The Windows Registry Adventure #5: The regf file format

googleprojectzero.blogspot.com/2024/12/the-windows-registry-adventure-5-regf.html

Posted by Mateusz Jurczyk, Google Project Zero

As previously mentioned in the second installment of the blog post series ("A brief history of the feature"), the binary format used to encode registry hives from Windows NT 3.1 up to the modern Windows 11 is called regf. In a way, it is quite special, because it represents a registry subtree simultaneously on disk and in memory, as opposed to most other common file formats. Documents, images, videos, etc. are generally designed to store data efficiently on disk, and they are subsequently parsed to and from different in-memory representations whenever they are read or written. This seems only natural, as offline storage and RAM come with different constraints and requirements. On disk, it is important that the data is packed as tightly as possible, while in memory, easy and efficient random access is typically prioritized. The regf format aims to bypass the reparsing step – likely to optimize the memory/disk synchronization process – and reconcile the two types of data encodings into a single one that is both relatively compact and easy to operate on at the same time. This explains, for instance, why hives don't natively support compression (but the clients are of course free to store compressed data in the registry). This unique approach comes with its own set of challenges, and has been a contributing factor in a number of historical vulnerabilities.

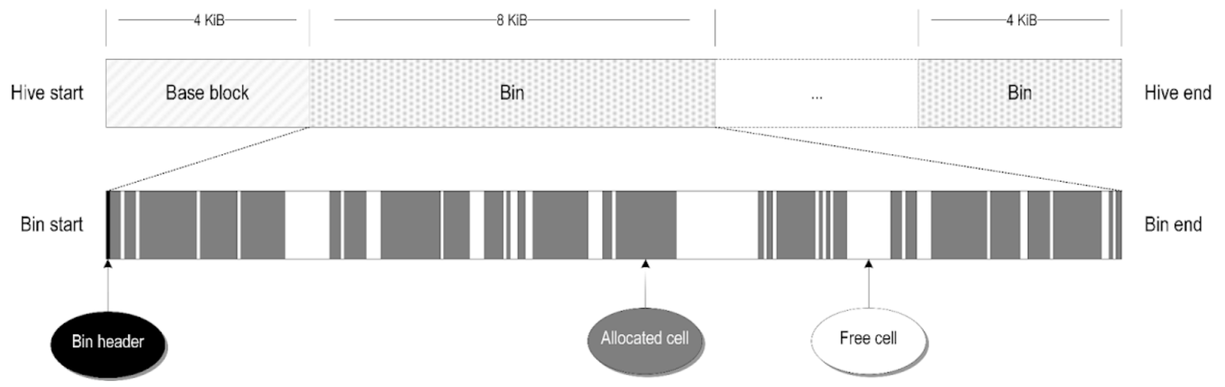
Throughout the 30 years of the format's existence, Microsoft has never released its official specification. However, the data layout of all of the building blocks making up a hive (file header, bin headers, cell structures) are effectively public through the PDB symbols for the Windows kernel image (ntoskrnl.exe) available on the Microsoft Symbol Server. Furthermore, the Windows Internals book series also includes a section that delves into the specifics of the regf format (named Hive structure). Lastly, forensics experts have long expressed interest in the format for analysis purposes, resulting in the creation of several unofficial specifications based on reverse engineering, experimentation and deduction. These sources have been listed in my earlier Learning resources blog post; the two most extensive specifications of this kind can be found here and here. The intent of this post is not to repeat the information compiled in the existing resources, but rather to highlight specific parts of the format that have major relevance to security, or provide some extra context where I found it missing. A deep understanding of the low-level regf format will prove invaluable in grasping many of the higher-level concepts in the registry, as well as the technical details of software bugs discussed in future blog posts.

The hive structure: header, bins and cells

On the lowest level, data in hives is organized in chunks of 4 KiB (0x1000 bytes), incidentally the size of a standard memory page in the x86 architecture. The first 4 KiB always correspond to the header (also called the base block), followed by one or more bins, each being a multiple of 4 KiB in length. The header specifies general information about the hive (signature, version, etc.), while bins are an abstraction layer designed to enable the fragmentation of hive mappings in virtual memory – more on that later.

Each bin starts with a 32-byte (0x20) header, followed by one or more cells that completely fill the bin. A cell is the smallest unit of data in a hive that has a specific purpose (e.g. describes a key, value, security descriptor, and so on). The data of a cell is preceded by a 32-bit integer specifying its size, which must be a multiple of eight (i.e. its three least significant bits are clear), and is either in the free or allocated state. A free (unused) cell is indicated by a positive size, and an allocated cell is indicated by a negative one. For example, a free cell of 32 bytes has a length marker of 0x00000020, while an active cell of 128 bytes has its size encoded as 0xFFFFFF80. This visibly demonstrates the hybrid on-disk / in-memory nature of the hive format as opposed to other classic formats, which don't intentionally leave large chunks of unused space in the files.

The overall file structure is illustrated in the diagram below:



In the Windows kernel, internal functions responsible for handling these low-level hive objects (base block, bins, cells) have names starting with "Hv", for example HvCheckHive, HvpAllocateBin or HvpViewMapCleanup. This part of the registry codebase is crucial as it forms the foundation of the registry logic, enabling the Configuration Manager to easily allocate, free, and access hive cells without concerning itself with the technical details of memory management. It is also a place with significant potential for optimizations, such as the incremental logging added in Windows 8.1, or section-based registry introduced in Windows 10 April 2018 Update (RS4). Both of these mechanisms are well described in the Windows Internals 7 (Part 2) book.

