How to use Trend Micro's Rootkit Remover to Install a Rootkit

billdemirkapi.me/how-to-use-trend-micros-rootkit-remover-to-install-a-rootkit/

May 18, 2020

Security Research

Bill Demirkapi

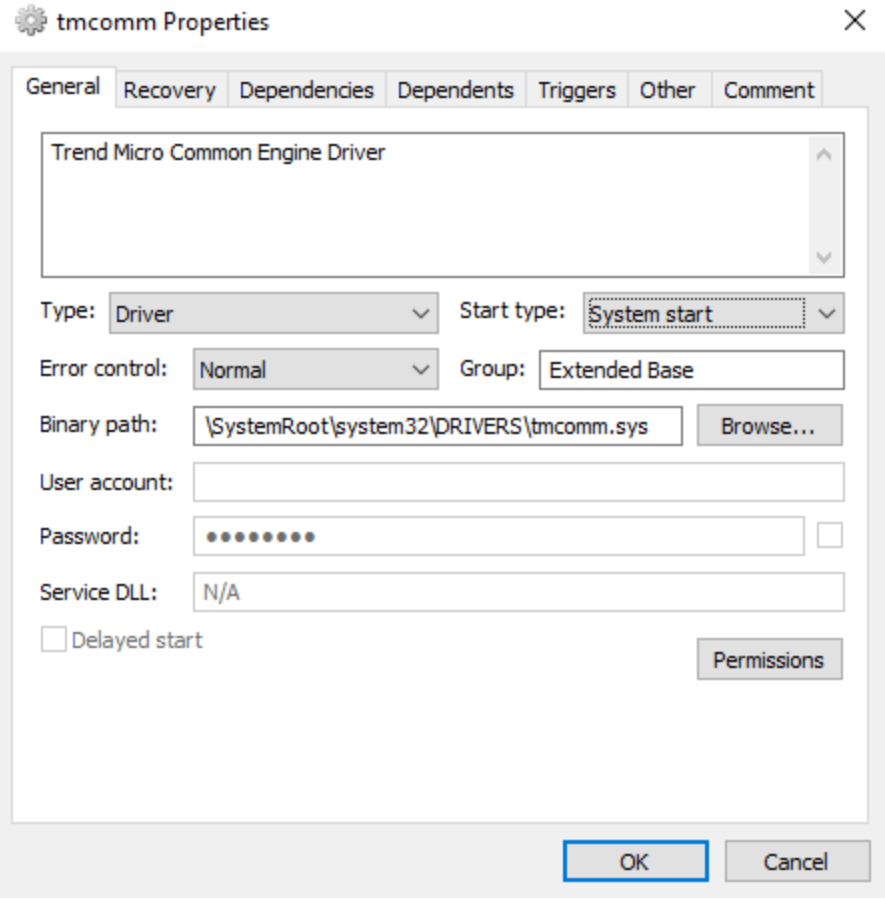
The opinions expressed in this publication are those of the authors. They do not reflect the opinions or views of my employer. All research was conducted independently.

For a recent project, I had to do research into methods rootkits are detected and the most effective measures to catch them when I asked the question, what are some existing solutions to rootkits and how do they function? My search eventually landed me on the TrendMicro RootkitBuster which describes itself as "A free tool that scans hidden files, registry entries, processes, drivers, and the master boot record (MBR) to identify and remove rootkits".

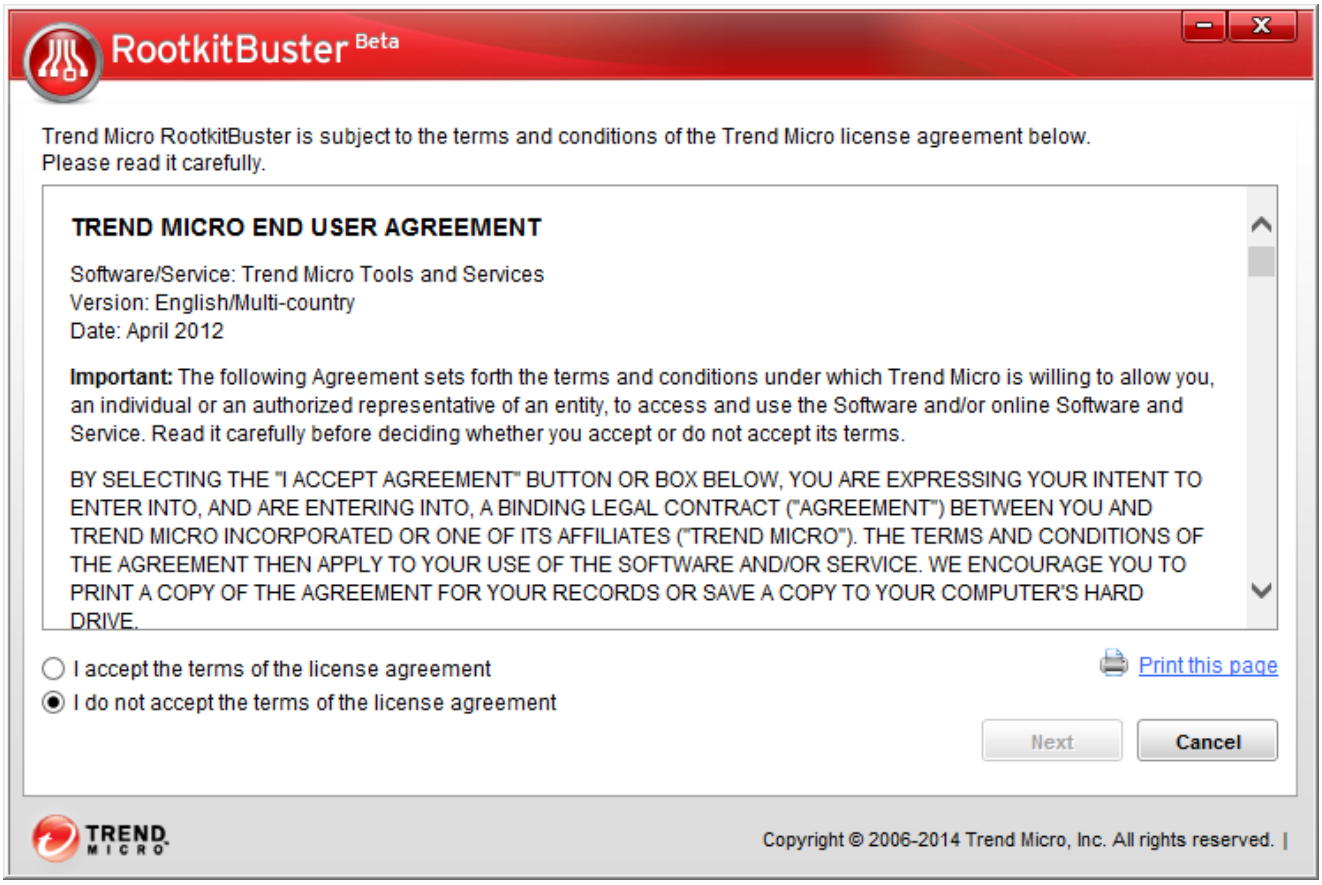
The features it boasted certainly caught my attention. They were claiming to detect several techniques rootkits use to burrow themselves into a machine, but how does it work under the hood and can we abuse it? I decided to find out by reverse engineering core components of the application itself, leading me down a rabbit hole of code that scarred me permanently, to say the least.

Discovery

Starting the adventure, launching the application resulted in a fancy warning by Process Hacker that a new driver had been installed.

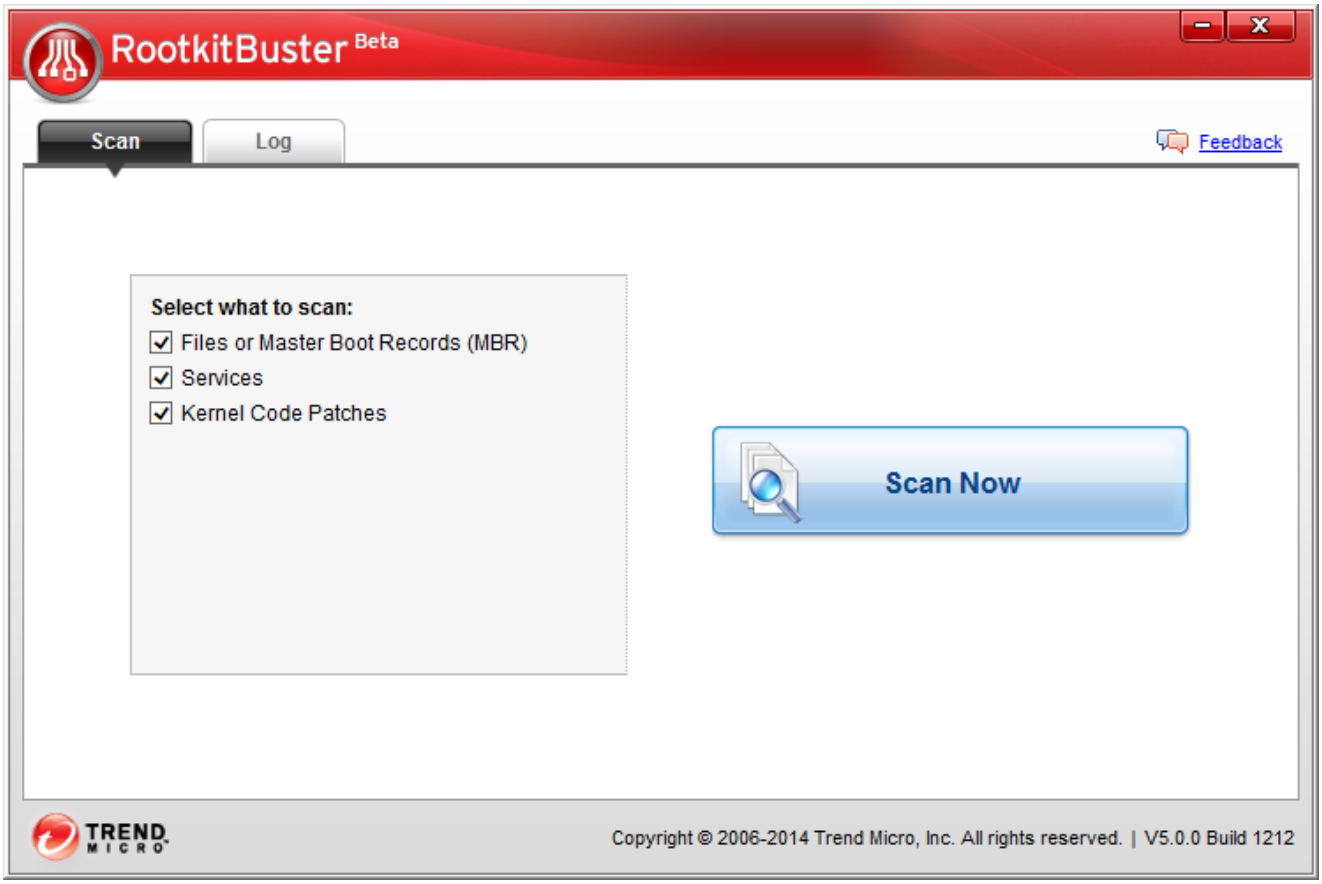


Already off to a good start, we got a copy of Trend Micro's "common driver", this was definitely something to look into. Besides this driver being installed, this friendly window opened prompting me to accept Trend Micro's user agreement.

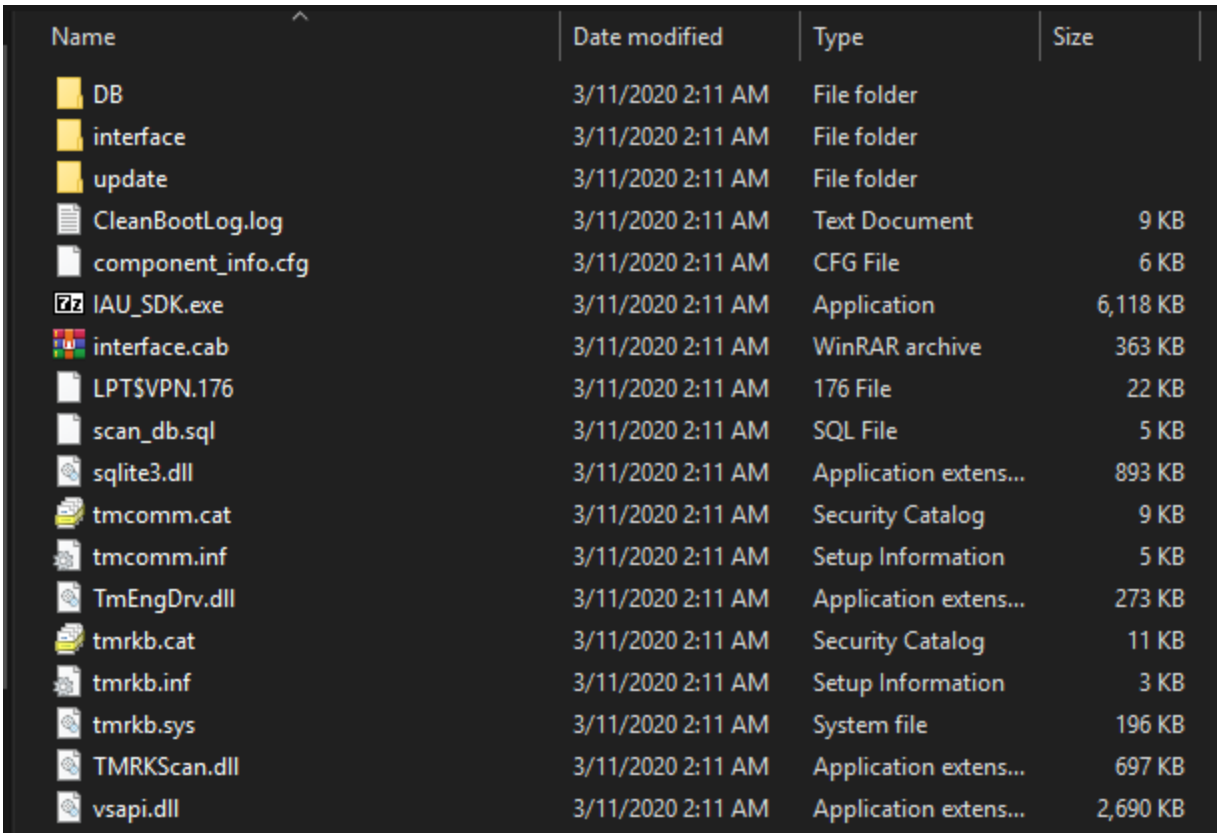


I wasn't in the mood to sign away my soul to the devil just yet, especially since the terms included a clause stating *"You agree not to attempt to reverse engineer, decompile, modify, translate, disassemble, discover the source code of, or create derivative works from..."*.

Thankfully, Trend Micro already deployed their software on to my machine before I accepted any terms. Funnily enough, when I tried to exit the process by right-clicking on the application and pressing "Close Window", it completely evaded the license agreement and went to the main screen of the scanner, even though I had selected the "I do not accept the terms of the license agreement" option. Thanks Trend Micro!



I noticed a quick command prompt flash when I started the application. It turns out this was the result of a 7-Zip Self Extracting binary which extracted the rest of the application components to `%TEMP%\RootkitBuster`.



Name	Date modified	Type	Size
DB	3/11/2020 2:11 AM	File folder	
interface	3/11/2020 2:11 AM	File folder	
update	3/11/2020 2:11 AM	File folder	
CleanBootLog.log	3/11/2020 2:11 AM	Text Document	9 KB
component_info.cfg	3/11/2020 2:11 AM	CFG File	6 KB
IAU_SDK.exe	3/11/2020 2:11 AM	Application	6,118 KB
interface.cab	3/11/2020 2:11 AM	WinRAR archive	363 KB
LPT$VPN.176	3/11/2020 2:11 AM	176 File	22 KB
scan_db.sql	3/11/2020 2:11 AM	SQL File	5 KB
sqlite3.dll	3/11/2020 2:11 AM	Application extens...	893 KB
tmcomm.cat	3/11/2020 2:11 AM	Security Catalog	9 KB
tmcomm.inf	3/11/2020 2:11 AM	Setup Information	5 KB
TmEngDrv.dll	3/11/2020 2:11 AM	Application extens...	273 KB
tmrkb.cat	3/11/2020 2:11 AM	Security Catalog	11 KB
tmrkb.inf	3/11/2020 2:11 AM	Setup Information	3 KB
tmrkb.sys	3/11/2020 2:11 AM	System file	196 KB
TMRKScan.dll	3/11/2020 2:11 AM	Application extens...	697 KB
vsapi.dll	3/11/2020 2:11 AM	Application extens...	2,690 KB

Let's review the driver we'll be covering in this article.

