“Move aside, signature scanning!” Better kernel data discovery through lookaside lists

windows-internals.com/lookaside-list-forensics

By Yarden Shafir & Alex Ionescu

Introduction

A while ago we did some research. That specific project might be published at some other time in the future and we won’t go into too much detail about it here. But as part of this project we wanted to gain access into an internal data structure used by some driver. Sadly, the driver’s global pointer to this data structure is not exported, and we couldn’t find a way to access it from outside the driver itself. It is stored in the pool, so we couldn’t even scan the driver address space for signs of this structure.

Of course, there is always the option of doing binary parsing on the driver based on a function signature that references the global, and/or using an array of known offsets for the global variable and adding the driver base to find it. But these methods require finding and using the correct RVA for every version of the driver, as well as all potential function signatures. Because this driver does not have exported functions, such signatures would be brittle and subject to change between releases. Therefore, although often used by malware authors, we find these techniques ugly and inconvenient to implement — we knew we could do better.

So, we reverse engineered the data structure itself and came up with an interesting idea that can give us easy access to this data structure and to many others. The data structure we were interested in is very large and contains, among other things, a few lookaside lists embedded in it. Lookaside lists are single linked lists containing pool allocations of a fixed size. They are used by drivers for caching memory allocations instead of always requesting them from the memory manager. Let’s see what makes these interesting.

System Lookaside Lists

Here is the `wdm.h` definition of a `GENERAL_LOOKASIDE_LAYOUT` (`GENERAL_LOOKASIDE` is just an aligned version of `GENERAL_LOOKASIDE_LAYOUT`):

```
//
// The goal here is to end up with two structure types that are identical
except
// for the fact that one (GENERAL_LOOKASIDE) is cache aligned, and the
other
// (GENERAL_LOOKASIDE_POOL) is merely naturally aligned.
```

```
//
// An anonymous structure element would do the trick except that C++ can't
handle
// such complex syntax, so we're stuck with this macro technique.
//
#define GENERAL_LOOKASIDE_LAYOUT                \
    union {                                     \
        SLIST_HEADER ListHead;                  \
        SINGLE_LIST_ENTRY SingleListHead;       \
    } DUMMYUNIONNAME;                           \
    USHORT Depth;                               \
    USHORT MaximumDepth;                        \
    ULONG TotalAllocates;                       \
    union {                                     \
        ULONG AllocateMisses;                   \
        ULONG AllocateHits;                     \
    } DUMMYUNIONNAME2;                          \
                                                \
    ULONG TotalFrees;                           \
    union {                                     \
        ULONG FreeMisses;                       \
        ULONG FreeHits;                         \
    } DUMMYUNIONNAME3;                          \
                                                \
    POOL_TYPE Type;                             \
    ULONG Tag;                                  \
    ULONG Size;                                 \
    union {                                     \
        PALLOCATE_FUNCTION_EX AllocateEx;       \
        PALLOCATE_FUNCTION Allocate;            \
    } DUMMYUNIONNAME4;                          \
                                                \
    union {                                     \
        PFREE_FUNCTION_EX FreeEx;               \
        PFREE_FUNCTION Free;                    \
    } DUMMYUNIONNAME5;                          \
                                                \
    LIST_ENTRY ListEntry;                       \
    ULONG LastTotalAllocates;                   \
    union {                                     \
        ULONG LastAllocateMisses;               \
        ULONG LastAllocateHits;                 \
    } DUMMYUNIONNAME6;                          \
   ULONG Future[2];
```

A useful fact to notice is that this structure contains a linked list (`GENERAL_LOOKASIDE.ListEntry`), meaaning all lookaside lists do. Depending on whether the lookaside list was created with `ExInitializeNPagedLookasideList` or `ExInitializePagedLookasideList` (or, if `ExInitializeLookasideListEx` was used, the `PoolType` which was passed in), the data structure will be entered into one of two list heads. As such, if we follow the `ListEntry` of any lookaside list, we'll eventually end up at either `ExPagedLookasideListHead` or `ExNPagedLookasideListHead`. Since we create our own lookaside list through these APIs, if we pick the same pool type as our target structure, we can therefore through all other lookasides, and eventually reach the one contained in our target structure. In this particular use case, using our own definition of the structure, the useful `CONTAINING_RECORD` macro, and the knowledge that the first member of the structure is a "magic" `ULONG` that always contains the same value, we searched all lookaside lists using this mechanism until we reached our structure.

But the possibilities don't stop there – this method gives us access to *any* kernel structure, exported or not, that contains a lookaside list. So what else is there?