While integral to the correct functioning of the registry, hive management does not constitute a very large part of the overall registry-related codebase. In my analysis of the registry code growth shown in blog post #2, I counted 100,007 decompiled lines of code corresponding to this subsystem in Windows 11 kernel build 10.0.22621.2134. Out of these, only 10,407 or around 10.4% correspond to hive memory management. This is also reflected in my findings: out of the 52 CVEs assigned by Microsoft, only two of them were directly related to a Hv* function implementation – CVE-2022-37988, a logic bug in HvReallocateCell leading to memory corruption, and CVE-2024-43452, a double-fetch while loading hives from remote network shares. This is not to say that there aren't more bugs in this mechanism, but their quantity is likely proportional to its size relative to the rest of the registry-related code.

Let's now have a closer look at how each of the basic objects in the hive are encoded and what information they store, starting with the base block.

Base block

The base block is represented by a structure called _HBASE_BLOCK in the Windows Kernel, and its layout can be displayed in WinDbg:

```
0: kd> dt _HBASE_BLOCK

nt!_HBASE_BLOCK

   +0x000 Signature        : Uint4B

   +0x004 Sequence1        : Uint4B

   +0x008 Sequence2        : Uint4B

   +0x00c TimeStamp        : _LARGE_INTEGER

   +0x014 Major            : Uint4B

   +0x018 Minor            : Uint4B

   +0x01c Type             : Uint4B

   +0x020 Format           : Uint4B

   +0x024 RootCell         : Uint4B

   +0x028 Length           : Uint4B

   +0x02c Cluster          : Uint4B

   +0x030 FileName         : [64] UChar

   +0x070 RmId             : _GUID

   +0x080 LogId            : _GUID
```

```
+0x090 Flags            : Uint4B
+0x094 TmId             : _GUID
+0x0a4 GuidSignature    : Uint4B
+0x0a8 LastReorganizeTime : Uint8B
+0x0b0 Reserved1        : [83] Uint4B
+0x1fc CheckSum         : Uint4B
+0x200 Reserved2        : [882] Uint4B
+0xfc8 ThawTmId         : _GUID
+0xfd8 ThawRmId         : _GUID
+0xfe8 ThawLogId        : _GUID
+0xff8 BootType         : Uint4B
+0xffc BootRecover      : Uint4B
```

The first thing that stands out is the fact that even though the base block is 4096-bytes long, it only really stores around 236 bytes of meaningful data, and the rest (the Reserved1 and Reserved2 arrays) are filled with zeros. For a detailed description of each field, I encourage you to refer to the two unofficial regf specifications mentioned earlier. In the sections below, I share additional thoughts on the usage and relevance of some of the most interesting header members.

Sequence1, Sequence2

These 32-bit numbers are updated by the kernel during registry write operations to keep track of the consistency state of the hive. If the two values are equal during loading, the hive is in a "clean" state and doesn't require any kind of recovery. If they differ, this indicates that not all pending changes have been fully committed to the primary hive file, and additional modifications must be applied based on the accompanying .LOG/.LOG1/.LOG2 files. From a security point of view, manually controlling these fields may be useful in ensuring that the log recovery logic (HvAnalyzeLogFiles, HvpPerformLogFileRecovery and related functions) gets executed by the kernel. This is what I did when crafting the proof-of-concept files for CVE-2023-35386 and CVE-2023-38154.

Major, Minor

These are some of the most consequential fields in the header: they represent the major and minor version of the hive. The only valid major version is 1, while the minor version has been historically an integer between 0 and 6. Here is an overview of the different 1.x versions in existence:

Version	Year	Introduced in	New features
1.0	1992	Windows NT 3.1 Pre-Release	Initial format
1.1	1993	Windows NT 3.1	
1.2	1994	Windows NT 3.5	Predefined keys
1.3	1995	Windows NT 4.0	Fast leaves
1.4	2000	Windows Whistler Beta 1	Big value support
1.5	2001	Windows XP	Hash leaves
1.6	2016	Windows 10 Anniversary Update	Layered keys

The later versions draw extensively on the earlier ones both conceptually and in terms of the actual implementation – there are non-trivial portions of code in Windows NT 3.1 Beta that are used to this day in the latest Windows 11. But when it comes to pure binary compatibility, versions 1.0 to 1.2 differ too much from the newer ones and have long been considered obsolete. This leaves us with versions ≥ 1.3, which are all cross-compatible and can be used freely on the current systems. Within this group, version 1.4 was an intermediate step in the development of the format, observed only in beta releases of Windows XP (codenamed Whistler). The other three are all in active use, and can be found in a default installation of Windows 10 and 11:

- 1.3: encodes volatile hives (the root hive, HKLM\HARDWARE), the BCD hive (HKLM\BCD00000000), the user classes hives (HKU\<SID>_Classes), and some application hives (backed by settings.dat).
- 1.5: encodes a majority of the system hives in HKLM (SYSTEM, SOFTWARE, SECURITY, SAM, DRIVERS), all user hives (HKU\<SID>), and most application hives (backed by ActivationStore.dat).
- 1.6: encodes all differencing hives, i.e. hives used by processes running inside Application and Server Silos, mounted under \Registry\WC.

It is worth noting that the hive version is supposed to be indicative of the features used inside; for example, only hives with versions ≥1.4 should use big values (values longer than 1 MiB), only hives with versions ≥1.5 should use hash leaves, etc. However, this is not actually enforced when loading a hive, and newer features being used in older hives will work completely fine. This behavior may become a problem if any part of the registry code makes any assumptions about the structure of the hive based solely on its version. One example of such a vulnerability was CVE-2022-38037, caused by the fact that the CmpSplitLeaf kernel function determined the format of a subkey list based on the hive version and not the binary representation of the list itself. In general, when writing a registry-specific fuzzer, it might be a good idea to flip the minor version between 3-6 to increase the chances of hitting some interesting corner cases related to version handling.