> The `tmcomm` driver which was labeled as the "TrendMicro Common Module" and "Trend Micro Eyes". A quick overlook of the driver indicated that it accepted communication from *privileged* user-mode applications and performed common actions that are not specific to the Rootkit Remover itself. This driver *is not* only used in the Rootkit Buster and is implemented throughout Trend Micro's product line.

In the following sections, we'll be deep diving into the `tmcomm` driver . We'll focus our research into finding different ways to abuse the driver's functionality, with the end goal being able to execute kernel code. I decided not to look into the `tmrkb.sys` because although I am sure it is vulnerable, it seems to only be used for the Rootkit Buster.

TrendMicro Common Module (tmcomm.sys)

Let's begin our adventure with the base driver that appears to be used not only for this Rootkit Remover utility, but several other Trend Micro products as well. As I stated in the previous section, a very brief look-over of the driver revealed that it does allow for communication from *privileged* user-mode applications.

```
status = WdmlibIoCreateDeviceSecure(
           DriverObject,
           0,
           &DeviceName,
           0x22u,
           0x100u,
           0,
           &DefaultSDDLString,              // System and Administrators allowed
           0i64,
           &DeviceObject);
if ( status >= 0 )
{
  status = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
  if ( status >= 0 )
  {
    DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)IOCTLHandler;
    DriverObject->MajorFunction[15] = DriverObject->MajorFunction[14];
    DriverObject->MajorFunction[2] = DriverObject->MajorFunction[15];
    DriverObject->MajorFunction[18] = DriverObject->MajorFunction[2];
    DriverObject->MajorFunction[0] = DriverObject->MajorFunction[18];
    DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
```

One of the first actions the driver takes is to create a device to accept IOCTL communication from user-mode. The driver creates a device at the path `\Device\TmComm` and a symbolic link to the device at `\DosDevices\TmComm` (accessible via `\\.\Global\TmComm`). The driver entrypoint initializes a significant amount of classes and structure used throughout the driver, however, for our purposes, it is not necessary to cover each one.

I was happy to see that Trend Micro made the correct decision of restricting their device to the `SYSTEM` user and Administrators. This meant that even if we did find exploitable code, because any communication would require at least Administrative privileges, a significant amount of the industry would not consider it a vulnerability. For example, Microsoft themselves do not consider Administrator to Kernel to be a security boundary because of the significant amount of access they get. This does not mean however exploitable code in Trend Micro's drivers won't be useful.

```
NTSTATUS __fastcall IOCTLHandler(PDEVICE_OBJECT Device, PIRP Irp)
{
  NTSTATUS status; // [rsp+20h] [rbp-48h]
  UCHAR majorFunction; // [rsp+24h] [rbp-44h]
  struct _IO_STACK_LOCATION *currentStackLocation; // [rsp+28h] [rbp-40h]
  TmIoctlRequest IoctlRequest; // [rsp+30h] [rbp-38h]

  status = 0xC0000001;
  currentStackLocation = CAMSchedule::GetEvent(Irp);
  Irp->IoStatus.Information = 0i64;
  majorFunction = currentStackLocation->MajorFunction;
  if ( currentStackLocation->MajorFunction )
  {
    if ( majorFunction == 2 )                   // IRP_MJ_CLOSE
    {
      IsLastHandle();
      status = 0;
    }
    else if ( majorFunction > 13u )
    {
      if ( majorFunction <= 15u )               // IRP_MJ_DEVICE_CONTROL
      {
        IoctlRequest.UserInputBuffer = currentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer;
        IoctlRequest.UserOutputBuffer = Irp->UserBuffer;
        IoctlRequest.InputSize = currentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
        IoctlRequest.OutputSize = currentStackLocation->Parameters.DeviceIoControl.OutputBufferLength;
        IoctlRequest.BytesWritten = &Irp->IoStatus.Information;
        status = DispatchIoctlRequest(currentStackLocation->Parameters.DeviceIoControl.IoControlCode, &IoctlRequest);
      }
```

TrueApi

A large component of the driver is its "TrueApi" class which is instantiated during the driver's entrypoint. The class contains pointers to *imported* functions that get used throughout the driver. Here is a reversed structure:

```
struct TrueApi
{
        BYTE Initialized;
        PVOID ZwQuerySystemInformation;
        PVOID ZwCreateFile;
        PVOID unk1; // Initialized as NULL.
        PVOID ZwQueryDirectoryFile;
        PVOID ZwClose;
        PVOID ZwOpenDirectoryObjectWrapper;
        PVOID ZwQueryDirectoryObject;
        PVOID ZwDuplicateObject;
        PVOID unk2; // Initialized as NULL.
        PVOID ZwOpenKey;
        PVOID ZwEnumerateKey;
        PVOID ZwEnumerateValueKey;
        PVOID ZwCreateKey;
        PVOID ZwQueryValueKey;
        PVOID ZwQueryKey;
        PVOID ZwDeleteKey;
        PVOID ZwTerminateProcess;
        PVOID ZwOpenProcess;
        PVOID ZwSetValueKey;
        PVOID ZwDeleteValueKey;
        PVOID ZwCreateSection;
        PVOID ZwQueryInformationFile;
        PVOID ZwSetInformationFile;
        PVOID ZwMapViewOfSection;
        PVOID ZwUnmapViewOfSection;
        PVOID ZwReadFile;
        PVOID ZwWriteFile;
        PVOID ZwQuerySecurityObject;
        PVOID unk3; // Initialized as NULL.
        PVOID unk4; // Initialized as NULL.
        PVOID ZwSetSecurityObject;
};
```

Looking at the code, the TrueApi is primarily used as an alternative to calling the functions directly. My educated guess is that Trend Micro is caching these imported functions at initialization to evade *delayed* IAT hooks. Since the TrueApi is resolved by looking at the import table however, if there is a rootkit that hooks the IAT on driver load, this mechanism is useless.

XrayApi

Similar to the TrueApi, the XrayApi is another major class in the driver. This class is used to access several low-level devices and to interact directly with the filesystem. A major component of the XrayConfig is its "config". Here is a partially reverse-engineered structure representing the config data:

```
struct XrayConfigData
{
        WORD Size;
        CHAR pad1[2];
        DWORD SystemBuildNumber;
        DWORD UnkOffset1;
        DWORD UnkOffset2;
        DWORD UnkOffset3;
        CHAR pad2[4];
        PVOID NotificationEntryIdentifier;
        PVOID NtoskrnlBase;
        PVOID IopRootDeviceNode;
        PVOID PpDevNodeLockTree;
        PVOID ExInitializeNPagedLookasideListInternal;
        PVOID ExDeleteNPagedLookasideList;
        CHAR unkpad3[16];
        PVOID KeAcquireInStackQueuedSpinLockAtDpcLevel;
        PVOID KeReleaseInStackQueuedSpinLockFromDpcLevel;
        ...
};
```

The config data stores the location of internal/undocumented variables in the Windows Kernel such as the `IopRootDeviceNode`, `PpDevNodeLockTree`, `ExInitializeNPagedLookasideListInternal`, and `ExDeleteNPagedLookasideList`. My educated guess for the purpose of this class is to access low-level devices directly rather than use documented methods which could be hijacked.

IOCTL Requests

Before we get into what the driver allows us to do, we need to understand how IOCTL requests are handled.

In the primary dispatch function, the Trend Micro driver converts the data alongside a `IRP_MJ_DEVICE_CONTROL` request to a proprietary structure I call a `TmIoctlRequest`.

```
struct TmIoctlRequest
{
        DWORD InputSize;
        DWORD OutputSize;
        PVOID UserInputBuffer;
        PVOID UserOutputBuffer;
        PVOID Unused;
        DWORD_PTR* BytesWritten;
};
```

The way Trend Micro organized dispatching of IOCTL requests is by having several "dispatch tables". The "base dispatch table" simply contains an IOCTL Code and a corresponding "sub dispatch function". For example, when you send an IOCTL request with the code

`0xDEADBEEF`, it will compare each entry of this base dispatch table and pass along the data if there is a table entry that has the matching code. A base table entry can be represented by the structure below:

```
typedef NTSTATUS (__fastcall *DispatchFunction_t)(TmIoctlRequest *IoctlRequest);

struct BaseDispatchTableEntry
{
        DWORD_PTR IOCode;
        DispatchFunction_t DispatchFunction;
};
```

After the `DispatchFunction` is called, it typically verifies some of the data provided ranging from basic `nullptr` checks to checking the size of the input and out buffers. These "sub dispatch functions" then do another lookup based on a code passed in the user input buffer to find the corresponding "sub table entry". A sub table entry can be represented by the structure below:

```
typedef NTSTATUS (__fastcall *OperationFunction_t)(PVOID InputBuffer, PVOID
OutputBuffer);

struct SubDispatchTableEntry
{
        DWORD64 OperationCode;
        OperationFunction_t PrimaryRoutine;
        OperationFunction_t ValidatorRoutine;
};
```

Before calling the `PrimaryRoutine`, which actually performs the requested action, the sub dispatch function calls the `ValidatorRoutine`. This routine does "action-specific" validation on the input buffer, meaning that it performs checks on the data the `PrimaryRoutine` will be using. Only if the `ValidatorRoutine` returns successfully will the `PrimaryRoutine` be called.

Now that we have a basic understanding of how IOCTL requests are handled, let's explore what they allow us to do. Referring back to the definition of the "base dispatch table", which stores "sub dispatch functions", let's explore each base table entry and figure out what each sub dispatch table allows us to do!

IoControlCode == 9000402Bh

Discovery

This first dispatch table appears to interact with the filesystem, but what does that actually mean? To start things off, the code for the "sub dispatch table" entry is obtained by dereferencing a `DWORD` from the start of the input buffer. This means that to specify which

sub dispatch entry you'd like to execute, you simply need to set a `DWORD` at the base of the input buffer to correspond with that entries' `**OperationCode**`.

To make our lives easier, Trend Micro conveniently included a significant amount of debugging strings, often giving an idea of what a function does. Here is a table of the functions I reversed in this sub dispatch table and what they allow us to do.

OperationCode	PrimaryRoutine	Description
2713h	IoControlCreateFile	Calls NtCreateFile, all parameters are controlled by the request.
2711h	IoControlFindNextFile	Returns STATUS_NOT_SUPPORTED.
2710h	IoControlFindFirstFile	Performs nothing, returns STATUS_SUCCESS always.
2712h	IoControlFindCloseFile	Calls ZwClose, all parameters are controlled by the request.
2715h	IoControlReadFileIRPNoCache	References a FileObject using HANDLE from request. Calls IofCallDriver and reads result.
2714h	IoControlCreateFileIRP	Creates a new FileObject and associates DeviceObject for requested drive.
2716h	IoControlDeleteFileIRP	Deletes a file by sending an IRP_MJ_SET_INFORMATION request.
2717h	IoControlGetFileSizeIRP	Queries a file's size by sending an IRP_MJ_QUERY_INFORMATION request.
2718h	IoControlSetFilePosIRP	Set's a file's position by sending an IRP_MJ_SET_INFORMATION request.
2719h	IoControlFindFirstFileIRP	Returns STATUS_NOT_SUPPORTED.
271Ah	IoControlFindNextFileIRP	Returns STATUS_NOT_SUPPORTED.
2720h	IoControlQueryFile	Calls NtQueryInformationFile, all parameters are controlled by the request.

OperationCode	PrimaryRoutine	Description
2721h	IoControlSetInformationFile	Calls NtSetInformationFile, all parameters are controlled by the request.
2722h	IoControlCreateFileOplock	Creates an Oplock via IoCreateFileEx and other filesystem API.
2723h	IoControlGetFileSecurity	Calls NtCreateFile and then ZwQuerySecurityObject. All parameters are controlled by the request.
2724h	IoControlSetFileSecurity	Calls NtCreateFile and then ZwSetSecurityObject. All parameters are controlled by the request.
2725h	IoControlQueryExclusiveHandle	Check if a file is opened exclusively.
2726h	IoControlCloseExclusiveHandle	Forcefully close a file handle.

IoControlCode == 90004027h

Discovery

This dispatch table is primarily used to control the driver's *process* scanning features. Many functions in this sub dispatch table use a separate scanning thread to synchronously search for processes via various methods both documented and undocumented.

OperationCode	PrimaryRoutine	Description
C350h	GetProcessesAllMethods	Find processes via ZwQuerySystemInformation and WorkingSetExpansionLinks.
C351h	DeleteTaskResults*	Delete results obtained through other functions like GetProcessesAllMethods.
C358h	GetTaskBasicResults*	Further parse results obtained through other functions like GetProcessesAllMethods.
C35Dh	GetTaskFullResults*	Completely parse results obtained through other functions like GetProcessesAllMethods.

OperationCode	PrimaryRoutine	Description
C360h	IsSupportedSystem	Returns TRUE if the system is "supported" (whether or not they have hardcoded offsets for your build).
C361h	TryToStopTmComm	Attempt to stop the driver.
C362h	GetProcessesViaMethod	Find processes via a specified method.
C371h	CheckDeviceStackIntegrity	Check for tampering on devices associated with physical drives.
C375h	ShouldRequireOplock	Returns TRUE if oplocks should be used for certain scans.

These IOCTLs revolve around a few structures I call "MicroTask" and "MicroScan". Here are the structures reverse-engineered:

```
struct MicroTaskVtable
{
        PVOID Constructor;
        PVOID NewNode;
        PVOID DeleteNode;
        PVOID Insert;
        PVOID InsertAfter;
        PVOID InsertBefore;
        PVOID First;
        PVOID Next;
        PVOID Remove;
        PVOID RemoveHead;
        PVOID RemoveTail;
        PVOID unk2;
        PVOID IsEmpty;
};

struct MicroTask
{
        MicroTaskVtable* vtable;
        PVOID self1; // ptr to itself.
        PVOID self2; // ptr to itself.
        DWORD_PTR unk1;
        PVOID MemoryAllocator;
        PVOID CurrentListItem;
        PVOID PreviousListItem;
        DWORD ListSize;
        DWORD unk4; // Initialized as NULL.
        char ListName[50];
};

struct MicroScanVtable
{
        PVOID Constructor;
        PVOID GetTask;
};

struct MicroScan
{
        MicroScanVtable* vtable;
        DWORD Tag; // Always 'PANS'.
        char pad1[4];
        DWORD64 TasksSize;
        MicroTask Tasks[4];
};
```

For most of the IOCTLs in this sub dispatch table, a MicroScan is passed in by the client which the driver populates. We'll look into how we can abuse this trust in the next section.