Pool-Based Lookaside Lists

With some WinDbg magic, we can also find out valuable information about the data – whether it's inside a driver (and which one!) or in the kernel pool, who it belongs to, the allocation size, etc. To explore the possibilities, we wrote a simple WinDbg script that iterates through all lookaside lists and uses the extremely helpful `!pool` extension to dump information about them. Although we could build similar functionality in a custom C driver, there is no Windows Kernel API that can supply us with similar information about pool allocations and parsing pool pages to retrieve it is a lot of work, so we decided to avoid implementing the same functionality in C due to laziness. In fact, while we tried to implement our own C-based pool parser, we ended up realizing that nobody had described the myriad of changes in Windows `10 RS5` and above's pool manager, so we're busy writing a book on the topic.

Using our script, we found structures containing lookaside lists that belong to `FltMgr.sys`, `Win32k.sys`, Windows Defender drivers, various display drivers, and much more.

```
dx -r0 @$GeneralLookaside = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExPagedLookasideListHead, "nt!_GENERAL_LOOKASIDE",
"ListEntry")
dx -r0 @$lookasideAddr = @$GeneralLookaside.Select(l =>
((__int64)&l).ToDisplayString("x"))
dx -r0 @$extractBetween = ((x,y,z) => x.Substring(x.IndexOf(y)
+ y.Length, x.IndexOf(z) - x.IndexOf(y) - y.Length))
dx -r0 @$extractWithSize = ((x,y,z) => x.Substring(x.IndexOf(y) + y.Length,
z))
dx -r2 @$poolData = @$lookasideAddr.Select(l
```

```
=> Debugger.Utility.Control.ExecuteCommand("!pool "+l+" 2")).Where(l =>
l[1].Length != 0x55 && l[1].Length != 0).Select(l => new {address = "0x" +
@$extractBetween(l[1], "*", "size:"), tag = @$extractWithSize(l[1], "
(Allocated) *", 4), tagDesc = l[2].Contains(",") ? @$extractBetween(l[2], ":
", ",") : l[2].Substring(l[2].IndexOf(":")+2), binary =
l[2].Contains("Binary") ? l[2].Substring(l[2].IndexOf("Binary :")+9) :
"unknown", size = "0x" + @$extractBetween(l[1], "size:", "previous
size:").Replace(" ", "")})

[0x4a]
    address : 0xffff988679939400
    tag :     Vi10
    tagDesc : Video memory manager process heap
    binary :  dxgmms2.sys
    size :    0x70
[0x4b]
    address : 0xffff98867b647650
    tag :     DxgK
    tagDesc : Vista display driver support
    binary :  dxgkrnl.sys
    size :    0x640
[0x4c]
    address : 0xffff98867b647650
    tag :     DxgK
    tagDesc : Vista display driver support
    binary :  dxgkrnl.sys
    size :    0x640
[0x4d]
    address : 0xffff9886790f5430
    tag :     Vi17
    tagDesc : Video memory manager pool
    binary :  dxgmms2.sys
    size :    0x150
[0x4e]
    address : 0xffff98867966e230
    tag :     Usla
    tagDesc : USERTAG_LOOKASIDE
    binary :  win32k!InitLockRecordLookaside
    size :    0xa0
[0x4f]
    address : 0xffff98867966ea50
    tag :     Usla
    tagDesc : USERTAG_LOOKASIDE
    binary :  win32k!InitLockRecordLookaside
    size :    0xa0
[0x50]
    address : 0xffff98867966e690
```

```
    tag :     Gla1
    tagDesc : GDITAG_HMGR_LOOKASIDE_DC_TYPE
    binary :  win32k.sys
    size :    0xa0
[0x51]
    address : 0xffff98867966e550
    tag :     Gla4
    tagDesc : GDITAG_HMGR_LOOKASIDE_RGN_TYPE
    binary :  win32k.sys
    size :    0xa0
[0x52]
    address : 0xffff98867966ecd0
    tag :     Gla5
    tagDesc : GDITAG_HMGR_LOOKASIDE_SURF_TYPE
    binary :  win32k.sys
    size :    0xa0
```

There are some results in which the pool tag is unknown, making the tracking of the driver they belong to difficult. A fun way to solve that is using driver verifier's pool tracking feature. We can modify our script and replace the `!pool <address> 2` command with `!verifier <address> 2` and receive information about the allocating driver and the completes stack trace of the allocation. But running this command on so many addresses is extremely slow and it dumps a lot of information that is hard to sort through. So another option is going for a more manual approach – enabling driver verifier but executing the previous script as it is, and only querying specific addresses that seem interesting with verifier.

