portable Media Serial Number Service
- xmlprov - Network Provisioning Service

- EventSystem - Event System service
- Ntmssvc - Removable Storage Service
- upnphost - Universal Plug and Play Device Host Service
- SSDPSRV - Simple Service Discovery Protocol service
- Netman - Network Connections service
- Nla - Network Location Awareness Service
- Tapisrv - Telephony service
- Browser - Browser service
- CryptSvc - Cryptographic Services
- helpsvc - Help Center Service
- RemoteRegistry - Remote Registry service
- Schedule - Schedule service

Below you can see the steps of this process. Before rewriting the system executable, the malware gets the address of *SfcFileException* export function in sfc_os.dll. Sounds familiar, right? This DLL implements the Windows System File Checker (SFC) API that can be used to scan or restore corrupted system files. The malware authors abuse one of these API functions to prevent SFC from automatically restoring the target system file after its modification.

Service hijacking scheme

The hijacked service DLL is responsible for communication with the driver, supplying it with pointers to undocumented ntoskrnl functions and kernel structure offsets, saving the driver from having to do it in kernel mode. The first IOCTL to be sent to the driver is 0x222440, which initializes it for further operation. The following data is to be sent.

- Addresses of the following functions: *MmGetSystemRoutineAddress*, *PspTerminateThreadByPointer*, *KeInsertQueueApc, KiInsertQueueApc*;

- First 12 bytes of each of those functions (their prologs); *PspTerminateThreadByPointer*;

- The offsets of the following kernel structures are EPROCESS->ThreadListHead, ETHREAD->ThreadListEntry.

```
struct ROOTKIT_INIT_STRUCT
{
    DWORD MmGetSystemRoutineAddress;
    DWORD PspTerminateThreadByPointer;
    DWORD KeInsertQueueApc;
    DWORD KiInsertQueueApc;
    BYTE PspTerminateThreadByPointer_First_C_bytes[12];
    BYTE KeInsertQueueApc_First_C_Bytes[12];
    BYTE KiInsertQueueApc_First_C_Bytes[12];
    DWORD EPROCESS_ThreadListHead_Offset;
    DWORD ETHREAD_ThreadListEntry_Offset;
};
```

When the malware needs to terminate a specific process, it sends a special IOCTL containing the PID. The driver gets a EPROCESS pointer to this PID and enumerates all threads belonging to this processes using the supplied offsets of EPROCESS->ThreadListHead and ETHREAD->ThreadListEntry and for each of them calls *PspTerminateThreadByPointer*.

The DLL contains an impressive list of processes to be terminated. Some of them belong to well-known security companies.

- **ESET** - nod32krn.exe, egui.exe, ekrn.exe;

- **Qihoo 360** - 360tray.exe, 360leakfixer.exe, 360Safe.exe, safeboxTray.exe, 360safebox.exe, 360sd.exe, ZhuDongFangYu.exe, 360rp.exe, 360sdupd.exe;

- **Kingsoft WebShield** - KSWebShield.exe, kxesapp.exe, kxeserv.exe, kwstray.exe, kxedefend.exe, upsvc.exe, kxescore.exe, KVExpert.exe, kxetray.exe, KSafeSvc.exe, KSafeTray.exe;

- **Rising AntiVirus** - RavMonD.exe, RsTray.exe, RsAgent.exe, RegGuide.exe, RsMain.exe, RsCopy.exe, Rav.exe;

- Jiangmin Antivirus - KVSrvXP.exe, KVExpert.exe, KVMonXp.exe

- Kaspersky - AVP;

- Rising PC Doctor - ras.exe, knownsvr.exe, rstray.exe;

- Tencent QQPCMgr - QQPCLeakScan.exe, QQPCWebShield.exe, QQPCTAVSrv.exe, QQPCRTP.exe, QQPCMgr.exe, QQPCUpdateAVLib.exe, QQPCTray.exe, QQRepair.exe, QQPCPatch.exe;

- Other - Calc.exe (:D), guiyingfix.exe, knsdtray.exe, knsd.exe, knsdsvc.exe, knsdsve.exe.

