(Anti-)Anti-Rootkit Techniques - Part II: Stomped Drivers and Hidden Threads

eversinc33.com/posts/anti-anti-rootkit-part-ii.html

```
_
                                                           o88
ooooooo     ooooooo
 oooooooo8  oooo   oooo  oooooooo8  oo  oooooo    oooooooo8  oooo  oo  oooooo    ooooooo
o88    888o o88    888o
888oooooo8   888   888  888oooooo8   888    888  888ooooooo   888   888   888  888
888   88888o     88888o
888           888  888  888          888                 888  888   888   888  888
88o    o888 88o    o888
  88oooo888     888     88oooo888  o888o      88oooooo88  o888o o888o o888o  88ooo888
88ooo88     88ooo88
```

09/19/2024

(Anti-)Anti-Rootkit Techniques II: Stomped Drivers & Hidden Threads

At the end of Part I of this Series, we ended up with a small anti-rootkit driver, that was able to detect malicious drivers mapped to unbacked memory if they either run as a standard Windows Driver (that registers a device object for `IRP` communication) or run any thread in unbacked memory at all - unless they employ some other anti-anti-rootkit techniques.

This post will cover some evasions against this specific anti-rootkit and as such build upon Part I - if you have not read it, you might want to do it now. It is a rather short read anyway. Also check out my rootkit Banshee and the anti-rootkit unKover. This post is mainly an aggregation of known anti-rootkit/anti-cheat evasion techniques and me coming up with ways to detect them.

Detection 1: Detecting driver "stomping"

The last part was mainly about detecting rootkits that are mapped to memory, using a mapper such as kdmapper. Generally, these tools map a driver manually to kernel memory, using an arbitrary write primitive in a vulnerable, signed driver - so the premise of the last blog post was that detecting threads originating from unbacked memory is one way to detect these types of rootkits.

I ended the previous post with a short word on driver "stomping", i.e. loading the rootkit over an existing driver in memory. As I mentioned, this can easily be detected by simply comparing a driver's `.text` section on disk to its `.text` section in memory (analogous to detecting module stomping).

The implementation is really straightforward (as usual, error handling ommited for brevity):

First, we iterate over the `\Driver` directory, as known from Part I:

```
// Get Handle to \Driver directory
InitializeObjectAttributes(&attributes, &directoryName, OBJ_CASE_INSENSITIVE, NULL,
NULL);
status = ZwOpenDirectoryObject(&handle, DIRECTORY_ALL_ACCESS, &attributes);
status = ObReferenceObjectByHandle(handle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
&directory, nullptr);

POBJECT_DIRECTORY directoryObject = (POBJECT_DIRECTORY)directory;
ULONG_PTR hashBucketLock = directoryObject->Lock;

DbgPrint("Scanning DriverObjects...\n");

// Lock the hashbucket
KeEnterCriticalRegion();
ExAcquirePushLockExclusiveEx(&hashBucketLock, 0);

for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets)
{
    while (entry != nullptr && entry->Object)
    {
        PDRIVER_OBJECT driver = (PDRIVER_OBJECT)entry->Object;
```

Then, we get the driver service name and look up its path in the registry. (This is flawed, as a rootkit can spoof this as well, by setting this value to point to the actual rootkit driver - but then, the attacker has to drop it to disk (or hook the filesystem driver and spoof it, but that requires some additional effort)):

```
        // Get driver service name to lookup path to binary. Strip \Driver prefix
        UkStripDriverPrefix(&driver->DriverName, &driverServiceName);

        // This queries the registry for the image path
        NTSTATUS status = UkGetDriverImagePath(&driverServiceName, &imagePath);

        // Create an absolute path
        status = UkPrependWindowsPathIfStartsWithSystem32(&imagePath,
&imagePathAbsolute);
```

With this absolute path, we can now compare it to the image in memory:

```
        // read the image and compare it to the in memory image
        ULONG fileSize = 0;
        status = UkReadFileToMemory(&imagePathAbsolute, &fileBuffer, &fileSize);

        // compare .text sections
        if (!NT_SUCCESS(UkGetPeSection(".text", fileBuffer, textSectionOnDiskBuffer,
&sectionSizeOnDisk))
            || !NT_SUCCESS(UkGetPeSection(".text", driver->DriverStart,
textSectionInMemBuffer, &sectionSizeInMem))
            || !textSectionOnDiskBuffer || !textSectionInMemBuffer)
        {
            goto Next;
        }

        if (RtlCompareMemory(textSectionOnDiskBuffer, textSectionInMemBuffer,
sectionSizeOnDisk) != sectionSizeOnDisk)
        {
            DbgPrint("-- [!] .TEXT SECTION DIFFERS\n");
        }

    Next:
        /* [...] Cleanup */
        entry = entry->ChainLink;
    }
}

ExReleasePushLockExclusiveEx(&hashBucketLock, 0);
KeLeaveCriticalRegion();
ObDereferenceObject(directory);
ZwClose(handle);
```