As a last note, the version number is internally converted to a single 32-bit integer stored in the _HHIVE.Version structure member using the following formula: Minor+(Major*0x1000)-0x1000. In the typical case where the major version is 1, the last two components cancel each other out, e.g. version 1.5 becomes simply "5". This would be fine, if not for the fact that a major version of 0 is also allowed by HvpGetHiveHeader, in which case the minor version can be any value greater or equal to 3. Furthermore, if the kernel enters the header recovery path (because the hive header is corrupted and needs to be recovered from a .LOG file), then one can set the major/minor fields to completely arbitrary values and they will be accepted, as HvAnalyzeLogFiles doesn't perform the same strict checks that HvpGetHiveHeader does. Consequently, it becomes possible to spoof the version saved in _HHIVE.Version and have it take virtually any value in the 32-bit range, but I haven't found any security implications of this behavior, and I'm sharing it simply as a curiosity.

RootCell

This is the cell index (offset in the hive file) of the root key, which marks a starting point for the Configuration Manager to parse the hive tree. The root cell is special in many respects: it is the only one in a hive that doesn't have a parent, it cannot be deleted or renamed, its name is unused (it is instead referenced by the name of its mount point), and its security descriptor is treated as the head of the security descriptor linked list. While the RootCell member itself has not been directly involved in any bugs I am aware of, it is worth keeping its special properties in mind when doing registry security research.

Length

Specifies the cumulative size of all bins in the hive, i.e. its file size minus 4096 (the size of the header). It is limited to 0x7FFFE000, which reflects the ~2 GiB capacity of the hive stable storage (the part of the hive that resides on disk). Combined with another ~2 GiB of volatile space (in-memory hive data that gets erased on reboot), we get a total maximum size of around 4 GiB when both types of storage space are completely maxed out. Incidentally, that's the same range as a single 32-bit cell index can address.

Flags

There are currently only two supported hive flags: 0x1, which indicates whether there are any pending transactions involving the hive, and 0x2, which expresses whether the hive is differencing and contains layered keys or not. The latter flag is typically set when the hive version is 1.6.

LastReorganizeTime

In order to address the problem of accumulating fragmentation over time, Windows 8.1 introduced a new mechanism to both shrink and optimize hives during load called reorganization. It happens automatically if the last reorganization took place more than seven days ago and the fragmentation rate of the hive is greater than 1 MiB. Reorganization achieves its goals by starting off with an empty hive and copying all existing keys recursively, taking into account which ones have been used during boot, during system runtime, and not at all since the last reorganization. The end result is that the hive becomes more packed, thanks to the elimination of free cells taking up unnecessary space, and more efficient to operate on, because the "hot" keys are grouped closer together.

As the name suggests, the LastReorganizeTime member stores the timestamp of the last time a successful reorganization took place. From an attacker's perspective, it can be adjusted to control the behavior of the internal CmpReorganizeHive function and deterministically trigger the reorganization or skip it, depending on the desired end result. In addition to indicating a timestamp, the LastReorganizeTime field may also be equal to one of two special marker values: 0x1 to have the hive unconditionally reorganized on the next load, and 0x2 to clear the access bits on all the keys in the hive, i.e. reset the key usage information that has been collected so far.

CheckSum

The CheckSum field at offset 0x1FC stores the checksum of the first 508 bytes of the header (i.e. all data prior to this field), and is simply a 32-bit XOR of the header data treated as a series of 127 consecutive DWORDs. If the computed value is equal to 0xFFFFFFFF (-1), then the checksum is set to 0xFFFFFFFE (-2), and if the computed value is 0x0, then the checksum is 0x1. This means that 0 (all bits clear) and -1 (all bits set) are never valid checksum values. If you wish to examine the kernel implementation of the algorithm, you can find it in the internal HvpHeaderCheckSum function.

The checksum is particularly important when making changes to existing hives, either for experimentation or during fuzzing. If any data within the first 508 bytes of the file is modified, the checksum needs to be adjusted accordingly. Otherwise, the system will reject the file early in the loading process with the STATUS_REGISTRY_CORRUPT error code, and none of the deeper code paths will be exercised. Therefore, fixing up the checksum is the bare minimum a hive fuzzer should do to maximize its chances of success.