Another interesting observation is that, before calling the undocumented function pointers passed from user mode, the driver first tries to restore the first 12 bytes of each function. The authors probably assumed that the drivers of some security products set hooks on these functions by patching their first bytes at runtime. The DLL provides the driver with these first 12 bytes, but before that, it copies them from ntoskrnl on disk, loading it into the process and finding those undocumented functions, i e *PspTerminateThreadByPointer* and *KeInsertQueueApc*. The screenshot below demonstrates this technique.

The code above walks through the list of all processes' threads as follows.

```
pEPROCESS = PsLookupProcessByProcessId(Supplied_Pid);

PLIST_ENTRY pThreadListHead = (PUCHAR)pEPROCESS + Supplied_ThreadListHead_Offse

PLIST_ENTRY pCurrentThreadEntry = pThreadListHead;
pCurrentThreadEntry = pThreadListHead->Flink;

while(pCurrentThreadEntry != pThreadListHead)
```

```
    {
        PETHREAD pCurrentThread = (PUCHAR)pCurrentThreadEntry -

        ObReferenceObjectByPointer(pCurrentThread);
        ObDereferenceObject(pCurrentThread);
        TerminateThread;
        pCurrentThreadEntry = pCurrentThreadEntry->Flink;
    }
```

Now we can take a look at the entire malware installation routine.

The DLL is a core part of the malware, which acts as a bot and communicates with its control servers. The address is hard coded in the DLL. After connecting to 183[.]60[.]132[.]220, it tries to download one of the executable files with the names 1.exe, 2.exe, 3.exe, ..., 32.exe and run it in the system with the Explorer access token. Unlike TDL4, this bootkit isn't interested in protecting downloaded executables on disk. They are stored as regular files on volume and the bootkit doesn't restrict access to them.

A few words about the Windows disk I/O subsystem

Being the OS that is based on the hybrid kernel architecture satisfying the highest standards, Windows has the multi-layered disk I/O architecture. It's based on the device stack model, where each layer is represented by a separate kernel mode driver. One of the main advantages of this approach is that each of these layers is isolated from others and can use the unified Windows I/O subsystem's interfaces to communicate with other drivers on the stack. For example, the File System Driver (FSD) can dynamically attach its device to the existing disk device stack to dispatch file operations.

At the top of this device stack is the FSD (ntfs.sys), which handles file operations and attaches its device to the lower one belonging to the volume manager (volmgrx.sys). In order to operate files data, the FSD converts file offsets to the volume ones and addresses volmgrx.sys. The latter converts its offsets to the disk ones and calls the disk driver disk.sys or it can reach out to the partition manager (partmgr.sys) that is also located down the device stack. Unlike the device belonging to the volume manager, the partition manager's one represents a raw disk partition without a file system. Upon receiving a request, disk.sys calls the disk port driver atapi.sys, clarifying exactly which device on the ATA bus it needs to reach. The disk port driver completes this request by communicating directly with the disk controller. The major difference between the FSD and other drivers on the stack is that the first should keep the context for any open file (FILE_OBJECT and its FsContext) while others simply operate with disk or volume offsets.

The driver objects of the aforementioned drivers were an attractive target for rootkits and bootkits in the x86 era. The malicious Ring 0 code aimed at modifying the function pointers in the drivers' dispatch table to intercept the I/O operations of interest. IRP_MJ_READ, *WRITE,* DEVICE_CONTROL were the primary targets. Thus, rootkit detectors had to go as low as possible to safely read disk sectors, skipping potentially infected drivers located higher on the stack.

Low-level disk access via ATA PIO mode

The rootkit includes a unique feature we haven't seen in other notorious bootkits such as TDL4, Rovnix or Mebroot. It's capable of communicating with the hard drive at the lowest level without calling any Windows disk or disk port drivers (disk.sys, atapi.sys).

In a nutshell, instead of sending the IOCTL_SCSI_PASS_THROUGH_DIRECT request to atapi, the rootkit works directly with the ATA bus via IO ports 0x170-0x177 and a device control register port such as 0x376. Once all the preparations are done, the rootkit calls *hal!READ_PORT_BUFFER_USHORT* to read data from disk or *hal!WRITE_PORT_BUFFER_USHORT* to write to it. At the beginning of this routine, the rootkit queries the information about the IDE controller using *hal!HalGetBusData* for *PCIConfiguration*.

In spite of the presence of this interesting feature, the malware uses it in very limited cases. These cases don't even require calculating disk offsets using the partition table.

