From Dead Data to Digestion

Extracting Windows Fibers for your digital forensics diet

ID

- Daniel Jary (@JanielDary) – Security researcher
- Previously:
  - Senior security researcher @WithSecure/F-Secure.
  - Security research & endpoint agent developer @UKGov.
  - IR @Mandiant.
- Professional interests:
  - OS internals.
  - Reverse engineering.
  - Tool & Sensor Dev.

Agenda

1. What are Fibers?
2. Abusing Fibers
3. Extracting Fibers from memory
4. Weetabix (Proof of concept tool)

Glossary

- Heap - An area of memory reserved for data that is created for and used by a process.
- Process Environment Block (PEB) – A data structure that represents information about a process in usermode.
- Thread Environment Block (TEB) – A data structure that provides a Thread’s user-mode representation.
- Thread Information Block (TIB) – First field of the TEB, contains FiberData field & stack information about a thread.

What are Fibers?

- Microsoft Definition – “A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers”.
- My definition:
  - Stackful coroutines.
  - Manually scheduled.
  - Usermode only.
  - 1 Fiber/Thread at any one time.
- Initial use cases:
  - Databases, server-side applications.
- Modern use cases:
  - Browsers. Audio software plugins.

Current recommendation is to avoid using fibers and UMS. This advice from 2005 remains unchanged: “...
[I]nstead of spending your time rewriting your app to use fibers (and it IS a rewrite), instead it's better to rearchitect your app to use a "minimal context" model - instead of maintaining the state of your server on the stack, maintain it in a small data structure, and have that structure drive a small one-thread-per-cpu state machine.” [WhyFibers].

What are Fibers?

Thread	Fiber
Mandatory aspect of any process.	Optional aspect of a thread.
At least one Thread / process.	One Fiber / Thread at a time.
Unit of execution which the operating system allocates processor time.	Unit of execution that sits within the context of a thread object.
Usermode & Kernel Object representation.	Usermode only.
Managed by the Windows system scheduler.	Manually scheduled by the application.
Thread->Thread transition: • Requires kernel transition. • Expensive context switch == More CPU cycles.	Fiber->Fiber transition: • Occurs in usermode. • Cheap context switch == Less CPU cycles.

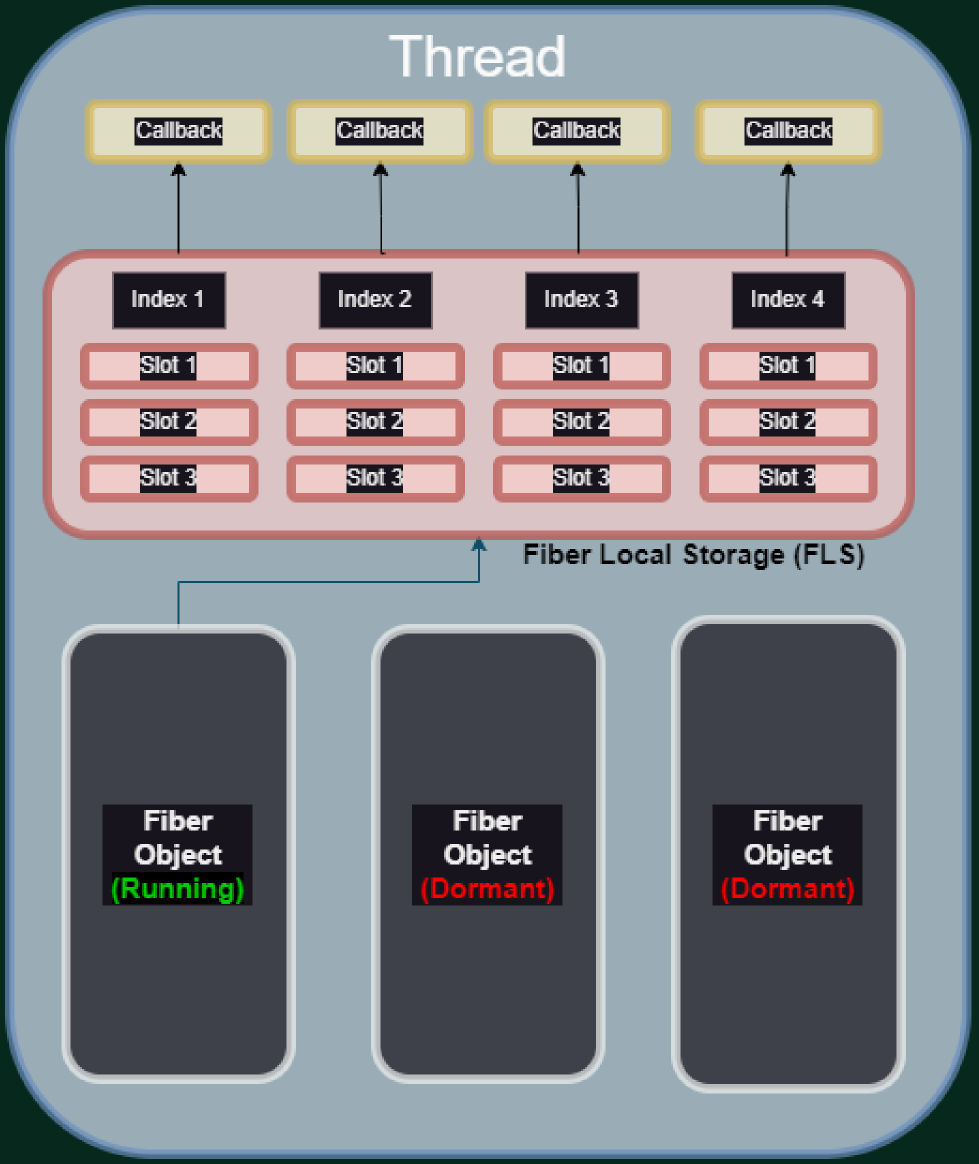
Components & Rules

Components:

- **Fiber Objects– including Fiber Data**
- **Fiber Local Storage (FLS):**
  - **Index**
  - **Slots**
- **Fiber Callback functions**

Basic Rules:

- ✓ A thread must first convert itself into a fiber.
- ✓ All fibers are equal, no “main” fibers.
- ✓ A fiber is free to create/delete another fiber.
- ✓ Only 1 fiber can run per thread at any time.