Other fields

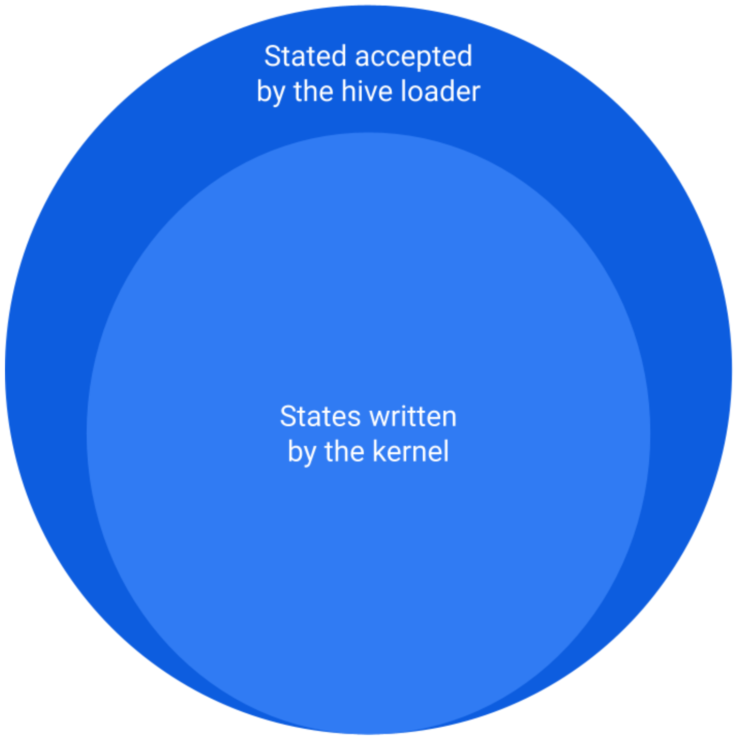
There are several other pieces of information in the header that carry some value, more so in the context of digital forensics and incident response than strictly low-level system security. For example, "Signature" identifies the file as a regf hive and may make it easier to identify the format in raw memory/disk dumps, while "TimeStamp" indicates the last time the hive has been written to, which can be critical for establishing a timeline of events during an investigation. Furthermore, the Offline Registry Library (offreg.dll) leaves further traces in the generated hive files: a 4-byte "OfRg" identifier at offset 0xB0 (nominally the Reserved1 field) and a serialization timestamp at offset 0x200 (nominally Reserved2). For more information about the meaning and usefulness of each part of the header, please refer to one of the unofficial format specifications.

Bins

Bins in registry hives are a simple organizational concept used to split a potentially large hive into smaller chunks that can be mapped in memory independently of each other. Each of them starts with a 32-byte _HBIN structure:

```
0: kd> dt _HBIN

nt!_HBIN

   +0x000 Signature        : Uint4B

   +0x004 FileOffset       : Uint4B

   +0x008 Size             : Uint4B

   +0x00c Reserved1        : [2] Uint4B

   +0x014 TimeStamp        : _LARGE_INTEGER

   +0x01c Spare            : Uint4B
```

The four meaningful fields here are the four-byte signature ("hbin"), offset of the bin in the file, size of the bin, and a timestamp. Among them, the signature is a constant, the file size is sanitized early in the hive process and effectively also a constant, and the timestamp is not security-relevant. This leaves us with the size as the most interesting part of the header. The only constraints for it is that it must be a multiple of 0x1000, and the sum of the offset and size must not exceed the total length of the hive (_HBASE_BLOCK.Length). At runtime, bins are allocated as the smallest 4 KiB-aligned regions that fit a cell of the requested size, so in practice, they typically end up being between 4-16 KiB in size, but they may organically be as long as 1 MiB. While longer bins cannot be produced by the Windows kernel, there is nothing preventing a specially crafted hive from being loaded in the system with a bin of ~2 GiB in size, the maximum length of a hive as a whole. This behavior doesn't seem to have any direct security implications, but more generally, it is a great example of how the hive states written by Windows are a strictly smaller subset of the set of states accepted as valid during loading:



Cells

Cells are the smallest unit of data in registry hives – they're continuous buffers of arbitrary lengths. They do not have a dedicated header structure like _HBASE_BLOCK or _HBIN, but instead, each cell simply consists of a signed 32-bit size marker followed by the cell's data. The size field is subject to the following constraints:

- A cell may be in one of two states – allocated and free – as indicated by the sign of the size value. Positive values are used for free cells, and negative ones for allocated cells.
- The size value accounts for the four bytes occupied by itself.

- The size value must be a multiple of 8 (i.e. have its three lowest bits set to zero). If a cell with size non-divisible by 8 is allocated at runtime, it is aligned up to the next multiple of 8, potentially leading to some unused padding bytes at the end of the cell.
- The sum of all consecutive cells in a bin must be equal to the length of the bin. In other words, the bin header followed by tightly packed cells (with no gaps) completely fill the bin space. If the hive loader detects that this is not the case, it forcefully fixes it by creating a single free cell spanning from the failing point up to the end of the bin. This invariant must subsequently hold for the entire time the hive is loaded in the system.

If cells remind you of heap allocations requested via malloc or HeapAlloc, it is not just your impression. There are many parallels to be drawn between hive cells and heap buffers: both can be allocated and freed, have arbitrary sizes and store a mixture of well-formatted structures and free-form user data. However, there are some significant differences too: heap implementations have evolved to include anti-exploitation mitigations like layout randomization, heap cookies for metadata protection, double-free detection and miscellaneous other consistency checks. On the other hand, hives have none of that: the allocation logic is fully deterministic and doesn't involve any randomness, there is no metadata protection, and generally little to no runtime checks. This is likely caused by the fact that heap chunks have been targets of memory corruption for many decades, whereas the registry was designed with the assumption that once loaded, the hive structure is always internally consistent and intra-hive memory corruption may never occur. This makes the exploitation of certain registry bugs particularly convenient and reliable, as I will demonstrate in future blog posts.

Like a typical memory allocator interface, cells have alloc, realloc, and free functions. Specifically, the internal routines responsible for these tasks in the Windows kernel are HvAllocateCell, HvReallocateCell and HvFreeCell, and reverse-engineering them allowed me to uncover some helpful insights. For instance, I have found that HvAllocateCell and HvReallocateCell reject allocation sizes larger than 1 MiB, and for requests above 16 KiB, they round the size up to the next power of two. Meanwhile, HvFreeCell performs coalescing of free cells, so there should never be two adjacent free cells in an organically created hive. These are some further examples of behavior that is guaranteed on output, but not enforced on input. This is a prevalent pattern in the Windows registry, and I found it useful to keep track of such primitives in my research, even if they didn't seem particularly useful at the time. Thanks to this, I have discovered at least three security bugs closely related to this phenomenon, including one in the interactions between HvReallocateCell and its callers (CVE-2022-37988).

