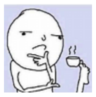
Hypervisor Detection with SystemHypervisorDetailInformation

medium.com/@matterpreter/hypervisor-detection-with-systemhypervisordetailinformation-26e44a57f80e

Matt Hand September 15, 2023



Matt Hand

I was recently doing some work that involved detecting whether or not the system my code was executing on was virtualized and collecting some details about the hypervisor. There are a number of documented ways of doing this (some more hacky than others), but the one that caught my eye was using `NtQueryInformationSystem()` and the `SystemHypervisorDetailInformation` information class. While the function itself is documented, official documentation from Microsoft regarding the information class and structure returned to the caller are notably missing.

As with most things in the undocumented Windows world, Geoff Chappell, may he rest in peace, and SystemInformer (previously Process Hacker) both had information related to how this class and structure are used. Additionally, the Hyper-V Hypervisor Top-Level Functional Specification (TLFS) provides a solid reference on what features are exposed and how the structure is populated.

> Microsoft publishes and maintains a specification called the TLFS which describes a hypervisor's features which can be observed from a guest operating system (i.e., a VM). These features are exposed to guests via the "Hv#1" interface, which will be discussed shortly. This specification defines features of the hypervisor, outlines datatypes used, provides a reference to supported hypercalls, and many other useful pieces of information. The internal functions that Windows calls when the `SystemHypervisorDetailInformation` information class is passed work almost exclusively with TLFS-compliant hypervisors.

To supplement the great work done by these folks and others who have contributed information publicly, I wanted to document the internal workings, "gotcha's," and information returned in a central location for anyone needing to do hypervisor discovery using this method.

NtQuerySystemInformation() and SystemHypervisorDetailInformation

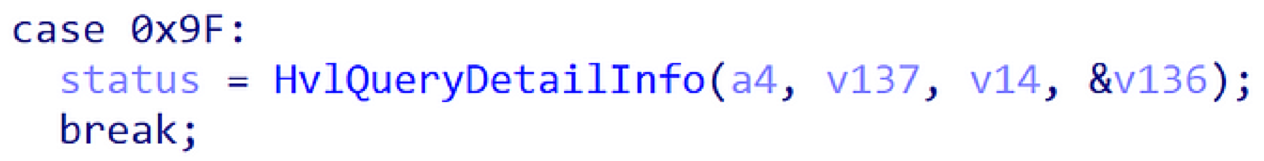
`NtQuerySystemInformation()` is a function exported by ntdll.dll that takes a value called a system information class that describes the type of information to be retrieved and a pointer to a structure which will hold the returned information. There are many information classes with which you may already be familiar, such as `SystemBasicInformation` (0) and `SystemHandleInformation` (0x10). The one most relevant to hypervisor identification is the aptly named `SystemHypervisorDetailInformation` (0x9f).

Though no official documentation is provided by Microsoft, others have — and I'll show you how later — found that when called, a `SYSTEM_HYPERVISOR_DETAIL_INFORMATION` structure is returned to the caller.

```
// https://github.com/winsiderss/phnt/blob/master/ntexapi.h
typedef _{
    ULONG Data[4];
} HV_DETAILS, *PHV_DETAILS;

      HV_DETAILS HvVendorAndMaxFunction;    HV_DETAILS HypervisorInterface;    HV_DETAILS
HypervisorVersion;    HV_DETAILS HvFeatures;    HV_DETAILS HwFeatures;    HV_DETAILS
EnlightenmentInfo;    HV_DETAILS ImplementationLimits;} SYSTEM_HYPERVISOR_DETAIL_INFORMATION,
*PSYSTEM_HYPERVISOR_DETAIL_INFORMATION;
```

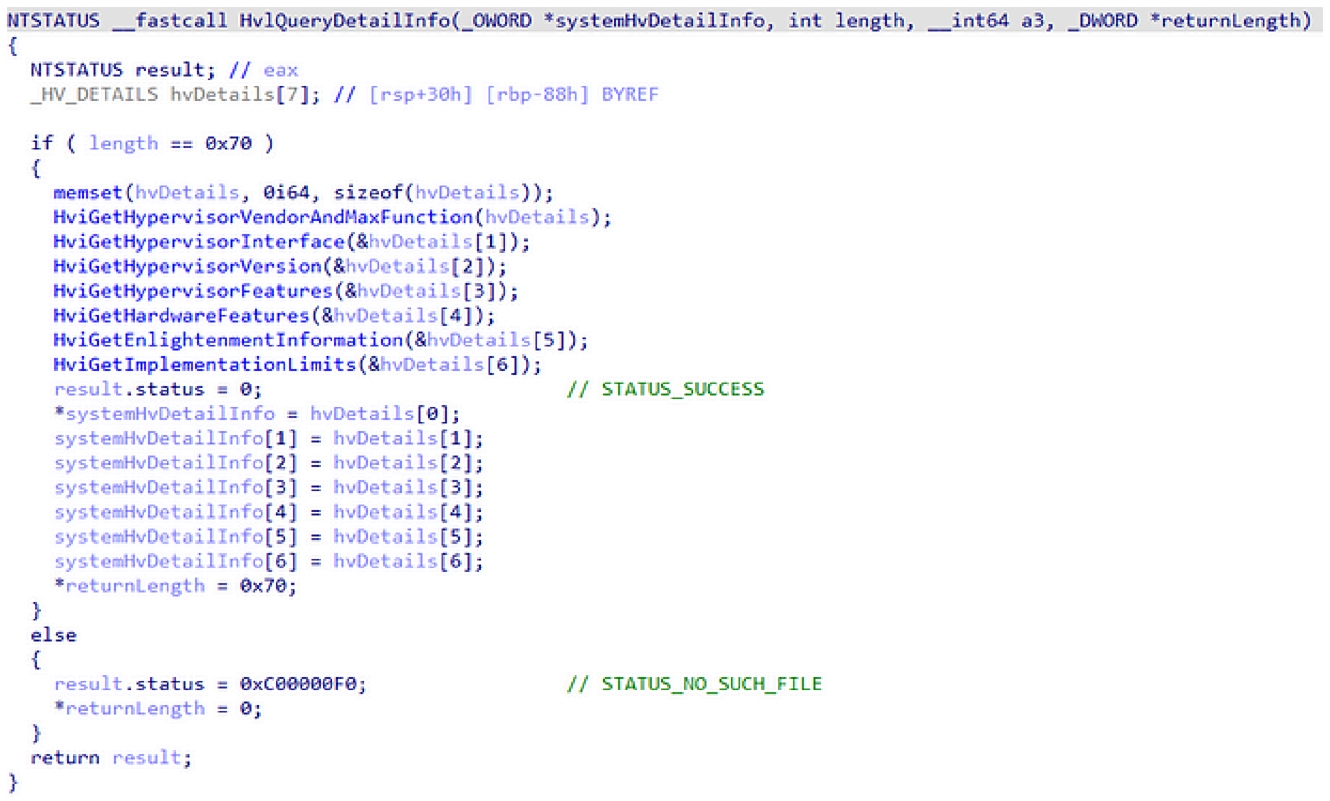
Internally, when the syscall for `NtQuerySystemInformation()` occurs and control is transitioned to the kernel, the information class identifier is evaluated in a switch case. If there is no match found, which there isn't for `SystemHypervisorDetailInformation`, `nt!ExpQuerySystemInformation()` is called. This function contains yet another switch case, this time matching on our value of 0x9f and calling `nt!HvlQueryDetailInfo()` where we'll begin digging in.



```
case 0x9F:
  status = HvlQueryDetailInfo(a4, v137, v14, &v136);
  break;
```

HvlQueryDetailInfo Internals

This function is responsible for populating the seven `HV_INFORMATION` structures that make up the `SYSTEM_HYPERVISOR_DETAIL_INFORMATION` structure that is returned to the caller of `ntdll!NtQuerySystemInformation()`.


```
NTSTATUS __fastcall HvlQueryDetailInfo(_OWORD *systemHvDetailInfo, int length, __int64 a3, _DWORD *returnLength)
{
  NTSTATUS result; // eax
  _HV_DETAILS hvDetails[7]; // [rsp+30h] [rbp-88h] BYREF

  if ( length == 0x70 )
  {
    memset(hvDetails, 0i64, sizeof(hvDetails));
    HviGetHypervisorVendorAndMaxFunction(hvDetails);
    HviGetHypervisorInterface(&hvDetails[1]);
    HviGetHypervisorVersion(&hvDetails[2]);
    HviGetHypervisorFeatures(&hvDetails[3]);
    HviGetHardwareFeatures(&hvDetails[4]);
    HviGetEnlightenmentInformation(&hvDetails[5]);
    HviGetImplementationLimits(&hvDetails[6]);
    result.status = 0;                          // STATUS_SUCCESS
    *systemHvDetailInfo = hvDetails[0];
    systemHvDetailInfo[1] = hvDetails[1];
    systemHvDetailInfo[2] = hvDetails[2];
    systemHvDetailInfo[3] = hvDetails[3];
    systemHvDetailInfo[4] = hvDetails[4];
    systemHvDetailInfo[5] = hvDetails[5];
    systemHvDetailInfo[6] = hvDetails[6];
    *returnLength = 0x70;
  }
  else
  {
    result.status = 0xC00000F0;                 // STATUS_NO_SUCH_FILE
    *returnLength = 0;
  }
  return result;
}
```

Assuming that the appropriate length (0x70) has been received from the caller, the function zeroes out `v7`, a temporary `SYSTEM_HYPERVISOR_DETAIL_INFORMATION` structure. Then a series of seven internal functions with the "Hvi" prefix are called. These functions are each responsible for populating one of the member `HV_DETAIL` structures with a specific piece (or pieces) of information. These internal functions also build off of or reference each other in many cases.

HviGetHypervisorVendorAndMaxFunction

Internally, this function first calls `nt!HviIsAnyHypervisorPresent()` which performs two actions. First, it queries CPUID leaf 1 and checks if the most significant bit in ECX is 1. This bit is referred to as the "hypervisor present bit" by Microsoft. If this set, the function then queries the 0x40000001 leaf, which is used to obtain vendor-neutral interface identification information.

> In the context of the x86 `cpuid` instruction, leaves refer to groups of information that can be retrieved. A caller moves a leaf identifier into the EAX register and executes `cpuid`, at which point information from the corresponding leaf will be moved into EAX, EBX, ECX, and EDX. The types and format of the data moved into these registers is dependent on the leaf queried.

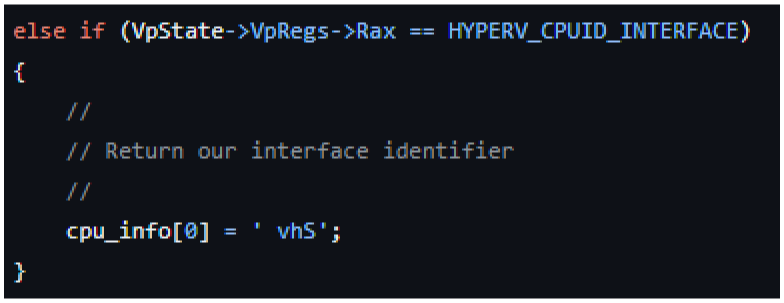
Specifically, checks the value returned in EAX to see if it matches `Xbnv`, the interface identifier for the Xbox Nanovisor, and returns false if so. Otherwise, it will return true. (Note: this same logic can be found in `nt!HviIsXboxNanovisorPresent()`). We'll see this function be called at the beginning of every subsequent function but we'll only discuss it here.

So now that `HviGetHypervisorVendorAndMaxFunction()` knows that the system is virtualized, it can continue its execution. It queries leaf `0x40000000` which returns two pieces of information — the maximum function and the vendor ID. There are a number of known vendor IDs (Hyper-V's being "Microsoft Hv") but remember that a hypervisor developer can set this value to whatever they'd like so it can't be assumed to be any of the known IDs in all cases. Even Microsoft recommend only using the information stored here for "reporting and diagnostic purposes." The maximum function value is far more useful to us. This value tells us up to what leaf number we can query. While we're guaranteed to have at least up to 0x4000005 on Microsoft hypervisors, we need to get this maximum value so that we're not querying higher than what is supported by the current VMM. After completing this query, EAX contains the maximum value and EBX, ECX, and EDX contain the vendor ID.

HviGetHypervisorInterface

The next function called , `nt!HviGetHypervisorInterface()`, queries CPUID leaf 0x40000001 after first calling `nt!HviIsAnyHypervisorPresent()`. The value stored in EAX is a vendor-neutral interface identifier. On TLFS-conforming hypervisors, this will be "Hv#1" and EBX, ECX, and EDX are reserved. Hypervisors conforming to this interface also guarantee the availability and structure of leaves 0x40000002–0x40000006.

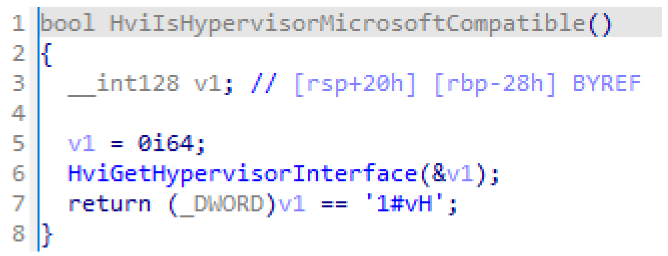
On nonconforming hypervisors, this can be anything. For instance, Alex Ionescu's SimpleVisor project sets this value to "Shv ".



```
else if (VpState->VpRegs->Rax == HYPERV_CPUID_INTERFACE)
{
    //
    // Return our interface identifier
    //
    cpu_info[0] = ' vhS';
}
```

HviGetHypervisorVersion — 0x40000002

In the next function, the kernel calls a new internal function: `HviIsHypervisorMicrosoftCompatible()`. This function is simply a wrapper for `HviGetHypervisorInterface()` that returns a boolean based on whether the hypervisor reports "Hv#1" as its interface identifier.



```
bool HviIsHypervisorMicrosoftCompatible()
{
  __int128 v1; // [rsp+20h] [rbp-28h] BYREF

  v1 = 0i64;
  HviGetHypervisorInterface(&v1);
  return (_DWORD)v1 == '1#vH';
}
```

If this function returns true, `HviGetHypervisorVersion()` queries leaf 0x40000002 to get the hypervisor's version information. The CPUID instruction will set EAX to the build number and bits 31–16 of EBX to the major version with bits 15–0 containing the minor version. As an example, Hyper-V would return something like:

- : 10
- : 0
- : 22000

HviGetHypervisorFeatures — 0x40000003

Here is where things start to get a little messy. This function again runs `HviIsHypervisorMicrosoftCompatible()`, querying leaf 0x40000003 if true. This leaf will return bitmasks in all registers that correlate to different sets of hypervisor features.

EAX and EBX, this will correspond to bits 31–0 (access to virtual model specific registers (MSRs)) and 63–32 (access to hypercalls) of the `HV_PARTITION_PRIVILEGE_MASK` structure respectively. At the time of this writing, Microsoft documents this structure and its members. In the context of this structure, a partition simply means a virtualized system (or VM).

The bitmask stored in ECX is a bit less utilized. The features assigned to each bit are documented in the TLFS. Currently, there are only 4 features listed:

- — used for performance monitoring regardless of power state
- — a component of Control-flow Enforcement Technology (CET)
- — used for measuring and monitoring CPU workloads
- — the guest has the capability to handle or manage exceptions that occur during its operation

Lastly, the bitmap stored in EDX is very similar to that in ECX. Microsoft documents what features these bits map to in the TLFS, but details regarding these features are scarce. I've found Alex Ionescu's Hyper-V Development Kit (hdv) to be a helpful starting point in many cases.

HviGetHardwareFeatures — 0x40000006

`HviGetHardwareFeatures()` works with a feature bitmask just like the previous function. First, though, it calls `HviGetHypervisorVendorAndMaxFunction()` and checks to make sure the value returned in EAX (the maximum supported leaf) is at least 0x40000006. If this check passes, it queries CPUID leaf 0x40000006 which returns a bitmask of hardware features in EAX. EBX, ECX, and EDX are reserved.

The hardware features whose states are described in this bitmask are documented in the TLFS, but just barely. More generalized documentation for features such as the ACPI Watchdog Action Table (WDAT) or High Precision Event Time (HPAT) are available from external sources. It may be worth noting that you can expect the SLAT bit (EAX[3]) to be set on virtually every hypervisor.

HviGetEnlightenmentInformation — 0x40000004

This function again calls `HviIsHypervisorMicrosoftCompatible()` to check if the interface identifier is “Hv#1” and queries leaf 0x40000004 if so. This leaf contains information for the guest operating system to ensure optimal performance under the hypervisor.

After execution of the cpuid instruction, EAX will contain yet another bitmask. This bitmask is where the hypervisor’s recommendations to the guest OS are primarily housed. Such recommendations include using a hypercall for an address switch rather than a `mov` to CR3 and whether or not the guest should use the enlightened VMCS interface.

The value stored in EBX is the recommended number of attempts the guest OS should retry on a spinlock failure before notifying the hypervisor. 0xFFFFFFFF means that the guest should not notify.

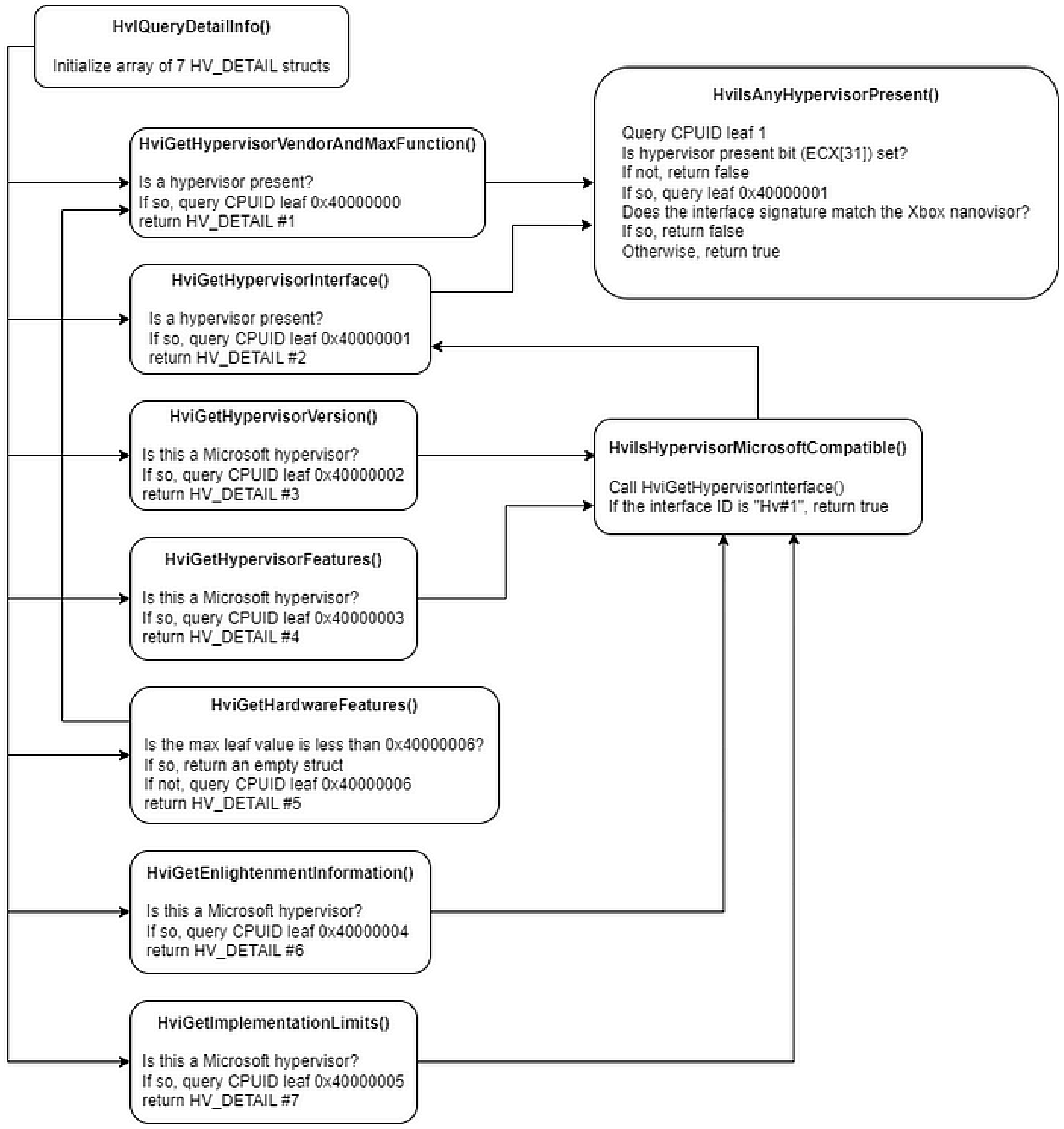
Lastly, bits 6–0 of ECX report the physical address width reported by the system’s physical processor. This is the exact same value as MAXPHYADDR, the physical address width. This appears to be the same thing as what is returned in EAX[7–0] when querying leaf 0x80000008, just evaluating 1 bit fewer...

HviGetImplementationLimits — 0x40000005

The last function called by `HvlQueryDetailInfo()`, `HviGetImplementationLimits()`, again checks that the interface ID is “Hv#1” before querying leaf 0x40000005. This leaf is one of the more simple ones as registers EAX, EBX, ECX only represent the maximum numbers of supported virtual processors, virtual processors, and physical interrupt vectors (used for interrupt remapping) respectively.

Recap

To help tie everything together, I've created a graphic to represent the internal flow of these functions and how they relate to one another.


HvlQueryDetailInfo()
Initialize array of 7 HV_DETAIL structs
HviGetHypervisorVendorAndMaxFunction()
Is a hypervisor present?
If so, query CPUID leaf 0x40000000
return HV_DETAIL #1
HviIsAnyHypervisorPresent()
Query CPUID leaf 1
Is hypervisor present bit (ECX[31]) set?
If not, return false
If so, query leaf 0x40000001
Does the interface signature match the Xbox nanovisor?
If so, return false
Otherwise, return true
HviGetHypervisorInterface()
Is a hypervisor present?
If so, query CPUID leaf 0x40000001
return HV_DETAIL #2
HviGetHypervisorVersion()
Is this a Microsoft hypervisor?
If so, query CPUID leaf 0x40000002
return HV_DETAIL #3
HviIsHypervisorMicrosoftCompatible()
Call HviGetHypervisorInterface()
If the interface ID is "Hv#1", return true
HviGetHypervisorFeatures()
Is this a Microsoft hypervisor?
If so, query CPUID leaf 0x40000003
return HV_DETAIL #4
HviGetHardwareFeatures()
Is the max leaf value is less than 0x40000006?
If so, return an empty struct
If not, query CPUID leaf 0x40000006
return HV_DETAIL #5
HviGetEnlightenmentInformation()
Is this a Microsoft hypervisor?
If so, query CPUID leaf 0x40000004
return HV_DETAIL #6
HviGetImplementationLimits()
Is this a Microsoft hypervisor?
If so, query CPUID leaf 0x40000005
return HV_DETAIL #7

But what about the other leaves?

You may have noticed that the maximum leaf value returned by `HviGetHypervisorVendorAndMaxFunction()` may be higher than what `HvlQueryDetailInfo()` queries, 0x40000006. Frankly, I'm not entirely sure why Microsoft chose to cut it here, especially when leaves 0x40000009 and 0x4000000A are documented in the TLFS. If you know the answer, please let me know. We'll just have to manually query those leaves ourselves for now 🤷‍♀️

Reimplementation

Due to the undocumented nature of `ntdll!NtQuerySystemInformation()` and the internal function, the structures with which we're working, and the data returned, I've created a basic C++ class to emulate the behavior of `nt!HvlQueryDetailInfo()`. This should allow anyone who needs the behavior and output of `NtQuerySystemInformation()` but isn't in love with using an undocumented function an alternative that includes a few quality-of-life improvements.

https://github.com/matterpreter/cpuid/

References & Additional Reading

Acknowledgements

Thank you to Satoshi Tanda and Sina Karvandi for their reviews of the draft of this post.