Image-Based Lookaside Lists

Initially we only searched for data in the pool because that is where the structure we were interested in was allocated. But with this trick we also get access to lookaside lists that are inside drivers, and we can use the cool new **`RtlPcToFileName`** function to find out what driver these structures are in. In this case we did choose to implement this in C code since it's more straightforward and faster to execute:

```
_Use_decl_annotations_
NTSTATUS
DriverEntry (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS status;
    LOOKASIDE_LIST_EX lookaside;
    PLIST_ENTRY lookasideList;
    PLIST_ENTRY lookasideListHead;
    PGENERAL_LOOKASIDE generalLookaside;
```

```
    UNICODE_STRING pcName = RTL_CONSTANT_STRING(L"RtlPcToFileName");
    DECLARE_UNICODE_STRING_SIZE(driverName, 32);
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = DriverUnload;

    auto RtlPcToFileNamePtr = (decltype(RtlPcToFileName)*)
(MmGetSystemRoutineAddress(&pcName));
    NT_ASSERT(RtlPcToFileNamePtr != nullptr);

    //
    // Create our own lookaside list to use for finding other lookaside
lists in the kernel.
    //
    status = ExInitializeLookasideListEx(&lookaside,
                                         nullptr,
                                         nullptr,
                                         PagedPool,
                                         0,
                                         8,
                                         'Fake',
                                         0);
    if (!NT_SUCCESS(status))
    {
        goto Exit;

    //
    // Iterate over our lookaside list to find all the other lookaside
lists
    // and print information about them
    //
    generalLookaside = nullptr;
    lookasideListHead = &lookaside.L.ListEntry;
    lookasideList = lookasideListHead->Flink;
    do
    {
        generalLookaside = CONTAINING_RECORD(lookasideList,
                                             GENERAL_LOOKASIDE,
                                             ListEntry);

        //
        // Use RtlPcToFileName to find whether the lookaside list is
        // inside a driver and if so, which one
        //
        status = RtlPcToFileNamePtr(generalLookaside, &driverName);
        if (NT_SUCCESS(status))
```

```
        {
            DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                       DPFLTR_ERROR_LEVEL,
                       "Lookaside list is in driver %wZ\n",
                       driverName);
        }
        else
        {
            DbgPrintEx(DPFLTR_IHVDRIVER_ID,
```
DPFLTR_ERROR_LEVEL,
"Lookaside list is not inside a driver\n");
```
        lookasideList = lookasideList->Flink;
    } while (lookasideList != lookasideListHead);

    status = STATUS_SUCCESS;

Exit:
    ExDeleteLookasideListEx(&lookaside);
    return status;
}
```

With this code we found lookaside lists inside of `Ntoskrnl.exe`, `Ci.dll`, `Ntfs.sys` and more. Of course, since these are embedded inside of the driver memory, our only way to know whether these are independent lookaside lists or they are part of a larger structure is to dump the addresses and reverse engineer the drivers. But we're all nerds who like reverse engineering, or we wouldn't be writing/reading this blog.

```
15   0.01140480   Lookaside list is in driver ntoskrnl.exe
16   0.01244490   Lookaside list is in driver ntoskrnl.exe
17   0.01345240   Lookaside list is in driver FLTMGR.SYS
18   0.01443390   Lookaside list is in driver FLTMGR.SYS
19   0.01562170   Lookaside list is in driver FLTMGR.SYS
20   0.01645300   Lookaside list is in driver FLTMGR.SYS
21   0.01738160   Lookaside list is in driver FLTMGR.SYS
22   0.01841000   Lookaside list is in driver FLTMGR.SYS
23   0.01938590   Lookaside list is in driver fileinfo.sys
24   0.01996580   Lookaside list is in driver Wof.sys
25   0.02044390   Lookaside list is in driver Wof.sys
```