The Windows Fiber API

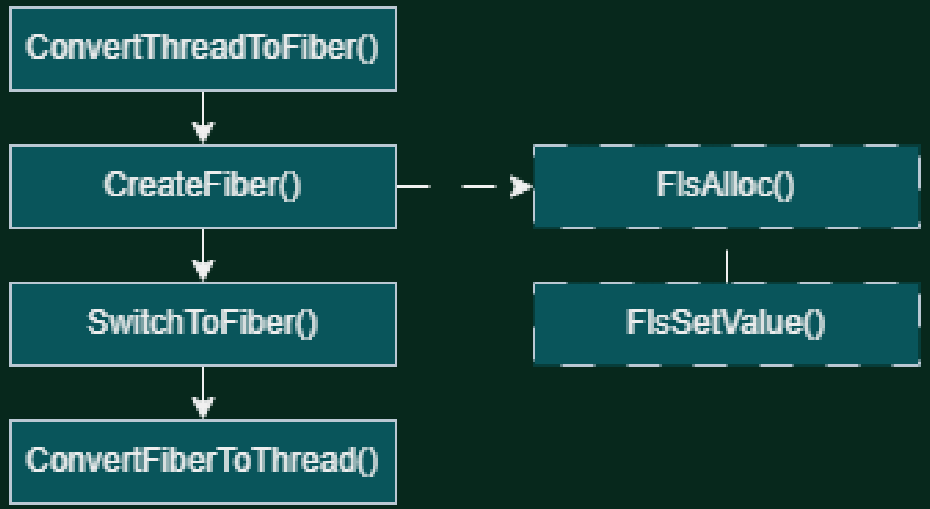
Setup, teardown & scheduling
ConvertThreadToFiber()
ConvertThreadToFiberEx()
ConvertFiberToThread()
DeleteFiber()
CreateFiber()
CreateFiberEx()
SwitchToFiber()

Fiber Local Storage
FlsAlloc()
FlsFree()
FlsSetValue()
FlsGetValue()

Local inspection
IsThreadAFiber()
GetCurrentFiber()
GetFiberData()

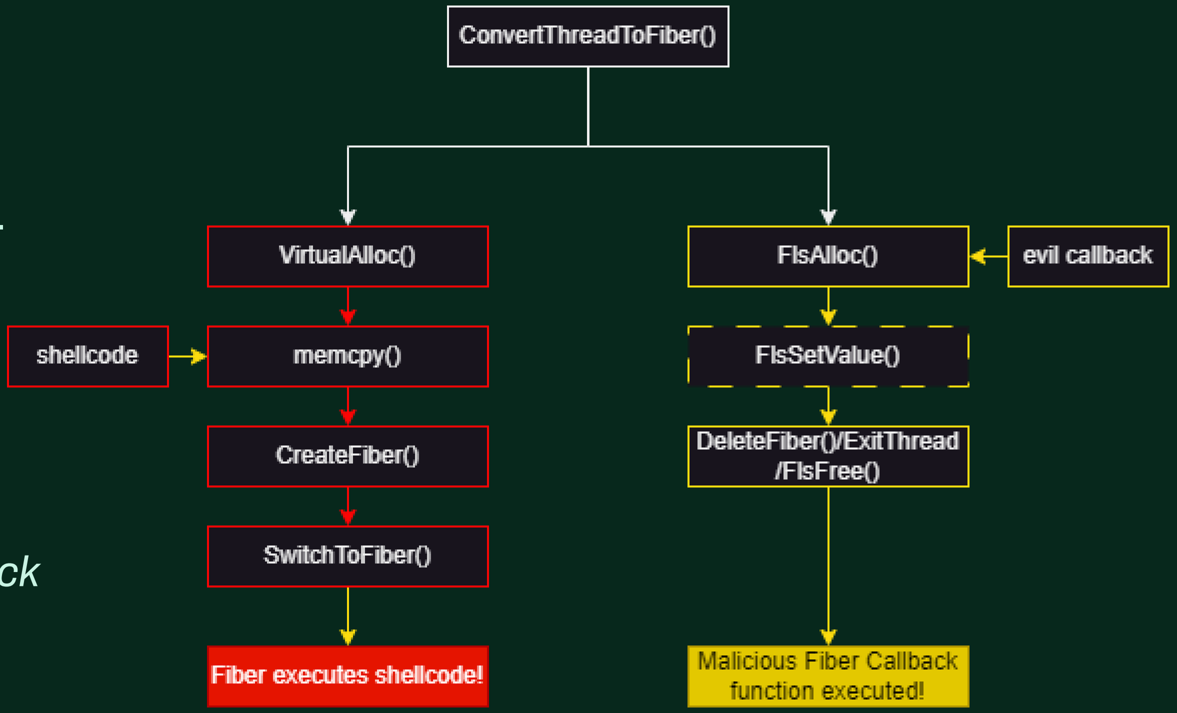
How to use Fibers

1. Thread converts itself to a fiber – ConvertThreadToFiber().
2. Create a second fiber– CreateFiber().
3. (Optional) Allocate FLS – FlsAlloc().
4. Switch to the newly created fiber – SwitchToFiber().
5. When finished, convert a fiber back to a thread – ConvertFiberToThread().



How to abuse fibers

- Executing shellcode in a local process using fibers:
    1. Convert a Thread to a Fiber.
    2. Allocate memory & copy over shellcode.
    3. Create a new fiber, supply the shellcode address.
    4. Schedule the newly created fiber.
- Fiber Local Storage and callback functions:
    1. Convert a Thread to a Fiber.
    2. Allocate FLS index, suppling an evil callback function.
    3. *(Optional) Set a FLS slot value to use as a callback parameter.*
    4. Free the FLS index / Delete fiber.



Why are fibers appealing to attackers?

Simplicity

- Easy to use API
- No complex syncronization objects
- In-built fiber safety

Obscure & niche

- Threat hunters unlikely to be looking for this
- How do they determine what is legitimate/malicious?
- Dormant Fibers hidden from view

Immature detection capability

- No Driver supplied callbacks

```
NTSTATUS PsSetCreateThreadNotifyRoutine(
  [in] PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
);
```

- Callstacks

Stack for thread 7860

```
0   ntoskrnl.exe!KeSynchronizeExecution+0x5b66
1   ntoskrnl.exe!KeWaitForMutexObject+0x1460
2   ntoskrnl.exe!KeWaitForMutexObject+0x98f
3   ntoskrnl.exe!KeWaitForMutexObject+0x233
4   ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x1bc18
5   ntoskrnl.exe!CmUnregisterMachineHiveLoadedNotification+0x1d6b8
6   ntoskrnl.exe!FsRtlRegisterFltMgrCalls+0x6cfc1
7   ntoskrnl.exe!RtlRandomEx+0x1f64
8   ntoskrnl.exe!setjmpex+0x838c
9   ntoskrnl.exe!setjmpex+0x1cf6
10  KERNELBASE.dll!DebugBreak+0x3
11  fls_callback_view.exe!fiberFunc+0xec
```

- No remote enumeration functions
- No opensource tools
- No ETW providers

Extracting Fibers from Memory

Goals:

1. Remotely Identify Threads using Fibers.
2. Identify how a Fiber is structured & stored.
3. Associate a Fiber with the correct FLS, Callbacks & TID.

The challenges:

- No remote enumeration functions.
- No opensource tools.
- Extremely limited documentation. No diagrams, no internals. (One short paragraph below in the whole of the current Windows Internals books!).

Fibers

Fibers allow an application to schedule its own threads of execution rather than rely on the priority-based scheduling mechanism built into Windows. Fibers are often called *lightweight threads*. In terms of scheduling, they're invisible to the kernel because they're implemented in user mode in Kernel32.dll. To use fibers, you first make a call to the Windows `ConvertThreadToFiber` function. This function converts the thread to a running fiber. Afterward, the newly converted fiber can create additional fibers via the `CreateFiber` function. (Each fiber can have its own set of fibers.) Unlike a thread, however, a fiber doesn't begin execution until it's manually selected through a call to the `SwitchToFiber` function. The new fiber runs until it exits or until it calls `SwitchToFiber`, again selecting [illegible] fiber to run. For more information, see the Windows SDK documentation on fiber functions.

Note

Using fibers is usually not a good idea. This is because they are invisible to the kernel. They also have issues such as sharing thread local storage (TLS) because several fibers can be running on the same thread. Although fiber local storage (FLS) exists, this does not solve all sharing issues, and I/O-bound fibers will perform poorly regardless. Additionally, fibers cannot run concurrently on more than one processor, and are limited to cooperative multi-tasking only. In most scenarios, it's best to let the Windows kernel handle scheduling by using the appropriate threads for the task at hand.

KERNELBASE!IsThreadAFiber

(Associated goal 1/3 - Remotely Identify Threads using Fibers)

Determines whether the current thread is a fiber. No remote option available:

- Third bit from the SameTebFlags is set == Thread is using Fibers.

```
BOOL __stdcall IsThreadAFiber()
{
  return (NtCurrentTeb()->SameTebFlags >> 2) & 1;
}
```

Write our own Fiber program, validate using WinDbg.

```
LPVOID lpFiberData = HeapAlloc(GetProcessHeap(), 0, 0x10);
LPVOID lpFirstFiber = NULL;
memset(lpFiberData, 0x42, 0x10);
lpFirstFiber = ConvertThreadToFiber(lpFiberData);
DebugBreak();
```

```
0:003> dt 0x000000106e8fd000 ntdll!_TEB Same*
   +0x17ee SameTebFlags : 4
```

How do we remotely enumerate fibers:

1. CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0) to take a snapshot of all thread IDs.
2. OpenThread() + NtQueryInformationThread() to get THREAD_BASIC_INFORMATION->TebBaseAddress.
3. ReadProcessMemory() to collect TebBaseAddress+SameTebFlags offset.

GetFiberData() & GetCurrentFiber()

(Associated goal 2/3 - Identify how a Fiber is structured & stored)

- Macros inside winnt.h.
- GetFiberData():
  - Retrieves the fiber data associated with the current fiber.
  - Value inside TEB.NT_TIB.FiberData field.
- GetCurrentFiber():
  - Returns the address of the current fiber.
  - Address of TEB.NT_TIB.FiberData field.
  - Implicitly reveals the first field in a Fiber Object is the FiberData field.

winnt.h

weetabix

```
__forceinline
PVOID
GetCurrentFiber (
    VOID
    )

{

    return (PVOID)__readgsqword(FIELD_OFFSET(NT_TIB, FiberData));
}

__forceinline
PVOID
GetFiberData (
    VOID
    )

{

    return *(PVOID *)GetCurrentFiber();
}
```

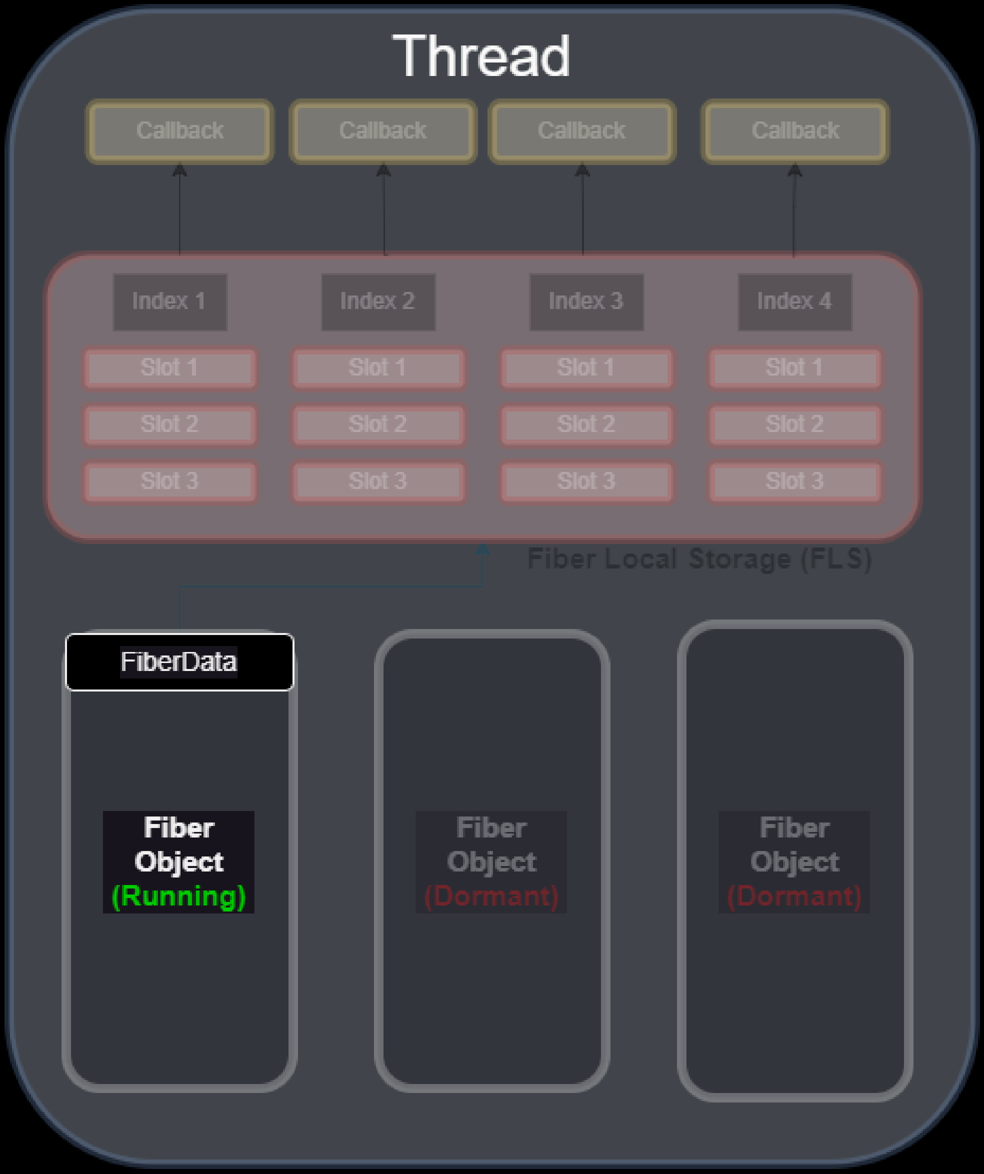
After step 1

What do we now have?

- Address of executing fiber object - (&FiberData).
- Executing fiber data - (*FiberData).

The next step?

- Identify the remaining Fiber Object fields.
- Collect dormant fibers.



KERNELBASE!ConvertThreadToFiber

(Associated goal 2/3 - Identify how a Fiber is structured & stored)

```
LPVOID ConvertThreadToFiber(
  [in, optional] LPVOID lpParameter
);
```

`[in, optional] lpParameter`

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the GetFiberData macro.

```
v5 = NtCurrentTeb();                              // <- TEB collected
if ( (v5->SameTebFlags & 4) != 0 )                // <- Is our current thread a Fiber
{
  v3 = 1280;
  goto LABEL_3;
}
Heap = (unsigned __int64)RtlAllocateHeap(NtCurrentPeb()->ProcessHeap, KernelBaseGlobalData, 0x530ui64);// <- Allocate heap block
v7 = (_QWORD *)Heap;
if ( !Heap )
{
  v3 = 8;
  goto LABEL_3;
}
*(_QWORD *)Heap = lpParameter;                    // <- Set FiberData
StackBase = (unsigned __int64)v5->NtTib.StackBase;
*(_QWORD *)(Heap + 16) = StackBase;
v9 = BasepFiberCookie ^ StackBase;
*(_QWORD *)(Heap + 24) = v5->NtTib.StackLimit;// Set TEB/TIB values to heap block
v10 = Heap ^ v9;
*(_QWORD *)(Heap + 32) = v5->DeallocationStack;
*(_QWORD *)(Heap + 8) = v5->NtTib.ExceptionList;
*(_QWORD *)(Heap + 1296) = v5->FlsData;
*(_DWORD *)(Heap + 1304) = v5->GuaranteedStackBytes;
*(_WORD *)(Heap + 1308) = v5->SameTebFlags & 0x200;
ActivationContextStackPointer = v5->ActivationContextStackPointer;
v7[165] = 0i64;
v7[160] = 0i64;
v7[161] = ActivationContextStackPointer;
result = v7;
v7[164] = v10;
*((_DWORD *)v7 + 24) = 1048587;
v5->SameTebFlags |= 4u;
v5->NtTib.FiberData = v7;
return result;
```

What does this tell us?

- Fiber Objects are stored in requested heap allocations of 0x530 bytes. (x64)
- Several fields from the TEB/TIB are used to populate a Fiber Object.

Building out the Fiber object

(Associated goal 2/3 - Identify how a Fiber is structured & stored)

- Decompile remaining setup, teardown & scheduling functions.
- Uncover new fields inside Fiber object.
- Test against our own Fiber C++ program.

Setup, teardown & scheduling
ConvertThreadToFiber()
ConvertThreadToFiberEx()
ConvertFiberToThread()
DeleteFiber()
CreateFiber()
CreateFiberEx()
SwitchToFiber()

```
00000000 fiber_object_t  struc ; (sizeof=0x530, align=0x10, mappedto_296)
00000000 FiberData       dq ?                    ; offset
00000008 ExceptionList   dq ?                    ; offset
00000010 StackBase       dq ?
00000018 StackLimit      dq ?
00000020 AllocatedStackBase dq ?
00000028                 db ? ; undefined
00000029                 db ? ; undefined
0000002A                 db ? ; undefined
0000002B                 db ? ; undefined
0000002C                 db ? ; undefined
0000002D                 db ? ; undefined
0000002E                 db ? ; undefined
0000002F                 db ? ; undefined
00000030 FiberContext    CONTEXT ?
00000500 Wx86Tib         dq ?                    ; offset
00000508 ActivationContextStackPointer dq ?      ; offset
00000510 FlsData         dq ?                    ; offset
00000518 GuaranteedStackBytes dd ?
0000051C TebFlags        dw ?
0000051E                 db ? ; undefined
0000051F                 db ? ; undefined
00000520 XoredCookie     dq ?
00000528 ShadowStack     dq ?
00000530 fiber_object_t  ends
```

=

Scanning the NT heap for Fiber objects

(Associated goal 2/3 - Identify how a Fiber is structured & stored)

1. Identify process heaps

```
Identify heaps process heaps
0:002 > dt ntdll!_PEB
   + 0x0e8 NumberOfHeaps
   + 0x0ec MaximumNumberOfHeaps
   + 0x0f0 ProcessHeaps
```

2. Collect NT heaps & heap segments within

```
Read segment signature of heap from _HEAP Header
dt ntdll!_HEAP
0:002> dt ntdll!_HEAP
   + 0x000 Segment          : _HEAP_SEGMENT
   + 0x000 Entry            : _HEAP_ENTRY
   + 0x010 SegmentSignature
   + 0x018 SegmentListEntry : _LIST_ENTRY
```

3. Decode & enumerate heap entries from segments

```
Encoding - Used to decode _HEAP_ENTRY values.
FirstEntry - Provides a ptr to first Heap block to begin enumeration.
0:002 > dt ntdll!_HEAP
   + 0x07c EncodeFlagMask
   + 0x080 Encoding : _HEAP_ENTRY
   + 0x040 FirstEntry : Ptr64 _HEAP_ENTRY
```

Decode & Enumerate HEAP_ENTRYs

```
Get HEAP_ENTRY Size, Flags, PreviousSize (PreviousBlockSize) & UnusedBytes.
0:002 > dt ntdll!_HEAP_ENTRY
   + 0x008 Size
   + 0x00a Flags
   + 0x00b SmallTagIndex
   + 0x00c PreviousSize
   + 0x00f UnusedBytes
```

4. Add new requestedBytes field

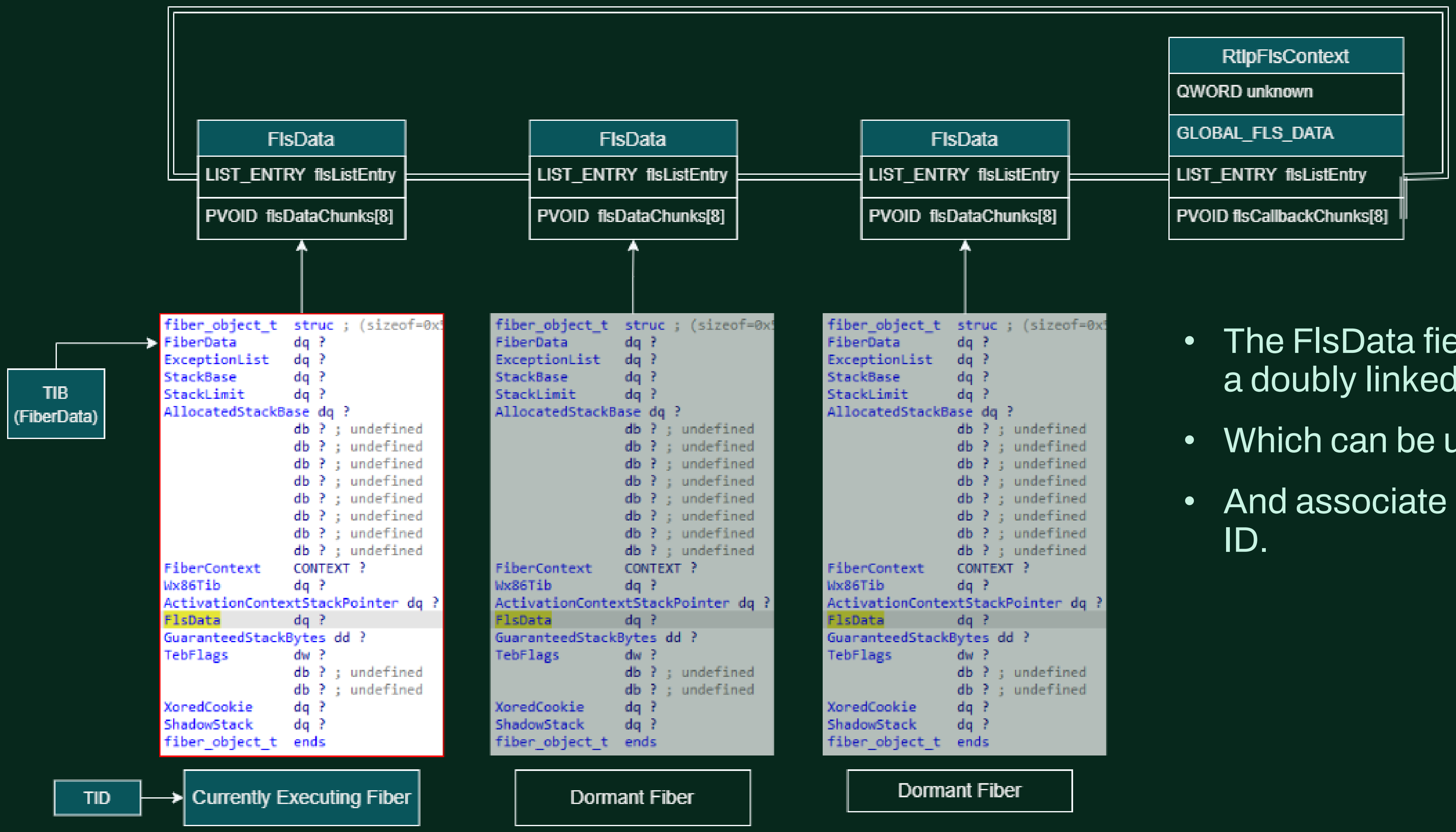
requestedBytes = Size - UnusedBytes

5. Collect potential Fiber object heap entries

Requested size is 0x530

Validating heap Fiber Objects using FLS

(Associated goal 3/3 – Associate Fiber with the correct FLS, Callbacks & TID)



- The FlsData field (in a fiber object) is part of a doubly linked list.
- Which can be used to find Dormant fibers.
- And associate dormant fibers with a thread ID.

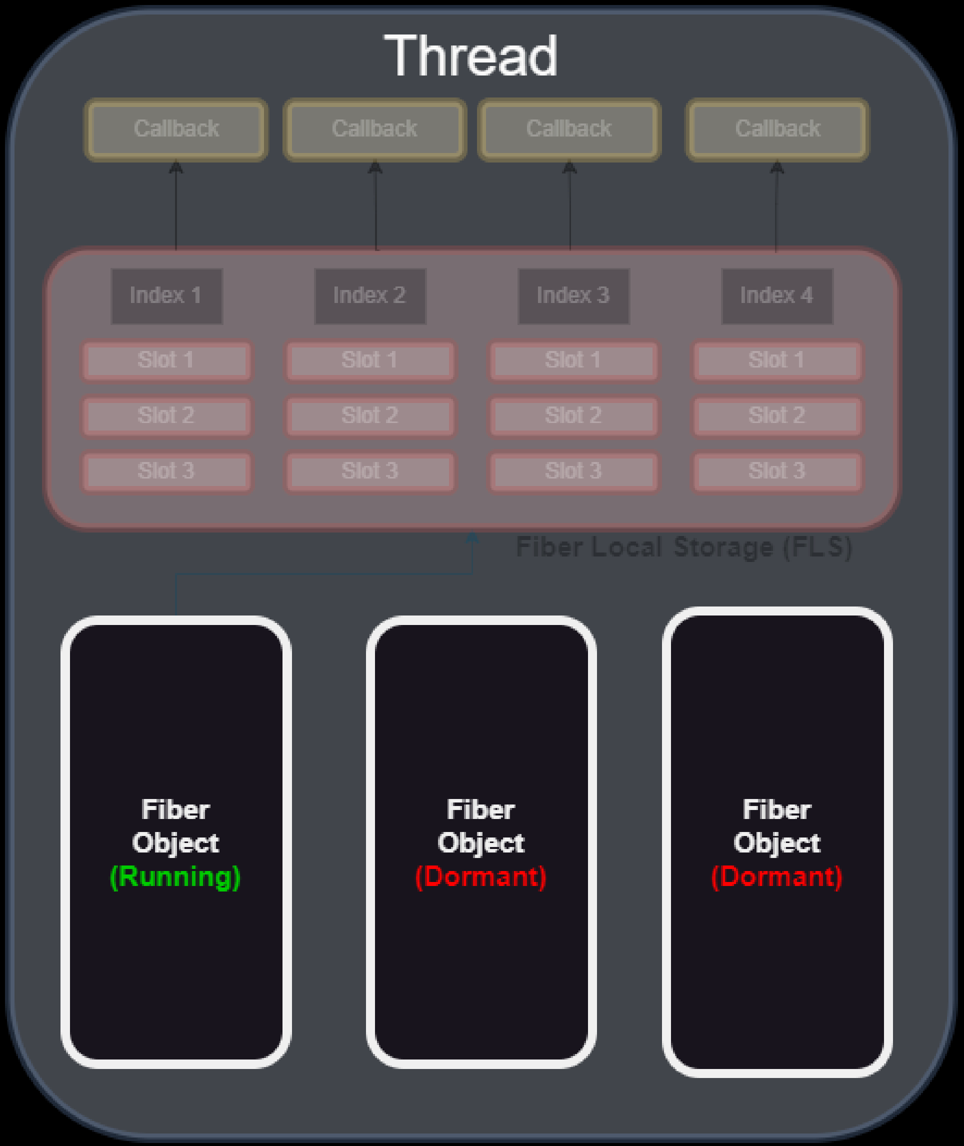
After step 2

What do we now have?

- A complete Fiber object structure.
- All Fiber objects (both dormant & running) associated with a thread.

The next step?

- Identify the number of FLS indexes in use.
- Identify the FLS slots used by each Fiber.



Extracting FLS Slot values - NTDLL!RtlFlsGetValue

(Associated goal 3/3 – Associate Fiber with the correct FLS, Callbacks & TID)

- The maximum FLS index is 4079.
- FLS slot values can be determined using the FiberData field.

```
PVOID __fastcall RtlFlsGetValue(DWORD dwFlsIndex, _QWORD *flsValue)
{
  FLS_DATA *FlsData; // rax
  DWORD xorValue; // r10d
  PVOID flsChunk; // rdx
  PVOID flsSlotAddr; // rax
  PVOID flsValueAddr; // rax

  FlsData = (FLS_DATA *)NtCurrentTeb()->FlsData;
  if ( dwFlsIndex - 1 > 4078 || !FlsData )
    return (PVOID)STATUS_INVALID_PARAMETER;
  xorValue = dwFlsIndex + 0x10;
  _BitScanReverse(&dwFlsIndex, dwFlsIndex + 0x10);
  flsChunk = FlsData->flsDataChunks[dwFlsIndex - 4];
  if ( flsChunk
    && (flsSlotAddr = (char *)flsChunk + 8 * ((unsigned int)(1 << dwFlsIndex) ^ (unsigned __int64)xorValue) + 8) != 0i64 )
  {
    flsValueAddr = *(PVOID *)flsSlotAddr;
  }
  else
  {
    flsValueAddr = 0i64;
  }
  *flsValue = flsValueAddr;
  return 0i64;
}
```

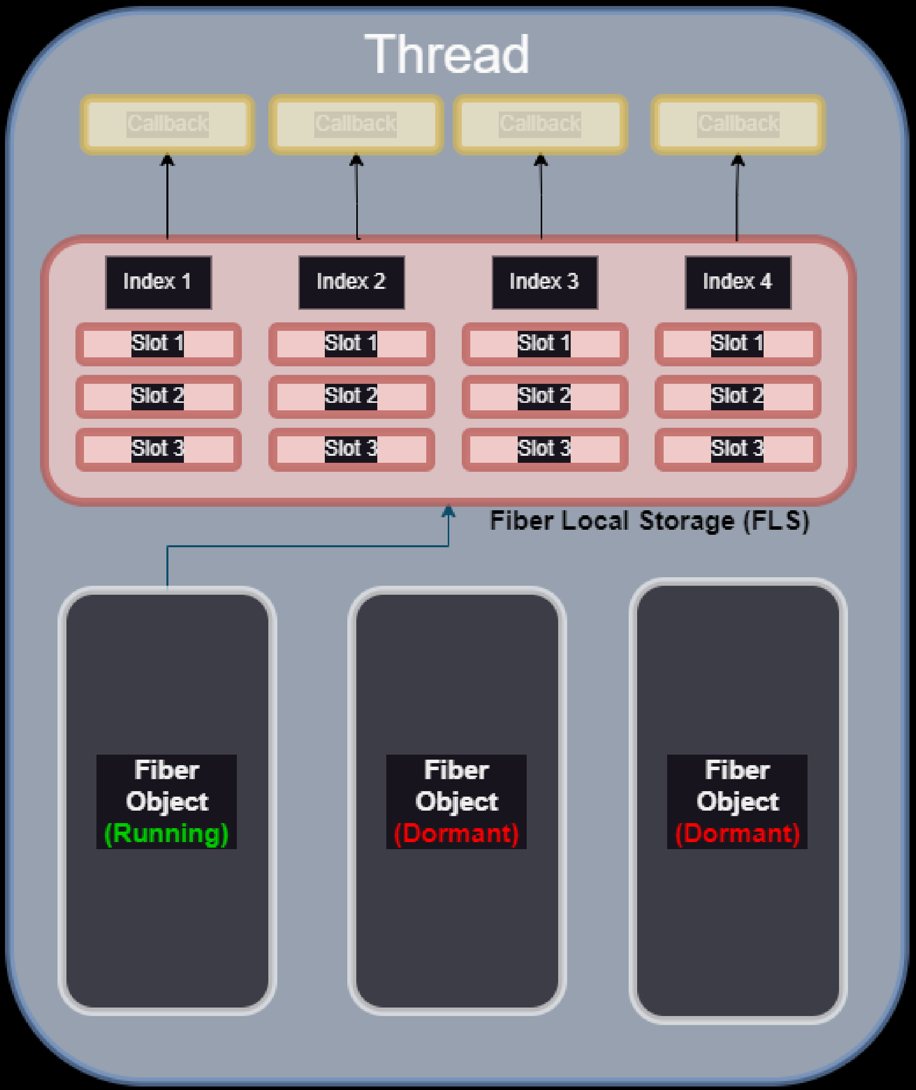
After step 3

What do we have?

- Fiber object fields.
- Dormant fiber objects.
- Associated TIDs.
- FLS.

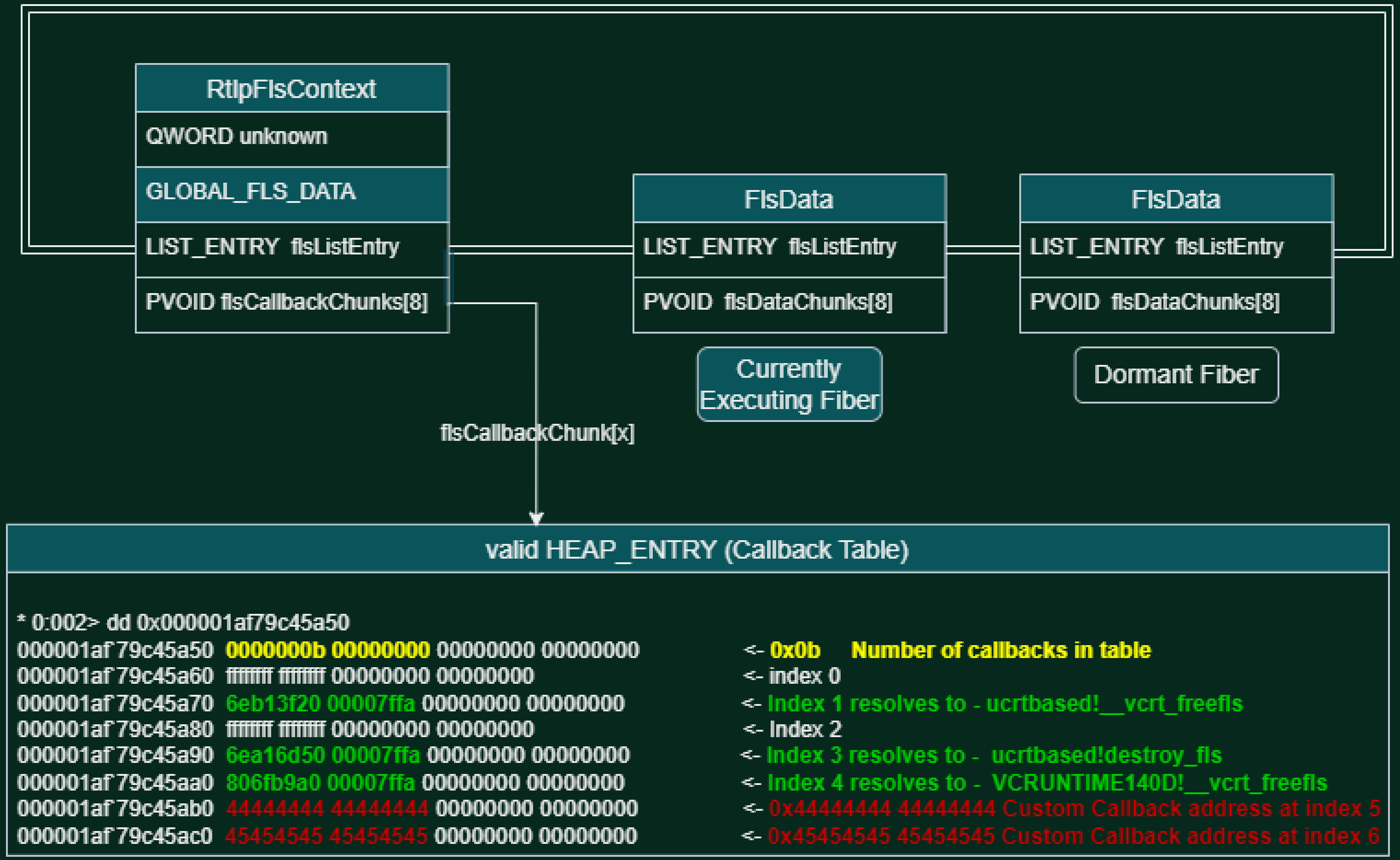
The next step?

- Identify the correct callbacks.



FLS Callbacks

(Associated goal 3/3 – Associate Fiber with the correct FLS, Callbacks & TID)



- Pointer to FLS callback table exists in the RtlFlsContext member of the linked List.
- Callback table indexes == FLS slot indexes.

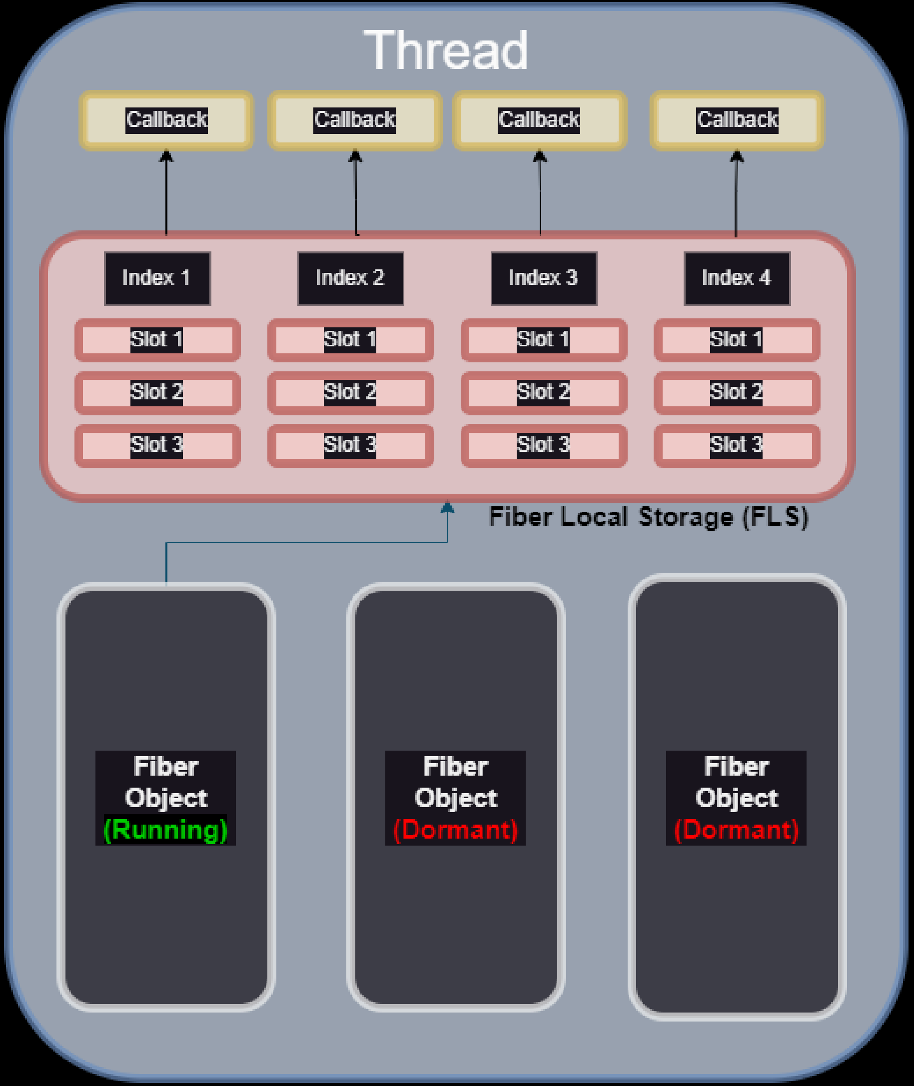
After step 4 – Raw telemetry achieved!

What do we have?

- Fiber object fields
- Dormant fiber objects
- Associated TIDs
- FLS
- FLS Callbacks

Goals :

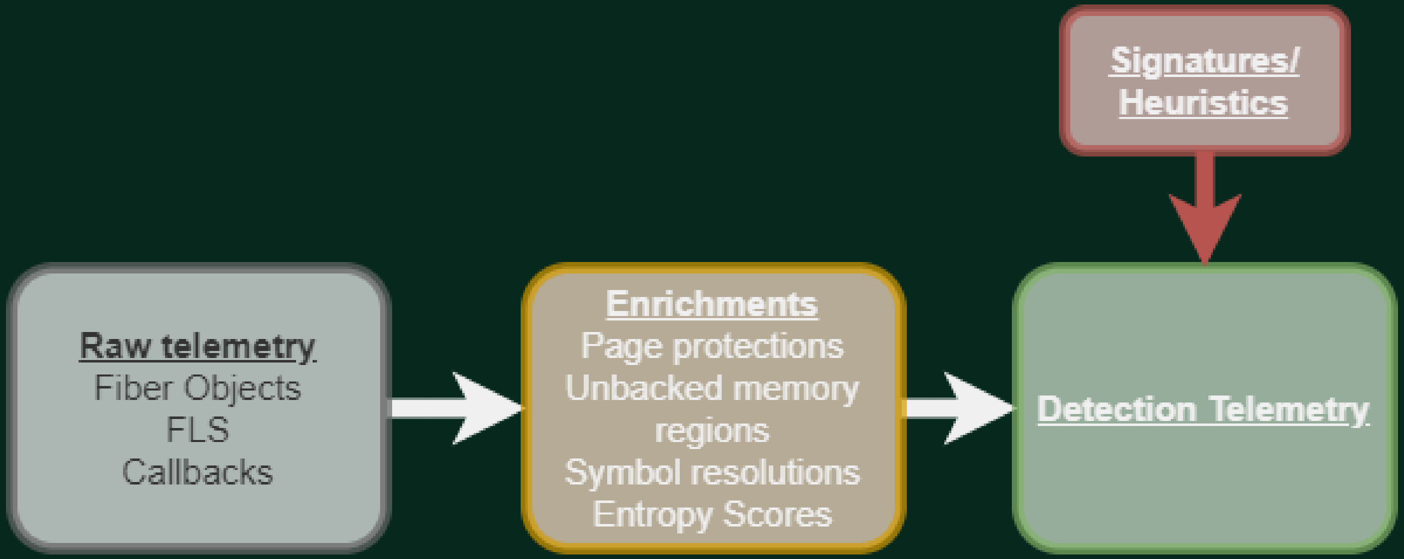
1. Remotely Identify Threads using Fibers. ✓
2. Identify how a Fiber is structured & stored. ✓
3. Associate a Fiber with the correct FLS, Callbacks & TID. ✓



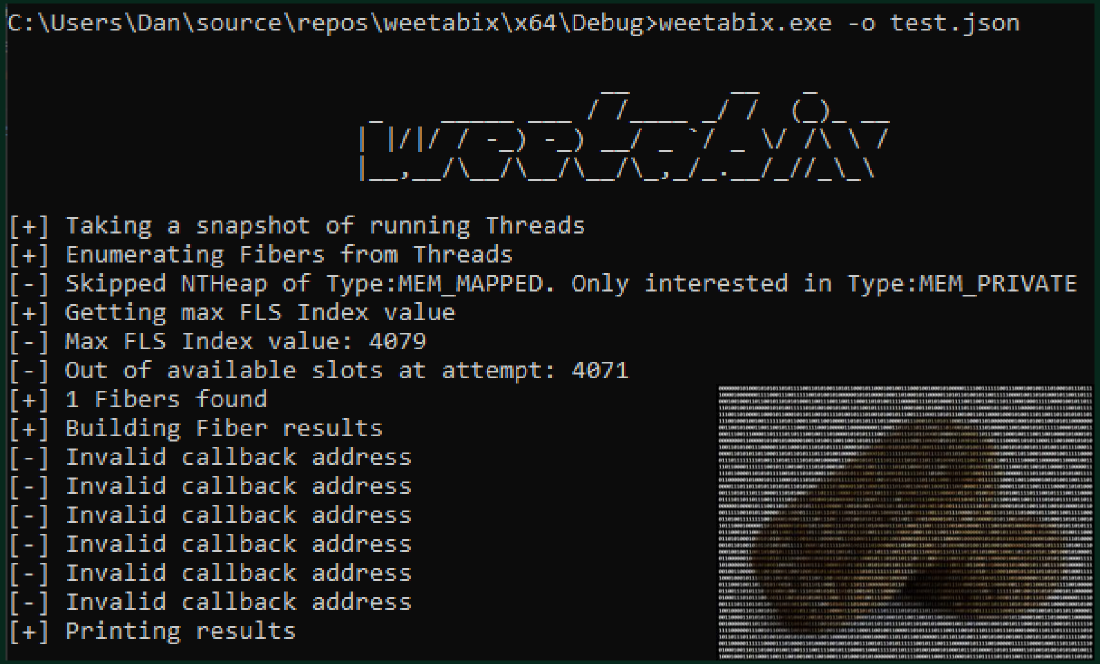
Enrichment of Fiber telemetry for detection purposes

Goals Achieved to generate raw telemetry:

- ✓ Remotely Identify Threads using Fibers.
- ✓ Identify how a Fiber is structured & stored.
- ✓ Associate a Fiber with the correct FLS, Callbacks & TID.



Weetabix - POC tool



- Written in C++.
- Automates the enumeration of Fibers from currently running threads.
- Applies a set of enrichments to:
  - Fiber Objects
  - FLS
  - Fiber Callback telemetry.
- Outputs data into NDJSON file.
- https://github.com/JanielDary/weetabix

Detection example – CS Artefact Kit

- June 2022 – Cobalt strike implements thread stack spoofing using Fibers.

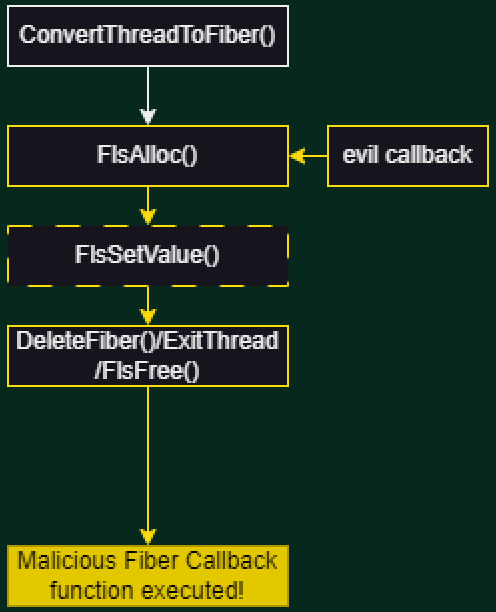
```
June 13, 2022 - Arsenal Kit
-------------
++ Artifact Kit
+ Add support for thread stack spoofing.  For more information see kits/artifact/README_STACK_SPOOF.md
```

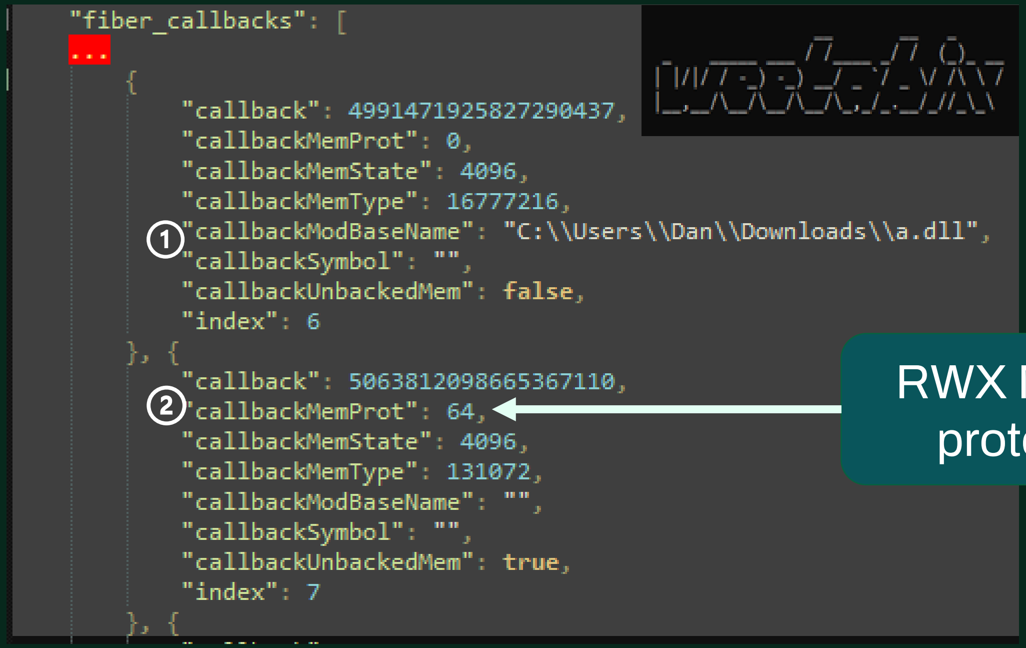
The Arsenal Kit has been updated to implement stack spoofing using the Microsoft Fiber functions. This technique does not invalidate the stack, and exercises a normal code execution workflow, leveraging the stack switching side effect of Microsoft Fibers.

- Unorthodox implementation:
  - Single fiber use!
  1. Unbacked FiberData.
  2. No FLS data.
  3. No FLS callbacks.


```
{
    "fiber_callbacks": [], 3
    "fiber_data_addr_module": "",
    "fiber_data_address": 2677486511408,
    "fiber_data_entropy": 0.0,
    "fiber_data_mem_prot": 4,
    "fiber_data_mem_state": 4096,
    "fiber_data_mem_type": 131072,
  1 "fiber_data_unbacked": true,
    "fiber_local_storage": [] 2
    "fiber_local_storage_address": 0,
    "pid": 10316,
    "process_name": "C:\\Users\\Dan\\Desktop\\fiber_example\\file.exe",
    "tid": 7840
}
```

Detection example 2 – Callback manipulation





RWX Memory protection

Can we go further?

Key takeaways

1. Threat actors can utilize/target obscure operating system concepts circumvent traditional telemetry, helping them evade blue team functions.
2. No telemetry == No detections. So, building purpose-built telemetry is vital to EDR product development!
3. Building new telemetry often requires deep understanding, but this can lead to high value low-volume & therefore low-cost solutions especially when deployed over enterprise scale environments.

Resources

- https://www.geoffchappell.com/
- https://doxygen.reactos.org/
- https://github.com/wine-mirror/wine
- https://devblogs.microsoft.com/oldnewthing/20191011-00/?p=102989
- https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1364r0.pdf
- William Burgess (@joehowwolf) – “What exactly are Fibers Dan?”

Thankyou!

@JanielDary
github.com/JanielDary/weetabix

Questions?