- To overwrite the MBR and drop the malware components onto the end of the disk during the disk infection process.

- The disk infection watchdog thread reads the MBR every 5 seconds in an infinite loop.

The disk infection routine is located in the main dropper and called in the context of Windows Explorer process after code injection is done as described above. If the injection fails, the dropper calls this function in the context of its process.

The rootkit provides the malware with two disk communication IOCTLs. The first is used to select the drive on which the malware wants to perform the RW operation and another describes this request.

```
#define IOCTL_ROOTKIT_DEVICE 0x22

#define IOCTL_ROOTKIT_SEND_MBR_SIGNATURE CTL_CODE(IOCTL_ROOTKIT_DEVICE, 0x90E,

#define IOCTL_ROOTKIT_READ_WRITE_DISK CTL_CODE(IOCTL_ROOTKIT_DEVICE, 0x90F, MET

DWORD dwMBRSignature; //in buffer for IOCTL_ROOTKIT_SEND_MBR_SIGNATURE

struct ROOTKIT_IO_DISK_PACKET
{
    DWORD dwStartLBALow;   //0x00
    DWORD dwStartLBAHigh;  //0x04
    WORD wNumberOfSectors; //0x08
    DWORD cbData; //0x0A
    enum
    {
        OpRead = 1,
        OpWrite
    } OpType;              //0x0E
};  //0x12
//in buffer for IOCTL_ROOTKIT_READ_WRITE_DISK

struct ROOTKIT_INTERNAL_DISK_IO_STRUCTURE
{
    DWORD dwStartLBALow;   //0x00
    DWORD dwStartLBAHigh;  //0x04
```

```
        WORD wNumberOfSectors; //0x08
        WORD wATA_Command;     //0x0A
        PVOID pDataBuffer;     //0x0E
        DWORD cbData;          //0x10
        enum
        {
            OpRead = 1,
            OpWrite
        } OpType;              //0x14
    }; //0x18
    //this is the structure that the driver uses internally to perform I/O
```

The rootkit uses the following ATA commands.

- 0x30 and 0x34 to write sectors in 28/48 bit PIO mode;
- 0x20 and 0x24 to read sectors;
- 0xEC - IDENTIFY command.

Before execution of any disk operation, the rootkit polls the drive to be sure that it's ready to transfer data.

To start using the IOCTL_ROOTKIT_READ_WRITE_DISK operation, the rootkit requires another IOCTL named IOCTL_ROOTKIT_SEND_MBR_SIGNATURE to be sent before. The latter is needed to prepare the rootkit internal structures for performing further I/O. This structure includes the following information: ATA bus port number, device control port number, ATA command type (8bit or 24). The rootkit globally stores this structure to supply the disk I/O functions with the necessary information required to perform I/O operations.

In order to get the necessary disk configuration information, the rootkit uses *HalGetBusData* with *PCIConfiguration* value as BusDataType and receives the PCI_COMMON_HEADER structure as output. In a loop, it iterates buses, slots, PCI classes, and subclasses until it gets PCI_CLASS_MASS_STORAGE_CTLR and PCI_SUBCLASS_MSC_IDE_CTLR. The implementation of the entire process of searching a specific drive is presented below.

```
typedef struct _PCI_SLOT_NUMBER {
    union {
        struct {
            ULONG   DeviceNumber:5;
            ULONG   FunctionNumber:3;
            ULONG   Reserved:24;
        } bits;
        ULONG   AsULONG;
     } u;
} PCI_SLOT_NUMBER, *PPCI_SLOT_NUMBER;

void RootkitPciIdeGetConfigurationInfo(DWORD dwDiskMbrSignature)
{
    PCI_COMMON_HEADER PCIDeviceConfig;
    DWORD cbWritten;

    for (DWORD iBus = 0; iBus <= PCI_MAX_BRIDGE_NUMBER; iBus++)
    {
        for (DWORD iDevice = 0; iDevice < PCI_MAX_DEVICES; iDevice++)
        {
            for (DWORD iFunction = 0; iFunction  < PCI_MAX_FUNCTION;
                  iFunction++)
            {
                PCI_SLOT_NUMBER SlotNumber = {0};
                DWORD dwATABusPortNumber = 0;
                DWORD dwDeviceControlRegisterPortNumber = 0;
                BYTE bStatus;

                ZeroMemory(&PCIDeviceConfig, sizeof(PCIDeviceConfig));

                SlotNumber.u.bits.DeviceNumber = iDevice;
                SlotNumber.u.bits.FunctionNumber = iFunction;

                cbWritten = HalGetBusData(PCIConfiguration, iBus, iSlot,
                    &PCIDeviceConfig, sizeof(PCIDeviceConfig));
```

```
              if (cbWritten != sizeof(PCIDeviceConfig) ) continue;

         if( PCIDeviceConfig.BaseClass != PCI_CLASS_MASS_STORAGE_CTLR ||
              PCIDeviceConfig.SubClass != PCI_SUBCLASS_MSC_IDE_CTLR )
                  continue;

         if ( !(PCIDeviceConfig.type0.BaseAddresses[0] &
                  PCI_ADDRESS_IO_SPACE) ||
               !(PCIDeviceConfig.type0.BaseAddresses[0] &
                  PCI_ADDRESS_MEMORY_ADDRESS_MASK) )
                      continue;

         if ( !(PCIDeviceConfig.type0.BaseAddresses[1] &
                  PCI_ADDRESS_IO_SPACE) ||
               !(PCIDeviceConfig.type0.BaseAddresses[1] &
                  PCI_ADDRESS_MEMORY_ADDRESS_MASK) )
                      continue;

              dwATABusPortNumber =
                  PCIDeviceConfig.type0.BaseAddresses[0] &
                  PCI_ADDRESS_MEMORY_ADDRESS_MASK;

              dwDeviceControlRegisterPortNumber =
                  PCIDeviceConfig.type0.BaseAddresses[1] &
                  PCI_ADDRESS_MEMORY_ADDRESS_MASK;

              bStatus = ReadPort(dwDeviceControlRegisterPortNumber);

              //if device control register returns 0xFF or 0x7F, there
              //is no driver available
              if (bStatus == 0xFF || bStatus == 0x7F) continue;

              bStatus = PrepareGlobalATACommandStructureForTheDrive(
                  dwATABusPortNumber, dwDeviceControlRegisterPortNumber,
                      PartitionSignature);

              if (bStatus == 1) return;
          }
```

```
        }
    }
    return 0;
}
```

From the rootkit code.

Now the malware can use IOCTL_ROOTKIT_READ_WRITE_DISK to write data to disk. Depending on the output of the IDENTIFY ATA command, the rootkit selects the appropriate type of the RW operation, 28 or 48 bit PIO (IDENTIFY_DEVICE_DATA.CommandSetSupport.BigLba). Let's look at the sequence of actions in case of processing 48 bit PIO RW operation.

```
out 0x1F2, wNumberOfSectors_high_byte
out 0x1F3, LBA4
out 0x1F4, LBA5
out 0x1F5, LBA6

out 0x1F2, wNumberOfSectors_low_byte
out 0x1F3, LBA1
out 0x1F4, LBA2
out 0x1F5, LBA3

//send READ/WRITE sector command

out 0x1F7, 0x20/0x24/0x30/0x34
```

To transfer the requested data the rootkit uses the aforementioned HAL functions *READ_PORT_BUFFER_USHORT* and *WRITE_PORT_BUFFER_USHORT* in a loop.

Summarizing the IOCTLs that the driver exposes.

- IOCTL_ROOTKIT_KILL_PROCESS 0x222444 to kill the specified process, as input accepts a Process ID (PID).

- IOCTL_ROOTKIT_INIT_DATA 0x222440 to initialize the rootkit structure containing undocumented offsets and functions, as input accepts initialized ROOTKIT_INIT_STRUCT (see below).
- IOCTL_ROOTKIT_READ_WRITE_DISK 0x22243D to read/write disk data.
- IOCTL_ROOTKIT_SEND_MBR_SIGNATURE to specify a disk to the rootkit for the following I/O operations.

Disk infection

Internally, the malware supports two mechanisms for accessing the disk in raw mode. It either calls *CreateFile/ReadFile/WriteFile* on PhysicalDrive0 when it's necessary to work with Master Boot Record (MBR) or uses the rootkit driver to communicate with the disk at a low level.

As is the case with its other components, the malware stores the bootkit 16-bit bootloader in the dropper's resource section in encrypted form. The first 0x200 bytes of this data represent the malicious MBR and the rest is supposed to be written at the end of the disk.

The following picture shows the bootkit data structure on disk.

The malware infects the disk as follows:

- Send the signature of the disk to be infected to the rootkit (IOCTL_ROOTKIT_SEND_MBR_SIGNATURE).
- Read the first 16 sectors of the disk with IOCTL_ROOTKIT_READ_WRITE_DISK or with *CreateFile/ReadFile* on \\.\PhysicalDrive0 if the rootkit driver failed to load. Frankly, I didn't get why the malware reads as many as 16 sectors as it uses only first one that represents the MBR to infect it.
- Infect the MBR with the code from the 112 resource with 16-bit bootloader code.
- Allocate a virtual memory region with the size of 0x7E00 (63 sectors) and copy there infected MBR, original MBR, the 16-bit bootcode and the payload DLL.
- Overwrite the original MBR with the infected one.
- Calculate the offset from the end of the disk to drop the bootkit data (DiskSize - 0x41 sectors).

- Write the bootkit data to the end of the disk.

This is the last step of system infection.

Bootloader

The malicious bootstrap code located in the MBR is responsible for loading the bootloader from the end of the disk. Due to limitations of memory addressing in real mode, the bootstrap code first relocates itself from 0x7C00 to 0x600. This memory region starting 0x7C00 is used for further loading the bootloader data. As was mentioned above, the infected MBR stores the start LBA of the bootkit extension and its size in sectors.

The malicious bootloader has its own powerful FAT and NTFS parsers. The NTFS parser is capable of performing RW operations on files and walking through the directory hierarchy. After reading the bootkit extension, it prepares a special array describing partitions of each connected disk supporting ATA/PATA interfaces. The following structure stores an item of that array.

```
struct
{
    DWORD dwStartLBA;
    UCHAR Unused[6];
    UCHAR bFSType; //0x4E for NTFS
    UCHAR bSectorsPerCluster;
    WORD wBytesPerSector;
    UCHAR bClustersPerMftRecord;
    DWORD MftLcnLow;
    UCHAR ClustersPerIndexRecord;
} BOOTKIT_PARTITION_ARRAY_ITEM;
```

After completing this, the code copies a partition item for the selected partition to the stack. The bootkit also keeps a structure describing the file that the malware is interested in.

```
struct
{
    DWORD dwOpenFileRequestSignature_ItsResult;
    //IN 'nepo' - open file request, OUT 'nep$'- open success
    MFT_REF MftRefFile;
    ULARGE_INTEGER SizeOfDataAttrib; //file size
    DWORD FileOperationSignature_ItsResult
    //IN 'tirw' - write file, 'daer' - read, OUT 'mron' - operation
    //success, 'tir$' - used to write data for low level disk functions
} BOOTKIT_FILE_ARRAY_ITEM;
```

Thus the bootloader internally supports two I/O types - file I/O and disk I/O. The appropriate functions work with file context (file array) and partition context (partition array). According to the code, the only file the bootkit is interested in - C:\WINDOWS\System32\sfc_os.dll and the path is stored in encrypted form. The following NTFS structures are key to understanding its parser.

```
struct MFT_RECORD
{
/*0x00*/ ULONG signature; //signature 'FILE'
/*0x04*/ USHORT usa_offs;
/*0x06*/ USHORT usa_count;
/*0x08*/ ULARGE_INTEGER lsn;
/*0x10*/ USHORT sequence_number;
/*0x12*/ USHORT link_count;
/*0x14*/ USHORT attrs_offset;
/*0x16*/ USHORT flags;//looks MFT_RECORD_FLAGS
/*0x18*/ ULONG bytes_in_use;
/*0x1C*/ ULONG bytes_allocated;
/*0x20*/ ULARGE_INTEGER base_mft_record;
/*0x28*/ USHORT next_attr_instance;
/*0x2A*/ USHORT reserved;
```

```
/*0x2C*/ ULONG mft_record_number;
//size - 48 bytes
};

typedef struct _ATTR_RECORD
{
/*0x00*/ ATTR_TYPES type; //тип атрибута
/*0x04*/ USHORT length; //header size; it's used to link attributes
/*0x06*/ USHORT Reserved;
/*0x08*/ UCHAR non_resident; //1-the attribute is resident, 0-not resident
/*0x09*/ UCHAR name_length; //attribute name size in chars
/*0x0A*/ USHORT name_offset; //offs of attribute name relative to header
/*0x0C*/ USHORT flags; //ATTR_FLAGS
/*0x0E*/ USHORT instance;
...
```