We can also implement the same query in WinDbg if we choose to, using the `ln` command which searches for the nearest symbol to an address:

```
dx -r0 @$GeneralLookaside = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExPagedLookasideListHead, "nt!_GENERAL_LOOKASIDE",
"ListEntry")
```

```
dx -r0 @$lookasideAddr = @$GeneralLookaside.Select(l =>
((__int64)&l).ToDisplayString("x"))
dx -r2 @$symData = @$lookasideAddr.Select(l => new {addr =
l, sym = Debugger.Utility.Control.ExecuteCommand("ln "+l)}).Where(l
=> l.sym.Count() > 3).Select(l => new {addr = l.addr, sym =
@$extractBetween(l.sym[3], "   ", "|")})
```

```
[0x9]
    addr        : 0xfffff8000e4eb300
    sym         : nt!AlpcpLookasides+0x100
[0xa]
    addr        : 0xfffff8000e4db180
    sym         : nt!IopSymlinkInfoLookasideList
[0xb]
    addr        : 0xfffff8000e4ef040
    sym         : nt!WmipDSChunkInfoLookaside
[0xc]
    addr        : 0xfffff8000e4eefc0
    sym         : nt!WmipGEChunkInfoLookaside
[0xd]
    addr        : 0xfffff8000e4ef140
    sym         : nt!WmipISChunkInfoLookaside
[0xe]
    addr        : 0xfffff8000e4ef0c0
    sym         : nt!WmipMRChunkInfoLookaside
[0xf]
    addr        : 0xfffff8001172a880
    sym         : FLTMGR!FltGlobals+0x340
[0x10]
    addr        : 0xfffff8001172ad00
    sym         : FLTMGR!FltGlobals+0x7c0
[0x11]
    addr        : 0xfffff8001172af00
    sym         : FLTMGR!FltGlobals+0x9c0
[0x12]
    addr        : 0xfffff8001172b080
    sym         : FLTMGR!FltGlobals+0xb40
```

This is a pretty cool trick, which led to all sorts of cool discoveries. And we only searched for paged lookaside lists. There is a whole world of non-paged lookaside lists that we didn't even look at yet. We ran the same WinDbg scripts as before, and just changed our starting point from `nt!ExPagedLookasideListHead` to `nt!ExNPagedLookasideListHead` to get the non-paged lookaside lists, and got some interesting results. We looked for non-paged lookaside lists in the pool:

```
[0x55]
    address : 0xffff97884ba5c990
    tag :     Vkin
    tagDesc : Hyper-V VMBus KMCL driver (incoming packets)
    binary :  vmbkmcl.sys
    size :    0x2d0
[0x56]
    address : 0xffff97884bad1590
    tag :     NDnd
    tagDesc : NDIS_TAG_POOL_NDIS
    binary :  ndis.sys
    size :    0x800
[0x57]
    address : 0xffff97884bad3000
    tag :     NDrt
    tagDesc : NDIS_TAG_RST_NBL
    binary :  ndis.sys
    size :    0x800
[0x58]
    address : 0xffff97884ba19130
    tag :     Nnbf
    tagDesc : NetIO NetBufferLists
    binary :  netio.sys
    size :    0x800
```

And inside of drivers:

```
[0x14]
    addr        : 0xfffff8000e4db100
    sym         : nt!IopOplockFoExtLookasideList
[0x15]
    addr        : 0xfffff8000e4ee880
    sym         : nt!WmipRegLookaside
[0x16]
    addr        : 0xfffff80010e40bc0
    sym         : ACPI!BuildRequestLookAsideList
[0x17]
    addr        : 0xfffff80010e40dc0
    sym         : ACPI!RequestLookAsideList
[0x18]
    addr        : 0xfffff80010e40c40
    sym         : ACPI!DeviceExtensionLookAsideList
[0x19]
    addr        : 0xfffff80010e40d40
    sym         : ACPI!RequestDependencyLookAsideList
[0x1a]
    addr        : 0xfffff80010e40cc0
```

```
    sym         : ACPI!ObjectDataLookAsideList
[0x17]
    addr        : 0xfffff80010e40f40
    sym         : ACPI!XswContextLookAsideList
```

Per-Processor Lookaside Lists

There's actually one more linked list of lookaside lists that we haven't talked about yet: `ExPoolLookasideListHead`. Since the first versions of Windows NT, and up until Windows `10 RS5` when the pool manager was rewritten to use the Backend Heap (again, the topic of a future book!), it leveraged a per-processor array of `32` lookaside lists, one for each indexed multiple of the pool block size. On `x86`, this basically meant any `8`-byte aligned allocation from `8` to `256` bytes, and on `x64`, any `16`-byte aligned allocation from `16` to `512` bytes.