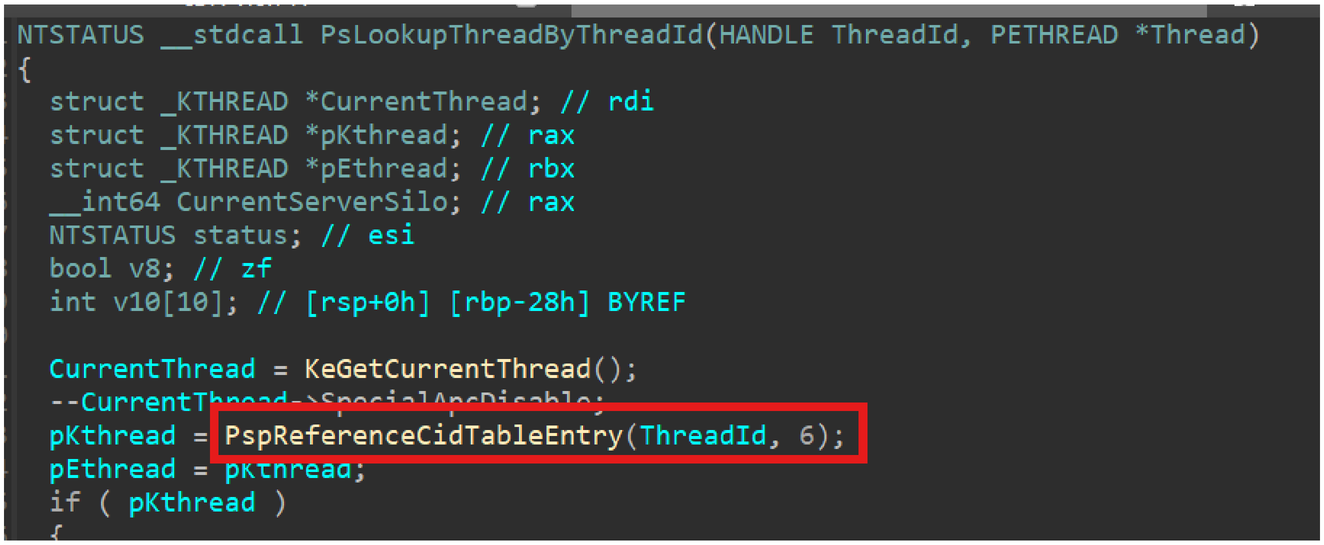
I implemented this in unKover and it seems to work quite well - I did not find any self-modifying driver's, i.e. false positives, on my machine at all (However, for the Ghost Drivers, some path resolving adjustments would need to be done).

But what if the mapper does not use the `.text` section, but rather some other section such as `.data` or `.rdata`? This is implemented in tools such as SinMapper or lpmapper - for the latter, VollRagm describes detection vectors in his blog post Abusing LargePageDrivers to copy shellcode into valid kernel modules himself. For `SinMapper`, we should be able to detect threads with the usual methods from Part I, except that this time we should check not only for unbacked memory, but also for threads originating from non-`.text` sections. One more for the unKover backlog...

Anyway, driver "stomping" seems to not be the silver bullet to get rid of thread detections, so what do we do to evade detection? We attack unKover's flawed implementation of emumerating threads and their start addresses.