Exploitation

When I was initially reverse engineering the functions in this sub dispatch table, I was quite confused because the code "didn't seem right". It appeared like the `MicroScan` kernel pointer returned by functions such as `GetProcessesAllMethods` was being directly passed onto other functions such as `DeleteTaskResults` by *the client*. These functions would then take this untrusted kernel pointer and with almost no validation call functions in the virtual function table specified at the base of the class.

```
NTSTATUS __fastcall DeleteTaskResults(__int64 InputBuffer)
{
  NTSTATUS status; // [rsp+20h] [rbp-38h]
  MicroScan *scan; // [rsp+30h] [rbp-28h]

  status = 0xC0000008;
  if ( *(_QWORD *)(InputBuffer + 0x10) )
  {
    scan = *(MicroScan **)(InputBuffer + 0x10);
    if ( (int)CheckTag(*(MicroScan **)(InputBuffer + 0x10)) >= 0 )
    {
      if ( scan )
        ((void (__fastcall *)(MicroScan *, __int64))scan->vtable->Constructor)(scan, 1i64);
      status = 0;
    }
  }
  return status;
}
```

Taking a look at the "validation routine" for the `DeleteTaskResults` sub dispatch table entry, the only validation performed on the `MicroScan` instance specified at the input buffer `+ 0x10` was making sure it was a valid kernel address.

```
_BOOL8 __fastcall ValidateDeleteTaskResults(__int64 InputBuffer)
{
  BOOL validKernelAddress; // [rsp+30h] [rbp-18h]

  validKernelAddress = 0;
  if ( InputBuffer )
    validKernelAddress = ValidateAddressWithSize(*(void **)(InputBuffer + 0x10), 0, 0x10ui64, 1u, 0);
  return validKernelAddress;
}
```

```
_BOOL8 __fastcall ValidateAddressWithSize(void *Buffer, BOOLEAN WriteOperation, SIZE_T BufferSize, ULONG Alignment, KPROCESSOR_MODE PreviousMode)
{
  BOOL validBuffer; // [rsp+30h] [rbp-18h]

  validBuffer = 0;
  if ( Buffer && BufferSize && Alignment )
  {
    if ( PreviousMode == 1 )
    {
      if ( WriteOperation )
      {
        if ( WriteOperation != 1 )
          ProbeForRead(Buffer, BufferSize, Alignment);
        ProbeForWrite(Buffer, BufferSize, Alignment);
      }
      else
      {
        ProbeForRead(Buffer, BufferSize, Alignment);
      }
      validBuffer = 1;
    }
    else if ( !((Alignment - 1) & (unsigned int)Buffer) )
    {
      validBuffer = ValidateKernelmodeAddress((unsigned __int64)Buffer, BufferSize);
    }
  }
  return validBuffer;
}
```

```
BOOL8 __fastcall ValidateKernelmodeAddress(unsigned __int64 Pointer, __int64 Size)
{
  return Pointer
      && Size
      && Pointer >= MmSystemRangeStart
      && (unsigned __int8)MmIsAddressValid(Pointer)
      && (unsigned __int8)MmIsAddressValid(Pointer + Size - 1);
}
```

The only other check besides making sure that the supplied pointer was in kernel memory was a simple check in `DeleteTaskResults` to make sure the `Tag` member of the `MicroScan` is `PANS`.

```
NTSTATUS __fastcall CheckTag(MicroScan *a1)
{
  NTSTATUS status; // [rsp+20h] [rbp-18h]

  status = 0;
  if ( (unsigned int)CheckMicroScanTag(a1) != 'PANS' )
    status = 0xC0000008;
  return status;
}
```

Since `DeleteTaskResults` calls the constructor specified in the virtual function table of the `MicroScan` instance, to call an arbitrary kernel function we need to:

1. Be able to allocate at least 10 bytes of kernel memory (for vtable and tag).
2. Control the allocated kernel memory to set the virtual function table pointer and the tag.
3. Be able to determine the address of this kernel memory from user-mode.

Fortunately a mentor of mine, Alex Ionescu, was able to point me in the right direction when it comes to allocating and controlling kernel memory from user-mode. A HackInTheBox Magazine from 2010 had an article by Matthew Jurczyk called "Reserve Objects in Windows 7". This article discussed using APC Reserve Objects, which was introduced in Windows 7, to allocate controllable kernel memory from user-mode. The general idea is that you can queue an Apc to an Apc Reserve Object with the `ApcRoutine` and `ApcArgumentX` members being the data you want in kernel memory and then use `NtQuerySystemInformation` to find the Apc Reserve Object in kernel memory. This reserve object will have the previously specified `KAPC` variables in a row, allowing a user-mode application to control up to 32 bytes of kernel memory (on 64-bit) and know the location of the kernel memory. I would strongly suggest reading the article if you'd like to learn more.

This trick still works in Windows 10, meaning we're able to meet all three requirements. By using an Apc Reserve Object, we can allocate at least 10 bytes for the `MicroScan` structure and bypass the inadequate checks completely. The result? The ability to call arbitrary kernel pointers:

```
Break instruction exception - code 80000003 (first chance)
nt!KeCheckStackAndTargetAddress+0x48:
fffff803`2ec5a1c8 cc              int     3
0: kd> k
 # Child-SP          RetAddr           Call Site
00 fffff905`a84f93d0 fffff803`2ed9d305 nt!KeCheckStackAndTargetAddress+0x48
01 fffff905`a84f9400 fffff803`2edcb1af nt!_C_specific_handler+0x45
02 fffff905`a84f9470 fffff803`2ecfa2d5 nt!RtlpExecuteHandlerForException+0xf
03 fffff905`a84f94a0 fffff803`2ecfe86e nt!RtlDispatchException+0x4a5
04 fffff905`a84f9bf0 fffff803`2edd431d nt!KiDispatchException+0x16e
05 fffff905`a84fa2a0 fffff803`2edd0505 nt!KiExceptionDispatch+0x11d
06 fffff905`a84fa480 00000000`deadbeef nt!KiPageFault+0x445
07 fffff905`a84fa618 fffff803`37473732 0xdeadbeef
08 fffff905`a84fa620 fffff803`3745d7ff tmcomm_sys+0x23732
09 fffff905`a84fa680 fffff803`3746082d tmcomm_sys+0xd7ff
0a fffff905`a84fa6e0 fffff803`37461014 tmcomm_sys+0x1082d
0b fffff905`a84fa730 fffff803`2ed0a939 tmcomm_sys+0x11014
0c fffff905`a84fa7a0 fffff803`2f2b2bd5 nt!IofCallDriver+0x59
0d fffff905`a84fa7e0 fffff803`2f2b29e0 nt!IopSynchronousServiceTail+0x1a5
0e fffff905`a84fa880 fffff803`2f2b1db6 nt!IopXxxControlFile+0xc10
```

Although I provided a specific example of vulnerable code in `DeleteTaskResults`, any of the functions I marked in the table with asterisks are vulnerable. They all trust the kernel pointer specified by the untrusted client and end up calling a function in the `MicroScan` instance's virtual function table.

IoControlCode == 90004033h

Discovery

This next sub dispatch table primarily manages the TrueApi class we reviewed before.

OperationCode	PrimaryRoutine	Description
EA60h	IoControlGetTrueAPIPointer	Retrieve pointers of functions in the TrueApi class.
EA61h	IoControlGetUtilityAPIPointer	Retrieve pointers of utility functions of the driver.
EA62h	IoControlRegisterUnloadNotify*	Register a function to be called on unload.
EA63h	IoControlUnRegisterUnloadNotify	Unload a previously registered unload function.

Exploitation

IoControlRegisterUnloadNotify

This function caught my eye the moment I saw its name in a debug string. Using this sub dispatch table function, an *untrusted client* can register up to 16 arbitrary "unload routines" that get called when the driver unloads. This function's validator routine checks this pointer

from the untrusted client buffer for validity. If the caller is from user-mode, the validator calls `ProbeForRead` on the untrusted pointer. If the caller is from kernel-mode, the validator checks that it is a valid kernel memory address.

This function cannot immediately be used in an exploit from user-mode. The problem is that if we're a user-mode caller, we *must* provide a user-mode pointer, because the validator routine uses `ProbeForRead`. When the driver unloads, this user-mode pointer gets called, but it won't do much because of mitigations such as SMEP. I'll reference this function in a later section, but it is genuinely scary to see an untrusted user-mode client being able to direct a driver to call an arbitrary pointer *by design*.

IoControlCode == 900040DFh

This sub dispatch table is used to interact with the XrayApi. Although the Xray Api is generally used by scans implemented in the kernel, this sub dispatch table provides limited access for the client to interact with physical drives.

OperationCode	PrimaryRoutine	Description
15F90h	IoControlReadFile	Read a file directly from a disk.
15F91h	IoControlUpdateCoreList	Update the kernel pointers used by the Xray Api.
15F92h	IoControlGetDRxMapTable	Get a table of drives mapped to their corresponding devices.

IoControlCode == 900040E7h

Discovery

The final sub dispatch is used to scan for hooks in a variety of system structures. It was interesting to see the variety of hooks Trend Micro checks for including hooks in object types, major function tables, and even function inline hooks.

OperationCode	PrimaryRoutine	Description
186A0h	TMXMSCheckSystemRoutine	Check a few system routines for hooks.
186A1h	TMXMSCheckSystemFileIO	Check file IO major functions for hooks.
186A2h	TMXMSCheckSpecialSystemHooking	Check the file object type and ntoskrnl Io functions for hooks.

OperationCode	PrimaryRoutine	Description
186A3h	TMXMSCheckGeneralSystemHooking	Check the Io manager for hooks.
186A4h	TMXMSCheckSystemObjectByName	Recursively trace a system object (either a directory or symlink).
186A5h	TMXMSCheckSystemObjectByName2*	Copy a system object into user-mode memory.

Exploitation

Yeah, `TMXMSCheckSystemObjectByName2` is as bad as it sounds. Before looking at the function directly, here's a few reverse engineered structures used later:

```
struct CheckSystemObjectParams
{
    PVOID Src;
    PVOID Dst;
    DWORD Size;
    DWORD* OutSize;
};

struct TXMSParams
{
    DWORD OutStatus;
    DWORD HandlerID;
    CHAR unk[0x38];
    CheckSystemObjectParams* CheckParams;
};
```

`TMXMSCheckSystemObjectByName2` takes in a Source pointer, Destination pointer, and a Size in bytes. The validator function called for `TMXMSCheckSystemObjectByName2` checks the following:

- `ProbeForRead` on the `CheckParams` member of the `TXMSParams` structure.
- `ProbeForRead` and `ProbeForWrite` on the `Dst` member of the `CheckSystemObjectParams` structure.

Essentially, this means that we need to pass a valid `CheckParams` structure and the `Dst` pointer we pass is in user-mode memory. Now let's look at the function itself:

```
signed __int64 __fastcall TMXMSCheckSystemObjectByName2(TXMSParams *inbuf)
{
  unsigned int i; // [rsp+20h] [rbp-38h]
  DWORD Size; // [rsp+24h] [rbp-34h]
  PVOID Src; // [rsp+30h] [rbp-28h]
  CheckSystemObjectParams *params; // [rsp+38h] [rbp-20h]
  PVOID Dst; // [rsp+48h] [rbp-10h]

  params = inbuf->CheckParams;
  Src = params->Src;
  Dst = params->Dst;
  Size = params->Size;
  if ( params->Src <= (PVOID)MmSystemRangeStart )
    return 0xC000000Di64;                       // STATUS_INVALID_PARAMETER
  for ( i = 0; i < (unsigned int)((((unsigned __int16)Src & 0xFFF) + (unsigned __int64)Size + 4095) >> 12); ++i )
  {
    if ( !(unsigned __int8)MmIsAddressValid((char *)Src + 4096 * i) )
      return 0xC000000Di64;                     // STATUS_INVALID_PARAMETER
  }
  memmove(Dst, Src, Size);
  *params->OutSize = Size;
  return 0i64;
}
```

Although that for loop may seem scary, all it is doing is an optimized method of checking a range of kernel memory. For every memory page in the range `Src` to `Src + Size`, the function calls `MmIsAddressValid`. The real scary part is the following operations:

```
20  memmove(Dst, Src, Size);
21  *params->OutSize = Size;
```

These lines take an untrusted `Src` pointer and copies `Size` bytes to the untrusted `Dst` pointer... yikes. We can use the `memmove` operations to read an arbitrary kernel pointer, but what about writing to an arbitrary kernel pointer? The problem is that the validator for `TMXMSCheckSystemObjectByName2` requires that the destination is user-mode memory. Fortunately, there is another bug in the code.

The next `*params->OutSize = Size;` line takes the `Size` member from our structure and places it at the pointer specified by the untrusted `OutSize` member. No verification is done on what `OutSize` points to, thus we can write up to a DWORD each IOCTL call. One caveat is that the `Src` pointer would need to point to valid kernel memory for up to `Size` bytes. To meet this requirement, I just passed the base of the `ntoskrnl` module as the source.

Using this arbitrary write primitive, we can use the previously found unload routines trick to execute code. Although the validator routine prevents us from passing in a kernel pointer if we're calling from user-mode, we don't actually need to go through the validator. Instead, we can write to the unload routine array inside of the driver's `.data` section using our write primitive and place the pointer we want.

Really, *really* bad code

Typically, I like sticking to strictly security in my blog posts, but this driver made me break that tradition. In this section, we won't be covering the security issues of the driver, rather the terrible code that's used by millions of Trend Micro customers around the world.

Bruteforcing Processes

```
14  csrssPid = 0i64;
15  currentProcess = 0i64;
16  strcpy(&csrssName, "csrss.exe");
17  LODWORD(CsrssLen) = 9;
18  for ( i = 0; i < 0x10000 && !csrssPid; i += 4 )
19  {
20    currentProcess = 0i64;
21    currentPid = ReturnFirstArg(i);
22    PsLookupProcessByProcessId(currentPid, &currentProcess);
23    if ( currentProcess )
24    {
25      currentProcessName = (char *)UtilGetProcessName(currentProcess, 0i64, 0i64);
26      if ( currentProcessName )
27      {
28        if ( !strnicmp(&csrssName, currentProcessName, (unsigned int)CsrssLen) )
29        {
30          Handle = 0i64;
31          LOBYTE(AccessMode) = 0;               // KernelMode
32          if ( (signed int)ObOpenObjectByPointer(currentProcess, 512i64, 0i64, 0i64, PsProcessType, AccessMode, &Handle) >= 0 )
33          {
34            if ( Handle )
35            {
36              SessionId = 0;
37              HIDWORD(CsrssLen) = 0;
38              if ( (signed int)ZwQueryInformationProcess(Handle, 0x18i64, &SessionId, 4i64, (char *)&CsrssLen + 4) >= 0// PROCESS_SESSION_INFORMATION
39                && !SessionId )
40              {
41                csrssPid = ReturnFirstArg(i);
42              }
43              ZwClose(Handle);
44            }
45          }
46        }
47      }
48      ObfDereferenceObject(currentProcess);
49    }
50  }
51  return csrssPid;
52}
```

Let's take a look at what's happening here. This function has a for loop from 0 to 0x10000, incrementing by 4, and retrieves the object of the process behind the current index (if there is one). If the index does match a process, the function checks if the name of the process is `csrss.exe`. If the process is named `csrss.exe`, the final check is that the session ID of the process is 0. Come on guys, there is literally documented API to enumerate processes from kernel... what's the point of bruteforcing?

Bruteforcing Offsets

EPROCESS ImageFileName Offset

```
__int64 GetImageNameOffset()
{
  unsigned int i; // [rsp+20h] [rbp-18h]
  unsigned int MaxCount; // [rsp+24h] [rbp-14h]
  PEPROCESS SystemProcess; // [rsp+28h] [rbp-10h]

  SystemProcess = IoGetCurrentProcess();
  MaxCount = strlen("System");
  for ( i = 0; i < 0x1000; ++i )
  {
    if ( !strnicmp("System", (const char *)SystemProcess + i, MaxCount) )
    {
      ImageNameOffset = i;
      return (unsigned int)ImageNameOffset;
    }
  }
  return (unsigned int)ImageNameOffset;
}
```

When I first saw this code, I wasn't sure what I was looking at. The function takes the current process, which happens to be the System process since this is called in a System thread, then it searches for the string "System" in the first 0x1000 bytes. What's happening is... Trend Micro is bruteforcing the `ImageFileName` member of the `EPROCESS` structure by looking for the known name of the System process inside of its `EPROCESS` structure. If you wanted the `ImageFileName` of a process, just use `ZwQueryInformationProcess` with the `ProcessImageFileName` class...

EPROCESS Peb Offset

```
__int64 __fastcall GetProcessPebOffset(__int64 csrssPid, _QWORD *pebOffset)
{
  unsigned int i; // [rsp+20h] [rbp-68h] MAPDST
  PEPROCESS csrssProcess; // [rsp+30h] [rbp-58h]
  PVOID csrssProcessPeb; // [rsp+40h] [rbp-48h]
  TmProcessInfo csrssInfo; // [rsp+48h] [rbp-40h]

  i = 0;
  GenerateTmProcessInfo(&csrssInfo, csrssPid);
  if ( pebOffset )
    *pebOffset = 0i64;
  if ( (unsigned int)IsCsrssProcessValid(&csrssInfo) )// checks if process object is not NULL
  {
    csrssProcess = GetProcessObject(&csrssInfo);
    csrssProcessPeb = GetProcessPeb(&csrssInfo);
    for ( i = 0; i < 0x2000; i += 8 )
    {
      if ( *(PVOID *)((char *)csrssProcess + i) == csrssProcessPeb )
      {
        if ( pebOffset )
          *pebOffset = csrssProcess;
        break;
      }
    }
  }
  CleanTmProcessInfo(&csrssInfo);
  return i;
}
```

In this function, Trend Micro uses the PID of the `csrss` process to brute force the `Peb` member of the `EPROCESS` structure. The function retrieves the `EPROCESS` object of the `csrss` process by using `PsLookupProcessByProcessId` and it retrieves the `PebBaseAddress` by using `ZwQueryInformationProcess`. Using those pointers, it tries every offset from 0 to 0x2000 that matches the known `Peb` pointer. What's the point of finding the offset of the `Peb` member when you can just use `ZwQueryInformationProcess`, as you already do...

ETHREAD StartAddress Offset

```
18  if ( *(_QWORD *)((char *)currentThread + (unsigned int)ThreadStartAddressOffset) != KnownStartAddress )
19  {
20    for ( j = 0; j < 0x1000; j += 8 )
21    {
22      if ( *(_QWORD *)((char *)currentThread + j) == KnownStartAddress )
23      {
24        ThreadStartAddressOffset = j;
25        if ( DebugLogInstance )
26        {
27          if ( CheckMicroScanTag(DebugLogInstance) & 0x10 )
28            CDebugLogEx::Write(
29              (CDebugLogEx *)DebugLogInstance,
30              "UtilGetThreadStartAddressOffset(): _ethread.StartAddress=%#x",
31              j);
32        }
33        return;
34      }
35    }
36  }
37}
```

Here Trend Micro uses the current system thread with a known start address to brute force the `StartAddress` member of the `ETHREAD` structure. Another case where finding the raw offset is unnecessary. There is a semi-documented class of `ZwQueryInformationThread` called `ThreadQuerySetWin32StartAddress` which gives you the start address of a thread.

Unoptimized Garbage

When I initially decompiled this function, I thought IDA Pro might be simplifying a `memset` operation, because all this function was doing was setting all of the `TrueApi` structure members to zero. I decided to take a peek at the assembly to confirm I wasn't missing something...

```
TrueApi *__fastcall InitTrueApi(TrueApi *a1)
{
  a1->Initialized = 0;
  a1->ZwQuerySystemInformation = 0i64;
  a1->ZwCreateFile = 0i64;
  a1->unk1 = 0i64;
  a1->ZwQueryDirectoryFile = 0i64;
  a1->ZwClose = 0i64;
  a1->ZwOpenDirectoryObjectWrapper = 0i64;
  a1->ZwQueryDirectoryObject = 0i64;
  a1->ZwDuplicateObject = 0i64;
  a1->unk3 = 0i64;
  a1->ZwOpenKey = 0i64;
  a1->ZwEnumerateKey = 0i64;
  a1->ZwEnumerateValueKey = 0i64;
  a1->ZwCreateKey = 0i64;
  a1->ZwQueryValueKey = 0i64;
  a1->ZwQueryKey = 0i64;
  a1->ZwDeleteKey = 0i64;
  a1->ZwTerminateProcess = 0i64;
  a1->ZwOpenProcess = 0i64;
  a1->ZwSetValueKey = 0i64;
  a1->ZwDeleteValueKey = 0i64;
  a1->ZwCreateSection = 0i64;
  a1->ZwQueryInformationFile = 0i64;
  a1->ZwSetInformationFile = 0i64;
  a1->ZwMapViewOfSection = 0i64;
  a1->ZwUnmapViewOfSection = 0i64;
  a1->ZwReadFile = 0i64;
  a1->ZwWriteFile = 0i64;
  a1->ZwQuerySecurityObject = 0i64;
  a1->unk4 = 0i64;
  a1->unk5 = 0i64;
  a1->ZwSetSecurityObject = 0i64;
  return a1;
}
```

```
.text:0000000180026004                mov     [rsp+arg_0], rcx
.text:0000000180026009                mov     rax, [rsp+arg_0]
.text:000000018002600E                mov     byte ptr [rax], 0
.text:0000000180026011                mov     rax, [rsp+arg_0]
.text:0000000180026016                mov     qword ptr [rax+8], 0
.text:000000018002601E                mov     rax, [rsp+arg_0]
.text:0000000180026023                mov     qword ptr [rax+10h], 0
.text:000000018002602B                mov     rax, [rsp+arg_0]
.text:0000000180026030                mov     qword ptr [rax+18h], 0
.text:0000000180026038                mov     rax, [rsp+arg_0]
.text:000000018002603D                mov     qword ptr [rax+20h], 0
.text:0000000180026045                mov     rax, [rsp+arg_0]
.text:000000018002604A                mov     qword ptr [rax+28h], 0
.text:0000000180026052                mov     rax, [rsp+arg_0]
.text:0000000180026057                mov     qword ptr [rax+30h], 0
.text:000000018002605F                mov     rax, [rsp+arg_0]
.text:0000000180026064                mov     qword ptr [rax+38h], 0
.text:000000018002606C                mov     rax, [rsp+arg_0]
.text:0000000180026071                mov     qword ptr [rax+40h], 0
.text:0000000180026079                mov     rax, [rsp+arg_0]
.text:000000018002607E                mov     qword ptr [rax+48h], 0
.text:0000000180026086                mov     rax, [rsp+arg_0]
.text:000000018002608B                mov     qword ptr [rax+50h], 0
.text:0000000180026093                mov     rax, [rsp+arg_0]
.text:0000000180026098                mov     qword ptr [rax+58h], 0
.text:00000001800260A0                mov     rax, [rsp+arg_0]
.text:00000001800260A5                mov     qword ptr [rax+60h], 0
.text:00000001800260AD                mov     rax, [rsp+arg_0]
.text:00000001800260B2                mov     qword ptr [rax+68h], 0
.text:00000001800260BA                mov     rax, [rsp+arg_0]
.text:00000001800260BF                mov     qword ptr [rax+70h], 0
.text:00000001800260C7                mov     rax, [rsp+arg_0]
.text:00000001800260CC                mov     qword ptr [rax+78h], 0
.text:00000001800260D4                mov     rax, [rsp+arg_0]
.text:00000001800260D9                mov     qword ptr [rax+80h], 0
.text:00000001800260E4                mov     rax, [rsp+arg_0]
.text:00000001800260E9                mov     qword ptr [rax+88h], 0
.text:00000001800260F4                mov     rax, [rsp+arg_0]
.text:00000001800260F9                mov     qword ptr [rax+90h], 0
.text:0000000180026104                mov     rax, [rsp+arg_0]
.text:0000000180026109                mov     qword ptr [rax+98h], 0
.text:0000000180026114                mov     rax, [rsp+arg_0]
.text:0000000180026119                mov     qword ptr [rax+0A0h], 0
.text:0000000180026124                mov     rax, [rsp+arg_0]
```

Yikes... looks like someone turned off optimizations.

Cheating Microsoft's WHQL Certification

So far we've covered methods to read and write arbitrary kernel memory, but there is one step missing to install our own rootkit. Although you could execute kernel shellcode with just a read/write primitive, I generally like finding the path of least resistance. Since this is a third-party driver, chances are, there is some `NonPagedPool` memory allocated which we can use to host and execute our malicious shellcode.

Let's take a look at how Trend Micro allocates memory. Early in the entrypoint of the driver, Trend Micro checks if the machine is a "supported system" by checking the major version, minor version, and the build number of the operating system. Trend Micro does this because they hardcode several offsets which can change between builds.

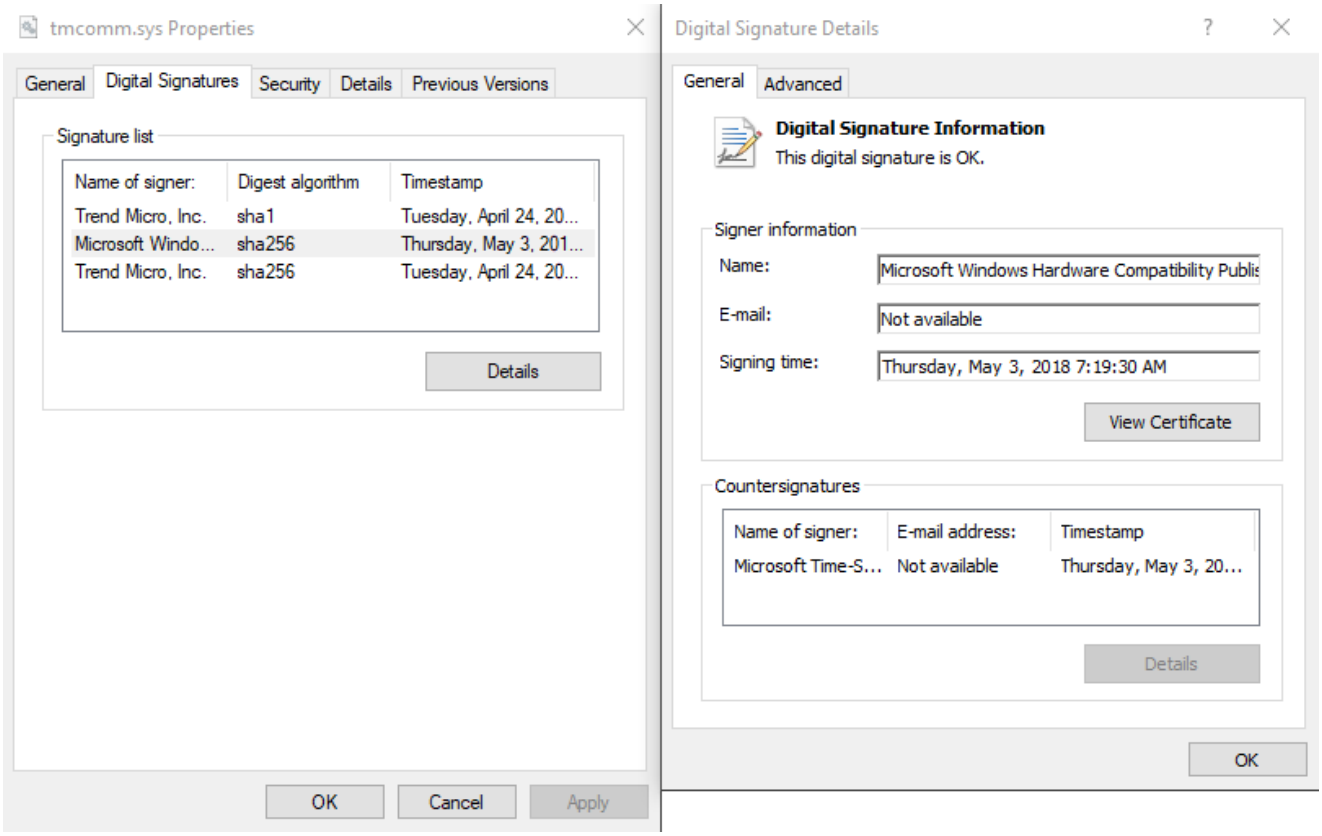
Fortunately, the `PoolType` global variable which is used to allocate non-paged memory is set to 0 (`NonPagedPool`) by default. I noticed that although this value was 0 initially, the variable was still in the `.data` section, meaning it could be changed. When I looked at what wrote to the variable, I saw that the same function responsible for checking the operating system's version also set this `PoolType` variable in certain cases.

```
83  if ( versionInfo.dwMajorVersion >= 10 && MysteriousCheck() )
84  {
85    PoolType = 512;                             // NonPagedPoolNx
86    MemoryPriority |= 0x40000000u;
87  }
```

From a brief glance, it looked like if our operating system is Windows 10 or a newer version, the driver prefers to use `NonPagedPoolNx` . Good from a security standpoint, but bad for us. This is used for all non-paged allocations, meaning we would have to find a spare `ExAllocatePoolWithTag` that had a hardcoded `NonPagedPool` argument otherwise we couldn't use the driver's allocated memory on Windows 10. But, it's not that straightforward. What about `MysteriousCheck()` , the second requirement for this if statement?

```
19  RtlInitUnicodeString(
20    &DestinationString,
21    L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\Session Manager\\Memory Management");
22  ObjectAttributes.Length = 48;
23  ObjectAttributes.RootDirectory = 0i64;
24  ObjectAttributes.Attributes = 704;
25  ObjectAttributes.ObjectName = &DestinationString;
26  ObjectAttributes.SecurityDescriptor = 0i64;
27  ObjectAttributes.SecurityQualityOfService = 0i64;
28  status = ZwOpenKey(&KeyHandle, 1u, &ObjectAttributes);
29  if ( KeyHandle && status >= 0 )
30  {
31    ResultLength = 0;
32    RtlInitUnicodeString(&ValueName, L"VerifyDriverLevel");
33    status = ZwQueryValueKey(KeyHandle, &ValueName, KeyValueFullInformation, 0i64, 0, &ResultLength);
55        if ( status >= 0 )
56        {
57          if ( KeyValueInformation->Type == 4 ) // REG_DWORD
58          {
59            valueptr = (int *)((char *)KeyValueInformation + KeyValueInformation->DataOffset);
60            value = *valueptr;
61          }
62          if ( value & 0x2000000 )
63            result = 1;
```

What `MysteriousCheck()` was doing was checking if Microsoft's Driver Verifier was enabled... Instead of just using `NonPagedPoolNx` on Windows 8 or higher, **Trend Micro placed an explicit check to only use secure memory allocations if they're being watched**. Why is this not just an example of bad code? Trend Micro's driver is WHQL certified:



Passing Driver Verifier has been a long-time requirement of obtaining WHQL certification. On Windows 10, Driver Verifier enforces that drivers do not allocate executable memory. Instead of complying with this requirement designed to secure Windows users, **Trend Micro decided to ignore their user's security and designed their driver to cheat any testing or debugging environment which would catch such violations**.

Honestly, I'm dumbfounded. I don't understand why Trend Micro would go *out of their way* to cheat in these tests. Trend Micro could have just left the Windows 10 check, why would you even bother creating an explicit check for Driver Verifier? The only working theory I have is that for some reason most of their driver is not compatible with `NonPagedPoolNx` and that only their entrypoint is compatible, otherwise there really isn't a point.

Delivering on my promise

As I promised, you can use Trend Micro's driver to install your own rootkit. Here is what you need to do:

1. Find any `NonPagedPool` allocation by the driver. As long as you don't have Driver Verifier running, you can use any of the non-paged allocations that have their pointers stored in the `.data` section. Preferably, pick an allocation that isn't used often.
2. Write your kernel shellcode anywhere in the memory allocation using the arbitrary kernel write primitive in `TMXMSCheckSystemObjectByName2`.
3. Execute your shellcode by registering an unload routine (directly in `.data`) or using the several other execution methods present in the `90004027h` dispatch table.

It's really as simple as that.

Conclusion

I reverse *a lot* of drivers and you do typically see some pretty dumb stuff, but I was shocked at many of the findings in this article coming from a company such as Trend Micro. Most of the driver feels like proof-of-concept garbage that is held together by duct tape.

Although Trend Micro has taken basic precautionary measures such as restricting who can talk to their driver, a significant amount of the code inside of the IOCTL handlers includes very risky DKOM. Also, I'm not sure how certain practices such as bruteforcing anything would get through adequate code review processes. For example, the Bruteforcing Processes code doesn't make sense, are Trend Micro developers not aware of enumerating processes via `ZwQuerySystemInformation`? What about disabling optimizations? Anti-virus already gets flak for slowing down client machines, why would you intentionally make your driver slower? To add insult to injury, this driver is used in several Trend Micro products, not just their rootkit remover. All I know going forward is that I won't be a Trend Micro customer anytime soon.