Understanding NTFS is a separate long story, so we'll limit ourselves to presenting these two structures. The code of this parser is quite large. It's capable of working with all the necessary resident and non-resident file attributes, including, INDEX_ROOT and INDEX_ALLOCATION for recursive directory traversal.

Below you can see an example of an MFT record describing explorer.exe. The record has several standard file attributes, each of which has a header (ATTR_RECORD). A file attribute can be resident (fits into the MFT record) and non-resident.

- $STANDARD_INFORMATION (0x10) - contains the basic file information such as the date of creation and modification, attributes, owner and security IDs.

- $FILE_NAME (0x30) - contains the file name, a ref to the file directory and its size.

- $DATA (0x80) is responsible for storing file data.

The $DATA attribute stores all necessary information to be useful to find file data. But the process of raw parsing FS data to get its content is more complicated in the case of a non-resident attribute. The parser needs to analyze a run list and convert VCN to LCN instead of just reading the attribute body inside the MFT record.

This is how part of the malicious bootloader's NTFS file read/write function looks like.

The code below represents a file system independent function that the bootloader uses to write a file.

Thus the bootkit works with FS as follows: creates a context for the targeted partition where the file is located, searches the file on this partition (volume) parsing the FS structures and creates a context for the found file.

The bootloader also intercepts int 13h in order to replace data of original MBR if someone tries to read it.

After intercepting int 13h, the bootloader transfers control to the original bootstrap code, which it previously read from disk.

```
DISK_ADDRESS_PACKET struc ;
dbSizeOfStructure db
dbUnused          db
dwNumberOfSectors dw
pBuffer           dd
dwStartLBALow     dd
dwStartLBAHigh    dd
DISK_ADDRESS_PACKET ends

push large 0; dwStartLBAHigh
push large 0; dwStartLBALow
push 0x7C00; pBuffer
push 1; dwNumberOfSectors
push 10h ; dbSizeOfStructure
mov si, sp
mov ah, 0x42
int 13h; DS:SI - disk address packet
```

```
    cli
    xor ax, ax
    mov ss, ax
    mov sp, 7B00h
    sti
    mov ds, ax
    mov es, ax
    mov dx, 80h
    push ax
    mov ax, 7C00h
    push ax
    retf; ; go to 0x7C00
```

Unfortunately, while reversing the bootloader, I didn't manage to get a clear answer about how sfc_os.dll is patched, but according to its behavior and indirect evidence, it appears to overwrite the legitimate DLL with a bootkit payload one. One of the proofs is that after infection the payload DLL and sfc_os.dll are identical. Another one is that the payload DLL has sfc_os.dll stub exports, each of which returns the appropriate value of interest to the malware.

Key takeaways

The direct disk access feature is quite unique for malware - it's not clear what advantages it provides for the authors. The rootkit doesn't intercept the disk or disk port driver dispatch functions to hide its malicious sectors so any disk dumper tool can be used to detect these anomalies. One can only assume that the authors decided to rely on that comprehensive list of security products to be disabled rather than on hiding malicious activity in the live system.

Unlike its notorious counterparts such as Tdss (Tidserv) or Rovnix, this bootkit doesn't support its own disk partition and file system to store the malware modules. The original MBR and malware modules are simply written to the end of the disk without any additional preparations. This hints to us that the malware doesn't support a plugin architecture and its features are limited to the original ones implemented in the payload DLL.

MITRE ATT&CK matrix (clickable)

Unfolded version (clickable)

Big thanks to Gabriel Landau for his review and Matthew Hickey, Rong Hwa Chong, Tom Kallo for their feedback. Much appreciated.

Appendix

Used tools

Malware research: IDA Pro, Cerbero PE Insider, Disk Explorer for NTFS, WinDbg legacy, Hiew, VMware running Win7.

Editing pics: Paint, ZoomIt.

Diagrams: Word 365.