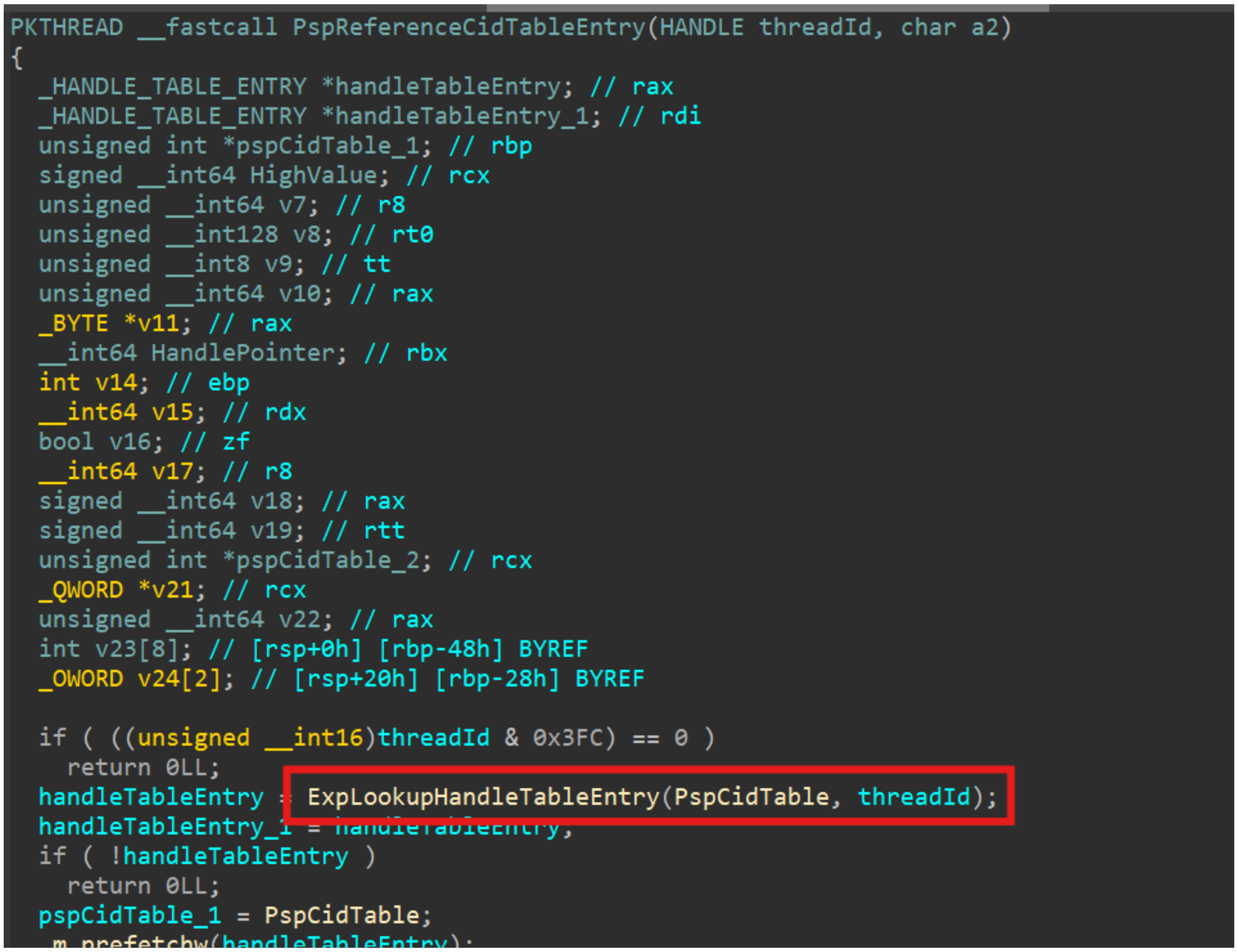
Windows Thread Internals: Handle Tables and the PspCidTable

In each implemented technique that checks thread's start addresses, unKover uses `PsLookupThreadByThreadId`, a routine exported by `ntoskrnl.exe`. If we look at its implementation by decompiling the function in IDA, we see that it internally calls the private `PspReferenceCidTableEntry` routine:



```
NTSTATUS __stdcall PsLookupThreadByThreadId(HANDLE ThreadId, PETHREAD *Thread)
{
  struct _KTHREAD *CurrentThread; // rdi
  struct _KTHREAD *pKthread; // rax
  struct _KTHREAD *pEthread; // rbx
  __int64 CurrentServerSilo; // rax
  NTSTATUS status; // esi
  bool v8; // zf
  int v10[10]; // [rsp+0h] [rbp-28h] BYREF

  CurrentThread = KeGetCurrentThread();
  --CurrentThread->SpecialApcDisable;
  pKthread = PspReferenceCidTableEntry(ThreadId, 6);
  pEthread = pKthread;
  if ( pKthread )
```

If we take a look at that function, we can see a reference to a global symbol named `PspCidTable`:



```
PKTHREAD __fastcall PspReferenceCidTableEntry(HANDLE threadId, char a2)
{
  _HANDLE_TABLE_ENTRY *handleTableEntry; // rax
  _HANDLE_TABLE_ENTRY *handleTableEntry_1; // rdi
  unsigned int *pspCidTable_1; // rbp
  signed __int64 HighValue; // rcx
  unsigned __int64 v7; // r8
  unsigned __int128 v8; // rt0
  unsigned __int8 v9; // tt
  unsigned __int64 v10; // rax
  _BYTE *v11; // rax
  __int64 HandlePointer; // rbx
  int v14; // ebp
  __int64 v15; // rdx
  bool v16; // zf
  __int64 v17; // r8
  signed __int64 v18; // rax
  signed __int64 v19; // rtt
  unsigned int *pspCidTable_2; // rcx
  _QWORD *v21; // rcx
  unsigned __int64 v22; // rax
  int v23[8]; // [rsp+0h] [rbp-48h] BYREF
  _OWORD v24[2]; // [rsp+20h] [rbp-28h] BYREF

  if ( ((unsigned __int16)threadId & 0x3FC) == 0 )
    return 0LL;
  handleTableEntry = ExpLookupHandleTableEntry(PspCidTable, threadId);
  handleTableEntry_1 = handleTableEntry;
  if ( !handleTableEntry )
    return 0LL;
  pspCidTable_1 = PspCidTable;
```

This is a pointer to a handle table, which (among others) contains handles to threads (as can be seen from calling `ExpLookupHandleTableEntry` with the target thread ID). A handle table is simply a page sized block that stores up to 256 handle entries or references to other handle tables (see [(3) What are Windows Handles - Windows Internals Explained (guidedhacking.com)]()). Below are the relevant C structs for handle tables:

```
typedef struct _HANDLE_TABLE
{
     ULONG TableCode;
     PEPROCESS QuotaProcess;
     PVOID UniqueProcessId;
     EX_PUSH_LOCK HandleLock;
     LIST_ENTRY HandleTableList;
     EX_PUSH_LOCK HandleContentionEvent;
     PHANDLE_TRACE_DEBUG_INFODebugInfo;
     LONG ExtraInfoPages;
     ULONG Flags;
     ULONG StrictFIFO: 1;
     LONG FirstFreeHandle;
     PHANDLE_TABLE_ENTRY LastFreeHandleEntry;
     LONG HandleCount;
     ULONG NextHandleNeedingPool;
} HANDLE_TABLE, *PHANDLE_TABLE;

typedef struct _HANDLE_TABLE_ENTRY
{
     union
     {
          PVOID Object;
          ULONG ObAttributes;
          PHANDLE_TABLE_ENTRY_INFO InfoTable;
          ULONG Value;
     };
     union
     {
          ULONG GrantedAccess;
          struct
          {
               WORD GrantedAccessIndex;
               WORD CreatorBackTraceIndex;
          };
          LONG NextFreeTableEntry;
     };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

If we dig a bit deeper, we actually find out that this specific handle table is also used by `PsLookupProcessByProcessId` - the `PspCidTable` is thus the pool which is used for generating unique Process and Thread (Client) IDs (CIDs). This also explains why two process and thread IDs can never be the same, because this ID pool is shared for both process as well as thread handles.

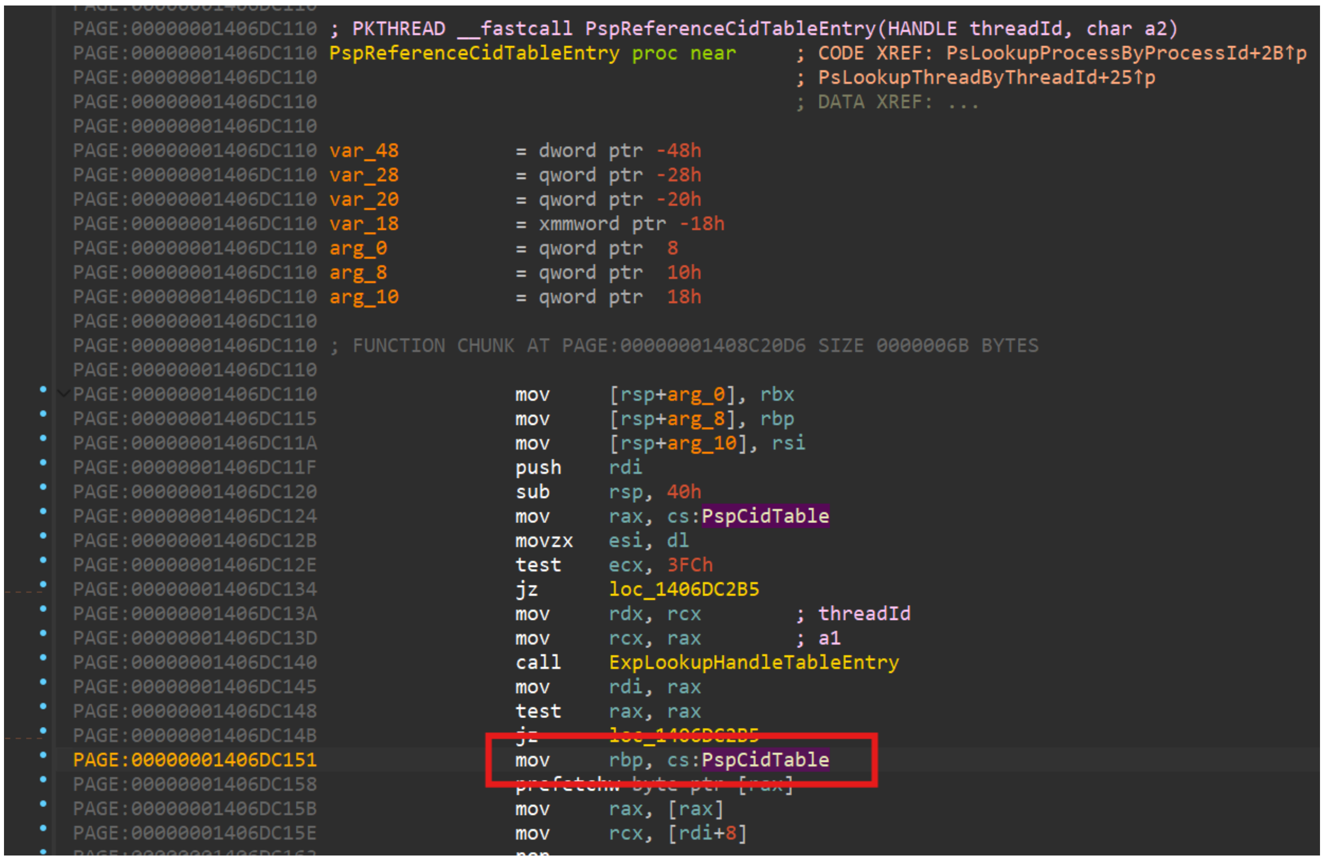
The flaw

As we saw in IDA, two lines below the call to `ExpLookupHandleTableEntry`, if no handle table entry is found for the ID that was queried, the functions returns `NULL` - which is our way to attack the unKover anti-rootkit. If we remove our rootkits thread IDs from this table, any security solution which relies on calls that leverage the `PspCidTable`, e.g. `PsLookupThreadByThreadId`, will not find it, as the function will return `NULL` instead of the actual thread.

With this, we are directly attacking our specific anti-rootkit implementation. The takeaway here is to find a flaw or an oversight in whatever security product you are facing, either through reverse engineering or code auditing, and abuse it.

Locating the PspCidTable

As I described in my blog post Keylogging in the Windows Kernel with undocumented data structures, when locating `gafAsyncKeyState`, we can usually just use signature scanning to find these kind of pointers. Our `PspReferenceCidTableEntry` function seems to be a good fit, as it contains a static reference to our object of interest - which means we can simply scan for the first `mov rbp, cs:` instruction and extract the displacement to the `PspCidTable` from the assembly instruction bytes. This signature might of course be different for different versions of `ntoskrnl` and you might want to either hardcode different target signatures or use something like YASS.

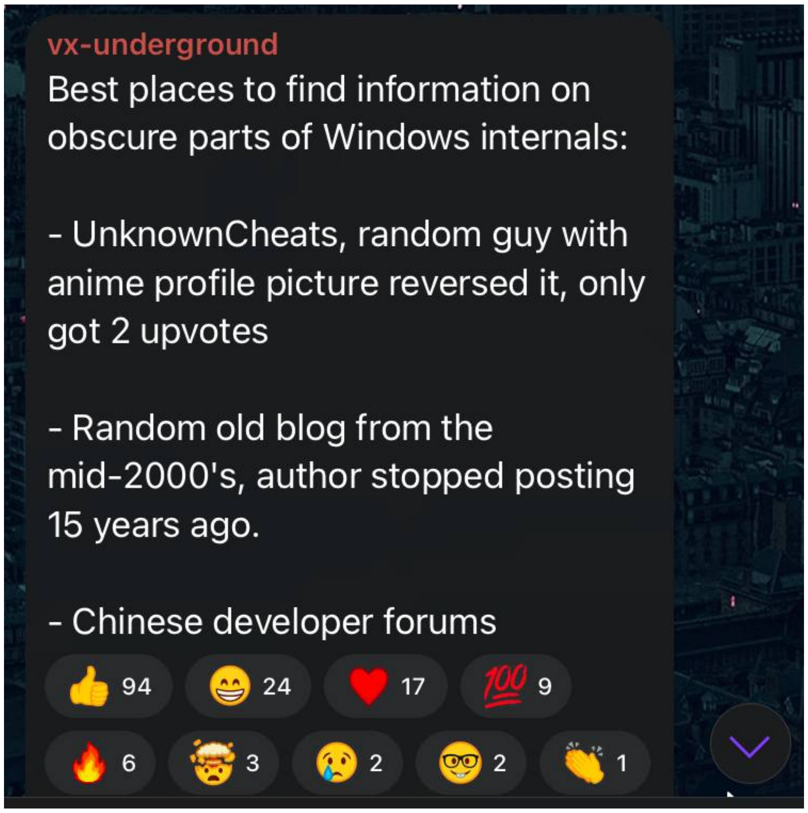


```
PAGE:00000001406DC110 ; PKTHREAD __fastcall PspReferenceCidTableEntry(HANDLE threadId, char a2)
PAGE:00000001406DC110 PspReferenceCidTableEntry proc near     ; CODE XREF: PsLookupProcessByProcessId+2B↑p
PAGE:00000001406DC110                                         ; PsLookupThreadByThreadId+25↑p
PAGE:00000001406DC110                                         ; DATA XREF: ...
PAGE:00000001406DC110
PAGE:00000001406DC110 var_48          = dword ptr -48h
PAGE:00000001406DC110 var_28          = qword ptr -28h
PAGE:00000001406DC110 var_20          = qword ptr -20h
PAGE:00000001406DC110 var_18          = xmmword ptr -18h
PAGE:00000001406DC110 arg_0           = qword ptr  8
PAGE:00000001406DC110 arg_8           = qword ptr  10h
PAGE:00000001406DC110 arg_10          = qword ptr  18h
PAGE:00000001406DC110
PAGE:00000001406DC110 ; FUNCTION CHUNK AT PAGE:00000001408C20D6 SIZE 0000006B BYTES
PAGE:00000001406DC110
PAGE:00000001406DC110                 mov     [rsp+arg_0], rbx
PAGE:00000001406DC115                 mov     [rsp+arg_8], rbp
PAGE:00000001406DC11A                 mov     [rsp+arg_10], rsi
PAGE:00000001406DC11F                 push    rdi
PAGE:00000001406DC120                 sub     rsp, 40h
PAGE:00000001406DC124                 mov     rax, cs:PspCidTable
PAGE:00000001406DC12B                 movzx   esi, dl
PAGE:00000001406DC12E                 test    ecx, 3FCh
PAGE:00000001406DC134                 jz      loc_1406DC2B5
PAGE:00000001406DC13A                 mov     rdx, rcx        ; threadId
PAGE:00000001406DC13D                 mov     rcx, rax        ; a1
PAGE:00000001406DC140                 call    ExpLookupHandleTableEntry
PAGE:00000001406DC145                 mov     rdi, rax
PAGE:00000001406DC148                 test    rax, rax
PAGE:00000001406DC14B                 jz      loc_1406DC2B5
PAGE:00000001406DC151                 mov     rbp, cs:PspCidTable
PAGE:00000001406DC158                 prefetchw byte ptr [rax]
PAGE:00000001406DC15B                 mov     rax, [rax]
PAGE:00000001406DC15E                 mov     rcx, [rdi+8]
```

(If you are wondering about the `cs` segment selector - `cs` base is usually set to 0 in 64 bit operating systems that do not rely on segmentation anymore, so you can directly extract the displacement/offset from this instruction and treat it like a "regular" `mov` instruction).

New dog, old tricks

How do we remove our handle from this table? According to vx-underground, there are three sources we can consult:



vx-underground
Best places to find information on obscure parts of Windows internals:

- UnknownCheats, random guy with anime profile picture reversed it, only got 2 upvotes

- Random old blog from the mid-2000's, author stopped posting 15 years ago.

- Chinese developer forums

我不会说中文, so here are the first two:

- In this UnknownCheats post, `ExDestroyHandle` is used to simply destroy/remove the handle from the table (see [Tutorial] Remove your systemthread from PspCidTable)
- In this blogpost from 2006 in Uninformed the `Object` property of the `HANDLE_TABLE_ENTRY` struct is set to `NULL` instead (funnily enough to evade the Blacklight anti-rootkit back then)

Let's implement the first approach as described in the tutorial.

Removing the Handle Table Entry

First, we scan for `ExMapHandleToPointer`, from which we scan for `ExpLookupHandleTableEntry`, which is our function that performs a lookup on the `PspCidTable` and gets our thread's `HANDLE_TABLE_ENTRY` for us. Simply using `ExMapHandleToPointer` is not possible, because this will cause a deadlock. We also need to scan for `ExDestroyHandle`. I am not going to explain this step by step again here, look for `xrefs` in IDA, extract a signature and scan.

We can remove our handle from the table, to hide it from the OS, with one very simple function:

```
NTSTATUS
RemoveEntryFromPspCidTable(
    ULONG id
)
{
    auto cidEntry = g_ExpLookupHandleTableEntry(*g_pPspCidTable, ULongToHandle(id));
    if (cidEntry != NULL)
    {
        g_ExDestroyHandle(*g_pspCidTable, ULongToHandle(id), cidEntry);
        return STATUS_SUCCESS;
    }
}
```

With the handle gone from the `PspCidTable`, any function that relies on it, such as most of the `Ps*` routines from `ntoskrnl` will not find our thread anymore.

Of course though, this is not a perfect cloak hiding us from vigilant anti-rootkit eyes...

Detection 2: Detecting tampering through finding inconsistencies

A general strategy to detecting tampering is to check for inconsistencies - usually, attackers that tamper with data do only as much as they need to and might overlook someplace else, where that data is still referenced. What I am saying is, there are other methods to list threads from a windows driver than simply using the provided `Ps*` API. If the thread shows up in one place, but not in the other, we found our offender.

One method, which I implemented in unKover, is walking a processes thread linked list.

Each process is represented in the Windows Kernel as an `KPROCESS`/`EPROCESS` object. This object contains a linked list of threads, containing all threads for that process, starting with the `ThreadListHead`.

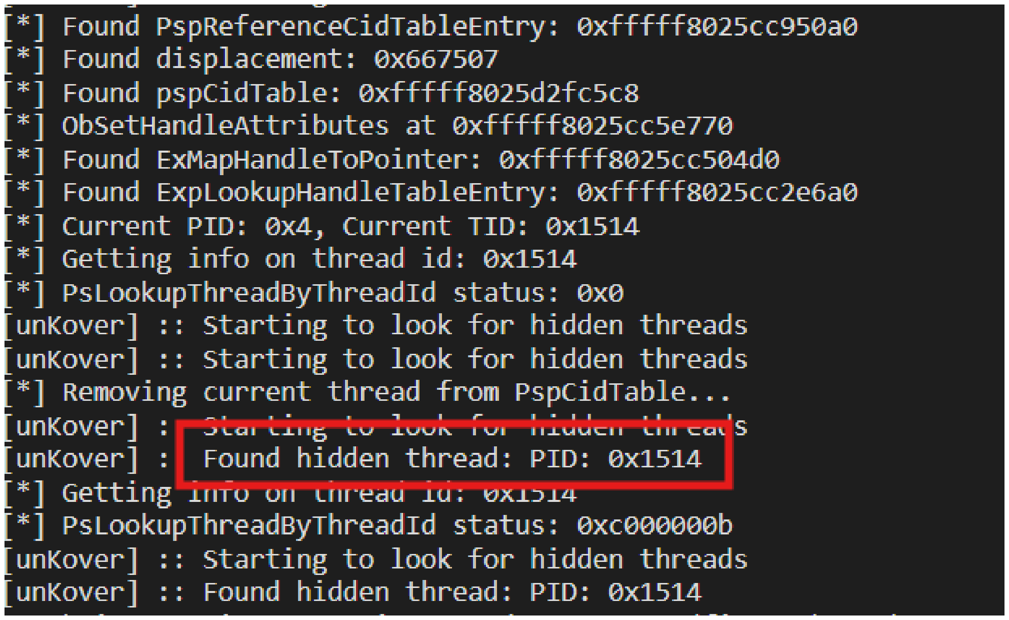
(The `E*` structs contain the corresponding `K*` struct as the first member, which means you can typecast between them as you like. `K*` is essentially a subset of `E*`)

```
0: kd> dt nt!_EPROCESS
   +0x000 Pcb                : _KPROCESS /* KPROCESS as first member of EPROCESS */
   +0x438 ProcessLock        : _EX_PUSH_LOCK
   +0x440 UniqueProcessId    : Ptr64 Void
   +0x448 ActiveProcessLinks : _LIST_ENTRY
   [ ... ]
   +0x5e0 ThreadListHead     : _LIST_ENTRY /* The thread linked list head */
```

Also, this can linked list can be referenced from a thread that is a member of that linked list, either from `KTHREAD` or `ETHREAD`:

```
1: kd> dt nt!_KTHREAD
   +0x000 Header            : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget     : Uint8B
   [ ... ]
   +0x2f8 ThreadListEntry   : _LIST_ENTRY /* The list entry reference */
```

Since all kernel drivers, and as such rootkits, run under the windows system process with the process ID `4` by default, if we get our current thread from our anti-rootkit driver, we are in the right linked list. We can then walk that list from the `ThreadListEntry` of our thread onwards, to enumerate all driver threads running under the system process. If we find a thread ID here that can not be found in the `PspCidTable`, e.g. via `PsLookupThreadByThreadId`, or points to a corrupted entry, we found our offender:

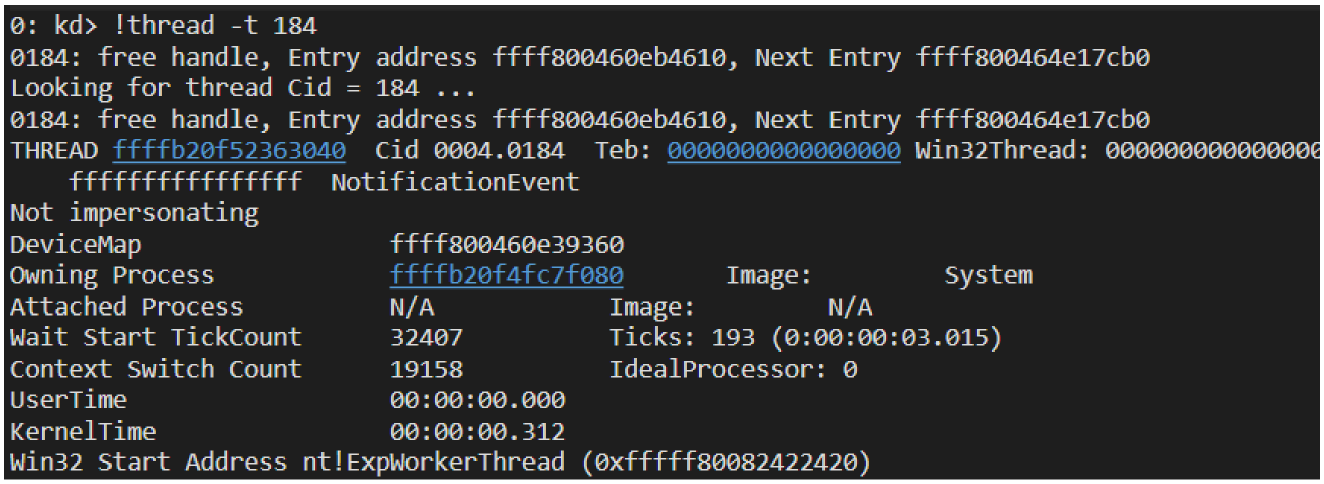