Since there was both a paged and nonpaged pool, each `KPRCB` had two such arrays — the `PPNPagedLookasideList` and the `PPPagedLookasideList`. With Windows `8` and the introduction of the non-executable nonpaged pool, a third array was created: `PPNxPagedLookasideList`. All of these lookaside lists are therefore inserted into the same linked list head, and on our system, you can easily see how many processors (`16`) are present:

```
lkd> dx -r0 @$poolasides = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExPoolLookasideListHead, "nt!_GENERAL_LOOKASIDE",
"ListEntry")
@$poolasides = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExPoolLookasideListHead, "nt!_GENERAL_LOOKASIDE",
"ListEntry")
lkd> dx @$poolasides.Count(), d
@$poolasides.Count(), d : 1536
lkd> dx 1536 / 32 / 3
1536 / 32 / 3 : 16
lkd> dx *(int*)&nt!KeNumberProcessors
*(int*)&nt!KeNumberProcessors : 16 [Type: int]
```

Originally, this seemed exciting, as it would imply the ability to easily locate not only structures that contain a lookaside list, but in fact, any pool structure that's a multiple of the pool block size. Unfortunately, if we take a look at these lists on modern Windows `10` systems, we find that they're completely unused:

```
lkd> dx @$poolasides.Sum(p => p.TotalAllocates + p.TotalFrees)
@$poolasides.Sum(p => p.TotalAllocates + p.TotalFrees) : 0x0
```

Indeed, looking at the code in `ExAllocatePoolWithTag` and friends, this logic was completely removed as part of the heap-related changes we'll cover in a future research paper.

Executive Resources

The even cooler thing is that lookaside lists are not the only kernel structures that are linked to all other structures of the same type! Another example is the `ERESOURCE`, a structure used to implement read/write locking for drivers. Executive resources are also contained inside of many kernel structures, and can give us access to even more internal kernel information, if we know how to find them. We changed our WinDbg scripts to iterate over the linked list found in `ERESOURCE.SystemResourcesList`, starting from `nt!ExpSystemResourcesList`.

We first searched for `ERESOURCE` objects in the pool:

```
[0xb8]
    address : 0xffff97884bf9fb90
    tag :     Ntfx
    tagDesc : Unrecognized NTFS tag (update base\published\pooltag.w)
    binary :  ntfs.sys
    size :    0x170
[0xb9]
    address : 0xffff97884bf50e80
    tag :     SeTl
    tagDesc : Security Token Lock
    binary :  nt!se
    size :    0x80
[0x4c]
    address : 0xffff97884bf9ed30
    tag :     Ntfx
    tagDesc : Unrecognized NTFS tag (update base\published\pooltag.w)
    binary :  ntfs.sys
    size :    0x170
```

And then for `ERESOURCE` objects inside of drivers:

```
[0x3c]
    addr        : 0xfffff8001268e8e0
    sym         : Ntfs!NtfsDynamicRegistrySettingsResource
[0x3d]
    addr        : 0xfffff80011211ef0
    sym         : NDIS!SharedMemoryResource
[0x3e]
    addr        : 0xfffff80012967630
    sym         : ksecpkg!g_rgCachedPagedSslProvs+0x410
```

```
[0x3f]
    addr        : 0xfffff80011a032f8
    sym         : tcpip!FlIsolationState+0x18
[0x40]
    addr        : 0xfffff80011d482e0
    sym         : mup!MupProviderTable+0x20
[0x41]
    addr        : 0xfffff80011d48100
    sym         : mup!MupiSurrogateList+0x20
[0x42]
    addr        : 0xfffff8000f4ac370
    sym         : CI!g_IgnoreLifetimeSigningEKU+0x70
[0x43]
    addr        : 0xfffff8000f4acb80
    sym         : CI!g_GRLContextLock
[0x44]
    addr        : 0xfffff80012f081c0
    sym         : netbios!g_erGlobalLock
```

We found some very interesting results that are probably worth further investigation, such as pool structures related to NTFS volume objects, structures inside `Ci.dll`, and much, much more. On our machine we found over `400 000` executive resources:

```
lkd> dx -r0 @$eresource = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExpSystemResourcesList, "nt!_ERESOURCE",
"SystemResourcesList")
@$eresource = Debugger.Utility.Collections.FromListEntry(*
(nt!_LIST_ENTRY*)&nt!ExpSystemResourcesList, "nt!_ERESOURCE",
"SystemResourcesList")
lkd> dx @$eresource.Count(),d
@$eresource.Count(),d : 400960
```

Because of the sheer number, making analysis with LINQ unwieldly, we wanted to get pool information for some of these `ERESOURCE` structures using C code and start analyzing them. Unfortunately, unlike lookaside lists, `ERESOURCE` structures don't have their pool tag as part of the structure, so we have to write a pool parser to get the pool information for each `ERESOURCE`. As we've mentioned before, as it turns out, in `RS5` and later, that is not an easy task at all, as you'll see in our upcoming research on the new backend heap-backed kernel pool.

Read our other blog posts: