Rootkit Analysis: Hiding SSDT hooks

Written by: Nick Jogie

System Service Descriptor Table (SSDT) patching has been widely used by rootkits and is usually easily detected. BlackEnergy version 2 has implemented a technique which successfully hides from basic rootkit detection. Basic rootkit detectors typically only check address ranges, on function pointers, listed in the SSDT. If the pointers are outside the kernel address range, it implies that the SSDT is hooked.

The following will illustrate a procedural check, used to uncover this technique, using a kernel debugger.

Checking for corruption-

The first step would be to check for discrepancies in any executable image loaded in memory. This is accomplished by using WinDbg's !chkimg extension.

Command: "!for_each_module !chkimg @#ModuleName"

The above command will loop through all loaded modules on the system and perform a check against the symbol store.

```
lkd> !chkimg -d nt
  804ded5a-804ded5c  3 bytes - nt!KiBBTUnexpectedRange+8
         [ 00 ff 09:6b f0 d5 ]
3 errors : nt (804ded5a-804ded5c)
```

The result above shows corruption in the "nt"(ntoskrnl.exe) module. The nature of this artifact seems to indicate a possible correction for an out-of-range value. As you will later find out, it was most likely introduced because of the new ServiceTables added by the rootkit.

Normal Output:

```
lkd> u nt!KiBBTUnexpectedRange+8
nt!KiBBTUnexpectedRange+0x8:
804ded5a 00ff          add   bh,bh
804ded5c 0900          or    dword ptr [eax],eax
804ded5e 0bc0          or    eax,eax
804ded60 58            pop   eax
804ded61 5a            pop   edx
804ded62 8bec          mov   ebp,esp
804ded64 89ae34010000  mov   dword ptr [esi+134h],ebp
804ded6a 0f8490020000  je    nt!KiFastCallEntry+0x8d (804df000)
804ded70 8d15509b5580  lea   edx,[nt!KeServiceDescriptorTableShadow+0x10
```

Corrupted Output:

```
lkd> u nt!KiBBTUnexpectedRange+8
nt!KiBBTUnexpectedRange+0x8:
804ded5a 6bf0d5        imul  esi,eax,0FFFFFFD5h
804ded5d 000b          add   byte ptr [ebx],cl
804ded5f c0585a8b      rcr   byte ptr [eax+5Ah],8Bh
804ded63 ec            in    al,dx
804ded64 89ae34010000  mov   dword ptr [esi+134h],ebp
804ded6a 0f8490020000  je    nt!KiFastCallEntry+0x8d (804df000)
804ded70 8d15509b5580  lea   edx,[nt!KeServiceDescriptorTableShadow+0x10
```

Since this was the only corruption identified, we need to look for additional indicators.

Checking the SSDT-

The SSDT is used for dispatching system calls either from INT 0x2E or SYSENTER.

The SSDT uses a structure called the System Service Table (SST).

SST Structure:

```
typedef struct _SYSTEM_SERVICE_TABLE
{
 PDWORD ServiceTable;       //Function pointer array
 PDWORD CounterTable;       //Not used in free build
 DWORD  ServiceLimit;       //Number of entries in array
 PBYTE  ArgumentTable;      //Array of arguments
}SYSTEM_SERVICE_TABLE;
```

The SST structure is used by two different tables in the kernel:

1. KeServiceDescriptorTable

```
lkd> dps nt!KeServiceDescriptorTable
80559b80  804e2d20 nt!KiServiceTable
80559b84  00000000
80559b88  0000011c
80559b8c  804d8f48 nt!KiArgumentTable
```

2. KeServiceDescriptorTableShadow

```
lkd> dps nt!KeServiceDescriptorTableShadow
80559b50  bf997600 win32k!W32pServiceTable
80559b54  00000000
80559b58  0000029b
80559b5c  bf998310 win32k!W32pArgumentTable
```

The ServiceTable field is a pointer to a linear array. The addresses contained in this array are the entry points to kernel routines. This array is known as the SSDT.

```
lkd> dps nt!KiServiceTable
804e2d20  80586691 nt!NtAcceptConnectPort
804e2d24  805706ef nt!NtAccessCheck
804e2d28  80579b71 nt!NtAccessCheckAndAuditAlarm
804e2d2c  80580b5c nt!NtAccessCheckByType
804e2d30  80598ff7 nt!NtAccessCheckByTypeAndAuditAlarm
804e2d34  80636b80 nt!NtAccessCheckByTypeResultList
804e2d38  80638d05 nt!NtAccessCheckByTypeResultListAndAuditAlarm
…
```

```
lkd> dps win32k!W32pServiceTable
```

```
bf997600  bf934ffe win32k!NtGdiAbortDoc
bf997604  bf946a92 win32k!NtGdiAbortPath
bf997608  bf8bf295 win32k!NtGdiAddFontResourceW
bf99760c  bf93e718 win32k!NtGdiAddRemoteFontToDC
bf997610  bf9480a9 win32k!NtGdiAddFontMemResourceEx
bf997614  bf935262 win32k!NtGdiRemoveMergeFont
bf997618  bf935307 win32k!NtGdiAddRemoteMMInstanceToDC
…
```

Dumping KeServiceDescriptorTable:

```
lkd> dds poi(nt!KeServiceDescriptorTable) L
poi(nt!KeServiceDescriptorTable+8)
```

Dumping KeServiceDescriptorTableShadow:

```
lkd> dds poi(nt!KeServiceDescriptorTableShadow+10) L
poi(nt!KeServiceDescriptorTableShadow+18)
```

The CounterTable is not used in the free build version of Windows.
The ServiceLimit holds the size of the SSDT array.
The ArgumentTable is a pointer to the System Service Parameter Table (SSPT). SSPT is an array which indicates the amount of space allocated for the function argument related to the SSDT routine.