Necessary skills

Windows Internals, reverse engineering, malware analysis, forensics.

Fingerprints

First level dropper e49ad00deda88a198f2728a3d276f0b55f892d3088bc861538a005e443d81a92

Main dropper b32cf71e325ceaa8982e6ebed33f95894f2591397e08404368fbaa6dce1095e3

Payload DLL eddbe87f2009cb3199def0845ccf01d0397c126aca6f55e2a9516616825cebb1

Driver (rootkit) 4fdc39276228cab7ef1ef26a084e920760fdaacd78b29e776f09da0a95ae39b0

Bootloader 8eb365237e4cfe478b228d276598ff58c0b133fbcd374024b5903137cf196a3d

For download:

https://www.kernelmode.info/forum/viewtopicf771-3.html?p=14683#p14683

Previous studies

https://zerosecurity.org/2013/06/guntior-bootkit-upgraded/

https://www.kernelmode.info/forum/viewtopicf168-2.html?f=16&t=1765

References

https://thestarman.pcministry.com/asm/mbr/NTFSBR.htm

https://thestarman.pcministry.com/asm/mbr/VistaVBR.htm

https://wiki.osdev.org/ATA_PIO_Mode

http://ntfs.com/ntfs-partition-boot-sector.htm

https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ata/ns-ata-_identify_device_data

https://wiki.osdev.org/ATA_Command_Matrix

https://github.com/microsoft/Windows-driver-samples/blob/main/storage/tools/spti/src/spti.c

https://learn.microsoft.com/en-us/previous-versions/windows/hardware/kernel/ff546644(v=vs.85)

https://wiki.osdev.org/PCI

https://forum.osdev.org/viewtopic.php?f=1&t=30125

https://codemachine.com/downloads/win71/wdm.h

https://reactos.org/forum/viewtopic.php?t=14520

https://systemroot.gitee.io/pages/apiexplorer/d5/d3/pci_8h-source.html

https://mybogi.files.wordpress.com/2011/08/interrupt-13h.pdf

https://en.wikipedia.org/wiki/DOS_memory_management

https://doxygen.reactos.org/d4/d45/drivers_2bus_2pcix_2enum_8c_source.html

Like Comment Share 107 · 6 Comments

Patrick Barker 11mo
Hunt Analyst

Insane. Nice work.

Like · Reply

Matt Suiche 11mo
Crash Dump Connoisseur

Very cool write up!

Like · Reply

Matthew Hickey 11mo
CEO, Founder & Hacker.

Thanks for sharing! Great write up!

Like · Reply

Abhijit Mohanta 11mo
CTO and Co-Founder Intelliroot, Book Author : "Malware Analysis and Detection Engineering" and "Preventing Ransomware". .....

Nice one Artem Baranov

Like · Reply

Rong Hwa Chong 11mo

Congratulations Artem Baranov on your awesome write up! Thanks for sharing deep insights and learnings with the wider community. Thank you!!

Like · Reply | 1 Reaction

See more comments

To view or add a comment, sign in

More articles by this author

Calculating the size of the Windows kernel…
Sep 6, 2024

How Much Code Does Anti-Malware Software…
Aug 13, 2024

A guide to exposing patched Secure Boot…
Jul 18, 2024

W
A

See all

Insights from the community

Computer Networking
How can code obfuscation protect against malware attacks?

Cybersecurity
What are the most effective techniques for malware attribution?

Network Security
How can you detect and analyze packed malware?

Computer Engineering
What are the most effective methods for deobfuscating malware?

Cybersecurity
How do you classify malware samples?

Malware Analysis
How do you handle packed, polymorphic, and metamorphic malware in static analysis?

Show more

Others also viewed

Two reasons you need to use a VM for everyday computing
Adam Winn · 9y

Lucifer's Spawn - New Nightmare!
Tim O. · 4y

Techniques used by Malware to evade AVs
Sagar P. · 1y

EarlyBird Technique: An Advanced Malware Evasion Strategy
KPMG Israel · 1mo

Malware Analysis Adventures: an Agent Tesla variant
István Tóth · 4y

Cybercriminals are using a new form of malware called Flagpro to execute OS commands.
ProTechmanize · 2y

Show more

Explore topics

Sales

Marketing

IT Services

Business Administration

HR Management

Engineering

Soft Skills

See All