```
[*] Found PspReferenceCidTableEntry: 0xfffff8025cc950a0
[*] Found displacement: 0x667507
[*] Found pspCidTable: 0xfffff8025d2fc5c8
[*] ObSetHandleAttributes at 0xfffff8025cc5e770
[*] Found ExMapHandleToPointer: 0xfffff8025cc504d0
[*] Found ExpLookupHandleTableEntry: 0xfffff8025cc2e6a0
[*] Current PID: 0x4, Current TID: 0x1514
[*] Getting info on thread id: 0x1514
[*] PsLookupThreadByThreadId status: 0x0
[unKover] :: Starting to look for hidden threads
[unKover] :: Starting to look for hidden threads
[*] Removing current thread from PspCidTable...
[unKover] :: Starting to look for hidden threads
[unKover] :: Found hidden thread: PID: 0x1514
[*] Getting info on thread id: 0x1514
[*] PsLookupThreadByThreadId status: 0xc000000b
[unKover] :: Starting to look for hidden threads
[unKover] :: Found hidden thread: PID: 0x1514
```

The code is just as straightforward as removing the handle is. Unfortunately though, the offset for the `ThreadListEntry` from `KTHREAD` is hardcoded for now. At least in the two Windows versions I have running as VMs, this offset is stable ¯_(ツ)_/¯

```
#define THREAD_LIST_ENTRY_OFFSET 0x2f8
typedef struct _myKTHREAD
{
    char padding[0x2F8];                // 0x0000
    struct _LIST_ENTRY ThreadListEntry; // 0x02F8
    // [ ... ]
} myKTHREAD, * myPKTHREAD;

NTSTATUS
UkWalkSystemProcessThreads()
{
    // Get current thread (an arbitrary thread in system process / PID 4 is ok)
    auto currentThread = KeGetCurrentThread();
    auto threadListEntry = (PLIST_ENTRY)((ULONG_PTR)currentThread +
THREAD_LIST_ENTRY_OFFSET);
    auto listEntry = threadListEntry;

    // walk all linked list entries
    while ((listEntry = listEntry->Flink) != threadListEntry)
    {
        auto entry = CONTAINING_RECORD(listEntry, myKTHREAD, ThreadListEntry);
        auto threadId = (ULONG)PsGetThreadId((PETHREAD)entry);

        if (threadId != 0)
        {
            PETHREAD pThread = NULL;
            NTSTATUS status = PsLookupThreadByThreadId(ULongToHandle(threadId),
&pThread);

            // If PsLookupThreadByThreadID fails, we found our offender
            if (!NT_SUCCESS(status))
            {
                LOG_MSG("Found hidden thread: PID: 0x%llx\n", threadId);
            }
        }
    }

    return STATUS_SUCCESS
```

I believe that this is what WinDbg's `!thread` command also does - if it does not find the thread in the `PspCidTable`, it walks the list to find it (notice the "free handle"). That is just a guess however.



```
0: kd> !thread -t 184
0184: free handle, Entry address ffff800460eb4610, Next Entry ffff800464e17cb0
Looking for thread Cid = 184 ...
0184: free handle, Entry address ffff800460eb4610, Next Entry ffff800464e17cb0
THREAD ffffb20f52363040  Cid 0004.0184  Teb: 0000000000000000 Win32Thread: 000000000000000
    ffffffffffffffff  NotificationEvent
Not impersonating
DeviceMap                 ffff800460e39360
Owning Process            ffffb20f4fc7f080       Image:         System
Attached Process          N/A            Image:         N/A
Wait Start TickCount      32407          Ticks: 193 (0:00:00:03.015)
Context Switch Count      19158          IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.312
Win32 Start Address nt!ExpWorkerThread (0xfffff80082422420)
```

While we could do this cat and mouse dance for some more time and start removing our thread elsewhere, as well as removing our process or even spoofing threads, this is an endless back and forth that I am not going to exercise. If you are aware of the process hiding trick from rootkits, which unlinks the process from the process linked list - you can essentially do the same here with the thread linked list. For a good explanation, see the readme of ZeroMemoryEx's Chaos-Rootkit.

Outlook

In the realm of userland malware, when thread callbacks were starting to detect many different code injection techniques, people figured that going threadless was the way to go. Since unKover is so heavily thread based, we will also go that route and implement a threadless rootkit in the conclusion of this series, part III, which will hopefully not take me half a year to write. That being said, my life right now is very busy, which is why part II took me so long ... anyway

Happy Hacking!

`---`

back to top

`helloskiddie.club` <3