Looking at the results from the analysis of the SSDT thus far, nothing seems unusual. All addresses seem to be in range and matching their respective symbols. A basic rootkit detector would report the same results we've just analyzed.

What technique could still manage to hook the SSDT and bypass basic rootkit detectors?

Let's investigate!

The SST itself is a member of another structure called Service Descriptor Table (SDT).

SDT Structure:

```
typedef struct _SERVICE_DESCRIPTOR_TABLE
{
        SYSTEM_SERVICE_TABLE ntoskrnl; //SSDT
        SYSTEM_SERVICE_TABLE win32k; // Shadow SSDT
        SYSTEM_SERVICE_TABLE Table3; // empty
        SYSTEM_SERVICE_TABLE Table4; // empty
}
```

It seems that there are two empty tables in the SDT.
Interestingly enough, these are reserved to allow other device drivers to add their own SSTs. Internet Information Services (IIS), for example, uses Spud.sys which calls KeAddSystemServiceTable to add its own kernel routines.

Seeing that adding new service tables is possible, how would applications access it?

ServiceTable Pointers-

The answer would be in the KTHREAD structure. Each thread has a pointer to a

ServiceTable which is set by KeInitThread. Also, depending on if the thread needs the GUI functions contained in the Shadow SSDT, PsConvertToGuiThread is called.

KTHREAD Structure:

```
lkd> dt -v nt!_KTHREAD
ntdll!_KTHREAD
struct _KTHREAD, 73 elements, 0x1c0 bytes
   +0x000 Header          : struct _DISPATCHER_HEADER, 6 elements, 0x10 bytes
   +0x010 MutantListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
   +0x018 InitialStack    : Ptr32 to Void
   +0x01c StackLimit      : Ptr32 to Void
   +0x020 Teb             : Ptr32 to Void
…
   +0x0e0 ServiceTable    : Ptr32 to Void
```

Legitimate threads would either point to KeServiceDescriptorTable or KeServiceDescriptorTableShadow. Any discrepancy should be investigated.

Command:

```
!for_each_thread ".echo Thread: @#Thread; dt nt!_kthread ServiceTable @#Thread"
```

Below are the parsed results for unique ServiceTables in active threads:

```
Thread: 0xffffffff812edda8
   +0x0e0 ServiceTable : 0x80559b80 (KeServiceDescriptorTable)
…
Thread: 0xffffffffffbaa2d0
   +0x0e0 ServiceTable : 0x80559b50 (KeServiceDescriptorTableShadow)
…
Thread: 0xffffffffffa317e0
   +0x0e0 ServiceTable : 0xffa07a68 (New table???)
…
Thread: 0xffffffffffa43da8
   +0x0e0 ServiceTable : 0x81133230 (New Table???)
```

It seems that there are two new tables, which are being reference, we were not aware of.

Let's take a closer look.

```
lkd> dps ffa07a68
ffa07a68  81267918
ffa07a6c  00000000
ffa07a70  0000011c
ffa07a74  804d8f48 nt!KiArgumentTable
```

Looking familiar?

```
lkd> dps 81133230
81133230  811b49f8
81133234  00000000
81133238  0000011c
8113323c  804d8f48 nt!KiArgumentTable
```

Let's follow the first rogue ServiceTable pointer. I consider these rogue pointers because they do not have an associated symbol.

```
lkd> dds 81267918 L 11c
81267918  80586691 nt!NtAcceptConnectPort
8126791c  805706ef nt!NtAccessCheck
81267920  80579b71 nt!NtAccessCheckAndAuditAlarm
…
81267a1c  8123e487 nt!NtDeleteValueKey(hooked)
81267a34  8123e16b nt!NtEnumerateKey(hooked)
81267a3c  8123e267 nt!NtEnumerateValueKey(hooked)
81267af4  8123e0c3 nt!NtOpenKey(hooked)
81267b00  8123de93 nt!NtOpenProcess(hooked)
81267b18  8123df0b nt!NtOpenThread(hooked)
81267b3c  8123e617 nt!NtProtectVirtualMemory(hooked)
81267bcc  8123dda0 nt!NtQuerySystemInformation(hooked)
81267c00  8123e56b nt!NtReadVirtualMemory(hooked)
81267c6c  8123e070 nt!NtSetContextThread(hooked)
81267cf4  8123e397 nt!NtSetValueKey(hooked)
81267d10  8123e01d nt!NtSuspendThread(hooked)
81267d20  8123dfca nt!NtTerminateThread(hooked)
81267d6c  8123e5c1 nt!NtWriteVirtualMemory(hooked)
…
```

This rogue SSDT seems to have correct pointers to most of the APIs, however I took the liberty to point out the few that didn't.

The second rogue SSDT turned out to be a mirror copy of the first.

Since new SSDTs were created, another mechanism must be responsible for updating the KTHREAD ServiceTable pointers. A possible way to accomplish this would be to patch PsSetCreateThreadNotifyRoutine or PsSetCreateProcessNotifyRoutine in order to intercept thread creation callbacks.

Conclusion-

While this technique is neither new nor ground-breaking, it is used in rootkits today. This article was intended to show the technique used from a kernel debugging point-of-view.

References-

http://www.microsoft.com/whdc/devtools/debugging/default.mspx
http://www.secureworks.com/research/threats/blackenergy2/
http://www.rootkit.com/newsread.php?newsid=922
http://www.kernelmode.info/forum/viewtopic.php?f=16&t=42

Please send questions and comments to nick.nopsled[at]gmail.com