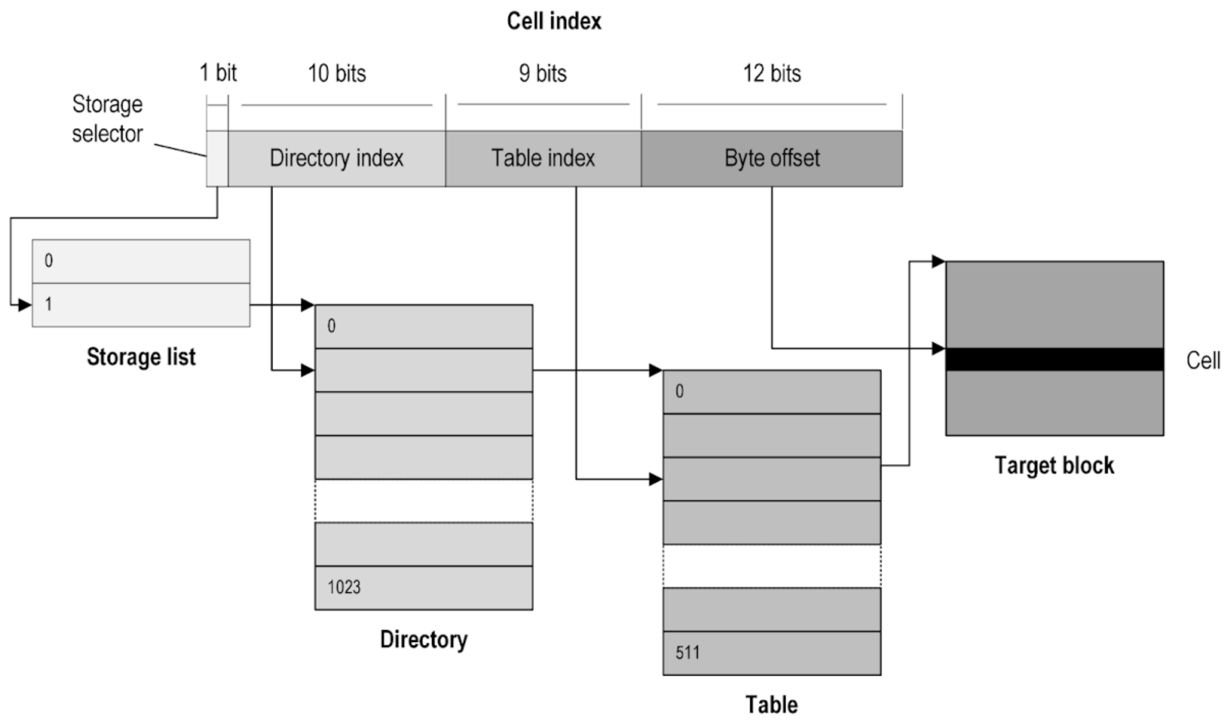
Cell indexes

If we equate cells to heap buffers in user-mode applications, then cell indexes would be pointers. Cells rely on these indexes to interrelate within the registry's complex structure. For example, keys reference security descriptors (to control access), their parent key (to navigate the hierarchy), and optionally the list of subkeys and list of values (to organize data). The list of values references specific value records, which in turn reference the actual data backing cells, and so on. This intricate web of relationships is no different from any semi-complex object in a C/C++ program, where pointers link various data structures.

On disk, cell indexes are nothing special: they are simply 32-bit offsets from the start of the hive data (after the 0x1000 byte header), which is a typical way of implementing cross-object references in most file formats. However, it's important to note that a cell index must point to the beginning of a cell (not inside it or in the bin header), and the cell must be in the allocated state – otherwise, the index is considered invalid. So when implementing a read-only regf parser operating on the hive as a contiguous memory block, translating cell indexes is as simple as adding them to the starting address of the hive in memory.

When a hive is loaded in Windows, the management of cell indexes becomes more complex. Hives at rest have a maximum size of 2 GiB, and all of their data is considered stable (persistently stored). On the other hand, an active hive also gains an additional 2 GiB of volatile storage, used for temporary keys and values that reside only in memory. These temporary entries exist only while the hive is loaded (or until the system is shut down) and can be created by calling RegCreateKeyEx with the REG_OPTION_VOLATILE flag, which designates the key as temporary. To distinguish between these two storage spaces in a cell index, the highest bit serves as an indicator: 0x0 for stable space and 0x1 for the volatile one, resulting in large index values (greater than 0x80000000) that readily identify volatile cells.

But an even bigger complication stems from the fact that hives can shrink and grow at runtime, so it is largely impractical to have them mapped as a single block of memory. To efficiently handle modifications to the registry, Windows maps hives in smaller chunks, which makes the previous method of translating cell indexes obsolete, and necessitates a more sophisticated solution. The answer to the problem are cell maps – pagetable-like structures that divide the 32-bit hive address space into smaller, nested layers, indexed by the respective 1, 10, 9, and 12-bit parts of the 32-bit cell index. Cell maps in the Windows kernel utilize a hierarchical structure consisting of storage arrays, directories, tables, and leaf entries, all defined within the ntoskrnl.exe PDB symbols (the relevant structures are _DUAL, _HMAP_DIRECTORY, _HMAP_TABLE and _HMAP_ENTRY). The layout of cell indexes and cell maps is illustrated in the diagram below, based on a similar diagram in the Windows Internals book, which itself draws from Mark Russinovich's 1999 article, Inside the Registry:



Cell indexes play a central role in core registry operations, such as creating, reading, updating, and deleting keys and values. The internal kernel function responsible for traversing the cell map and translating cell indexes into virtual addresses is HvpGetCellPaged. In normal conditions, the indexes stay within the bounds of the storage space size (_HHIVE.Storage[x].Length), so HvpGetCellPaged assumes their validity and doesn't perform any additional bounds checking. However, certain memory corruption vulnerabilities may allow attackers to manipulate these cell indexes at runtime. Crucially, I discovered that out-of-bounds cell indexes can serve as a powerful primitive for exploit development, enabling the construction of proof-of-concept exploits that achieve local elevation of privileges. I will elaborate further on this in future exploit-focused blog posts.

As a last note, the special marker of -1 (0xFFFFFFFF) is used to represent non-existent cells, and can be found in cell indexes pointing at optional data that doesn't exist – basically a hive equivalent of a NULL pointer. The internal name for the constant in the Windows kernel is HCELL_NIL, and under normal circumstances, it should never be passed directly to HvpGetCellPaged. Doing so without guaranteeing that the cell index is valid first would constitute a bug in the Windows kernel (for example, see CVE-2023-35357 or CVE-2023-35358).

Cell types

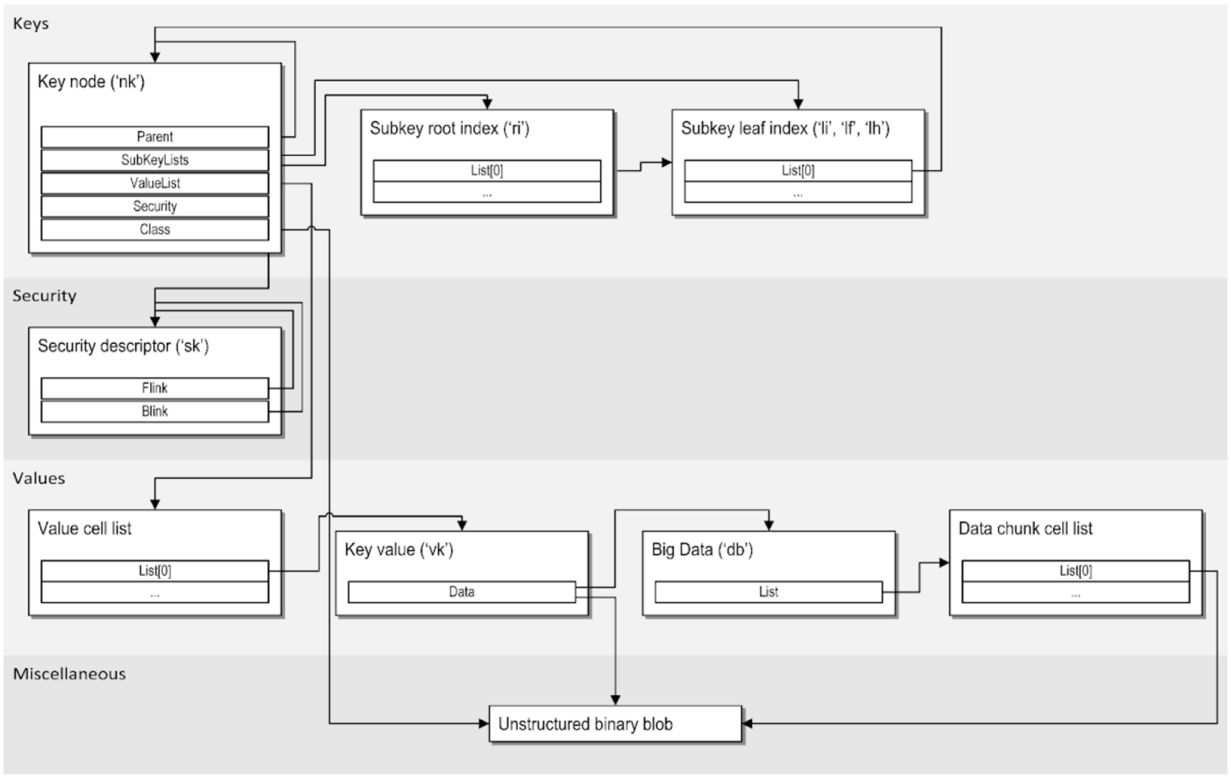
Now that we have familiarized ourselves with the low-level structure of hives that facilitates their efficient management in memory, let's go a little further and learn about the types of information stored in the cells. These are the objects that actually define the registry tree and all of its properties: keys, values, security descriptors, etc. The first subsection provides a general overview of the various cell types found within a hive and the relations between them. The second one goes into the intricate details of their format and usage within the Windows kernel, uncovering obscure implementation details rarely documented elsewhere.

Overview of cell types

Registry hives utilize only seven distinct cell types to represent the various data structures within the registry, as outlined below:

1. Key Node: Represents a single registry key and its associated metadata. It is defined by the _CM_KEY_NODE structure and contains references to other cells, including its parent key, security descriptor, class data (optional), and lists of subkeys (stable and volatile) and values (optional).
2. Subkey Index: A variable-length list of key node cell indexes, representing the subkeys of a specific key. For performance reasons, there are four variations of subkey indexes: index leaf, fast leaf, hash leaf, and root index. All are represented by the _CM_KEY_INDEX structure.
3. Security Descriptor: Defines access control information for one or more keys, specifically a security descriptor in a self-relative format. Represented by the _CM_KEY_SECURITY structure, it is the only cell type that can be referenced from multiple key nodes and is therefore reference-counted. It also contains links to the next and previous security descriptors in the hive.
4. Key Value: Defines a single value associated with a key, including its name, type, data length, and a reference to the cell containing the actual data. It is represented by the _CM_KEY_VALUE structure.
5. Big Data: Used to store value data exceeding 16,344 bytes (~16 KiB) in hive versions 1.4 and later. The data is divided into chunks of up to 16 KiB each, allowing for values approaching 1 GiB. The _CM_BIG_DATA structure represents this cell type, containing the number of chunks and a reference to the list of chunk cells.
6. Value List and Chunk List Cells: These cells are simple arrays of 32-bit cell indexes. They are used to store lists of values associated with a key and lists of chunks for large value data.
7. Data Cells: These cells store the raw data associated with keys and values. They hold the optional class data for a key, the complete data for small values (up to 1 MiB in older hives, ~16 KiB in newer hives), and the individual chunks of large values.

The diagram below illustrates the relationships between these cell types:



Deep dive into each cell type

Now that we know the general purpose of each cell type, it's a good time to dig a little deeper into each of them. This lets us explore both their implementation details, as well as the spirit behind these objects and how they interact with each other in a real-life environment. I have tried my best to avoid repeating the existing unofficial specifications and instead only focus on the security-relevant and sparsely documented aspects of the format, but if any redundant information makes it into this section, please bear with me. 🙂

Key nodes

As keys are the most important part of the registry, key nodes are the most important and complex of all cell types. When dumped in WinDbg, the layout of the _CM_KEY_NODE structure is as follows:

```
0: kd> dt _CM_KEY_NODE /r

nt!_CM_KEY_NODE

  +0x000 Signature        : Uint2B

  +0x002 Flags            : Uint2B

  +0x004 LastWriteTime    : _LARGE_INTEGER

  +0x00c AccessBits       : UChar
```

```
+0x00d LayerSemantics   : Pos 0, 2 Bits
+0x00d Spare1           : Pos 2, 5 Bits
+0x00d InheritClass     : Pos 7, 1 Bit
+0x00e Spare2           : Uint2B
+0x010 Parent           : Uint4B
+0x014 SubKeyCounts     : [2] Uint4B
+0x01c SubKeyLists      : [2] Uint4B
+0x024 ValueList        : _CHILD_LIST
   +0x000 Count            : Uint4B
   +0x004 List             : Uint4B
+0x01c ChildHiveReference : _CM_KEY_REFERENCE
   +0x000 KeyCell          : Uint4B
   +0x008 KeyHive          : Ptr64 _HHIVE
+0x02c Security         : Uint4B
+0x030 Class            : Uint4B
+0x034 MaxNameLen       : Pos 0, 16 Bits
+0x034 UserFlags        : Pos 16, 4 Bits
+0x034 VirtControlFlags : Pos 20, 4 Bits
+0x034 Debug            : Pos 24, 8 Bits
+0x038 MaxClassLen      : Uint4B
+0x03c MaxValueNameLen  : Uint4B
+0x040 MaxValueDataLen  : Uint4B
+0x044 WorkVar          : Uint4B
+0x048 NameLength       : Uint2B
+0x04a ClassLength      : Uint2B
```

```
   +0x04c Name             : [1] Wchar
```

In the following subsections, each member is discussed in more detail.

Signature

This field always stores the special value 0x6B6E, which translates to 'nk' when written in little-endian. It exists for informational purposes only, and isn't used for anything meaningful in the code after the initial sanitization during load.

Flags

This is a highly interesting and security-relevant field, as it indicates the role of the key in the hive, and clarifies how certain parts of the key node are formatted. The present and historical flags are presented in the table below together with their names and descriptions:

Mask	Name	Description
0x0001	KEY_VOLATILE	(Deprecated) The flag used to indicate that the key and all its subkeys were volatile, but it is obsolete now and hasn't been used in several decades. Information about the key stable/volatile state can be inferred from the highest bit of the key's cell index.
0x0002	KEY_HIVE_EXIT	Indicates that the key is the mount point of another registry hive. These special mount points are used to facilitate attaching new registry hives to the global registry view starting at \Registry in a live system. Exit nodes only ever exist in memory, so hives on disk mustn't have the flag set. More on the subject of mount points and exit nodes can be found in the next section, "Link nodes".
0x0004	KEY_HIVE_ENTRY	Indicates that the given key is the entry to a hive, or in other words, the root of a hive. The flag must be set on the root key of each hive, and mustn't be set on any other nested keys. A hive entry key cannot be a symbolic link (KEY_SYM_LINK mustn't be set).
0x0008	KEY_NO_DELETE	Indicates that the key cannot be deleted: any attempt to do so will return the error code STATUS_CANNOT_DELETE. This flag is always set on hive exit and hive entry keys, but is not allowed for any other keys.

0x0010	KEY_SYM_LINK	Indicates that the key is a symbolic link, which has been created by specifying the REG_OPTION_CREATE_LINK flag in the RegCreateKeyEx call. They are freely accessible and don't come with many restrictions: every key other than a hive exit/entry key can be a symbolic link. However, they are required to adhere to additional structural requirements: they may only contain up to one value, and that value must be of type REG_LINK (6), named "SymbolicLinkValue", and a maximum of 65534 bytes long (32767 wide characters).
0x0020	KEY_COMP_NAME	Indicates that the name of the key consists of ASCII characters only, and thus it has been "compressed" to fit two 8-bit characters in each of the 16-bit wide characters of _CM_KEY_NODE.Name. This optimization aims to save storage space and memory, especially as a great majority of keys have simple, alphanumeric names. This flag can be set on virtually every key in the registry, and indeed, it is by far the most commonly used one.

0x0040	KEY_PREDEF_HANDLE	(Deprecated) The flag used to indicate that the key was a "predefined-handle key", which was a special kind of a symbolic link. The name refers to Predefined Keys, a set of top-level keys such as HKLM or HKCU recognized by the Win32 API. Keys with the KEY_PREDEF_HANDLE flag set allowed the system to redirect certain keys to chosen 32-bit HKEY pseudo-handles, and were specifically introduced in Windows NT 3.5 in 1994 for the purpose of redirecting two system keys related to reading performance data through the registry: • HKLM\Software\Microsoft\Windows NT\CurrentVersion\Perflib\009 → HKEY_PERFORMANCE_TEXT • HKLM\Software\Microsoft\Windows NT\CurrentVersion\Perflib\CurrentLanguage → HKEY_PERFORMANCE_NLSTEXT Contrary to regular symbolic links, predefined keys re-purposed parts of the key node structure (specifically the value list length) to store the link destination, instead of using higher-level features of the format (such as the "SymbolicLinkValue" which is otherwise a perfectly normal value associated with a key). Such a change in semantics required a significant amount of special handling of predefined keys, which were not supposed to be operated on other than being opened. This, in turn, led to a number of security vulnerabilities related to the feature. For a detailed case study of one of them, CVE-2023-35633, see my Windows Registry Deja Vu: The Return of Confused Deputies talk from CONFidence 2024. As recently as 2023, all keys other than hive roots could be predefined keys, provided that they had been manually crafted in a binary controlled hive, because there was otherwise no supported way to create them via API. As a consequence of my reports, the feature was deprecated completely in July 2023 for Windows 10 1607+ and 11, and in December 2023 for older systems. At the time of this writing, the only two predefined keys left in existence are the original "009" and "CurrentLanguage" ones, and all other such keys are transparently converted to normal keys during hive load.

Furthermore, there are also three flags related to Registry Virtualization, which was introduced in Windows Vista and is supported up to and including Windows 11:

Mask	Name	Description
0x0080	VirtualSource	Indicates that the key has been subject to virtualization, i.e. that it has a counterpart in the virtual store subtree. It is typically set on keys inside HKLM\Software which have been attempted to be opened with write access by a program running as a non-administrator.
0x0100	VirtualTarget	Indicates that the key is a virtual replica of a key in a global system hive that has been subject to virtualization. It is typically set on keys inside HKU\<SID>_Classes\VirtualStore that have been created as a result of virtualization. It can only be set if VirtualStore (0x200) is set on the key, too.
0x0200	VirtualStore	Indicates that the key is part of the virtual store registry subtree, typically HKU\<SID>_Classes\VirtualStore and its subkeys. It means that new virtualization targets may be created inside the key, but it itself isn't necessarily a virtual key (unless the VirtualTarget flag is also set).

As we can see, the purpose of these flags is to keep track of the virtualization state of each key. Given that they express the internal state of the key and are intended to be modified by the kernel only, there doesn't seem to be a good reason to allow user-mode clients to modify the flags on demand. But in practice, unprivileged users have a lot of control over them: programs may arbitrarily set them in hives loaded from disk that they control (app hives and the user hive), and they may also set and clear them at runtime with the NtSetInformationKey(KeySetVirtualizationInformation) system call, as long as they are granted KEY_SET_VALUE access to the key. This makes it effectively possible to "spoof" virtual source/target/store keys, and opens up all of the registry virtualization code for potential abuse by unprivileged users. This has led to the discovery of multiple bugs directly related to virtualization: CVE-2015-0073 and CVE-2019-0881 by James Forshaw, and several more as part of my recent research.

LastWriteTime

This is yet another timestamp, in this case tracked on a key-granularity level. I assume it may be an interesting artifact for purposes of digital forensics, but otherwise it doesn't seem particularly security-relevant. One thing of note is that this information is very easy to query at runtime, as it is returned by the RegQueryInfoKey API, and is also a part of the output structures of numerous key information classes that can be queried via the NtQueryKey system call.

AccessBits

While theoretically an 8-bit field, this is effectively a 2-bit bitmask introduced in Windows 8 as part of the hive reorganization logic described earlier. It tracks the system phase(s) in which the key has been accessed: 0x0 if not accessed at all, 0x1 if accessed during boot, and 0x2 if accessed during normal system operation. This information is then used during reorganization to allocate key nodes with similar access bits close together.

LayerSemantics

This member is a 2-bit enum, used exclusively in hive version 1.6, which corresponds to differencing hives (also known as delta hives). Differencing hives are closely related to containerization support, and their purpose is to be overlaid on another hive in the system rather than being mounted as a standalone hive. For this reason, every key in a differencing hive is in one of four states, which indicate how the key should be interpreted in relation to the keys below it (i.e. the corresponding keys in lower-layer hives).

These four states are:

- Merge-Backed (0): the properties of the key are meant to be merged with the properties of the underlying keys in the key stack.
- Tombstone (1): the key is deleted at the current level, so none of the keys below it should be considered.
- Supersede-Local (2): the properties of the key fully supersede any state in the key stack below it: only values associated with that level (and any upper layers) are visible to the user.
- Supersede-Tree (3): same as Supersede-Local, but it applies to the key itself and recursively to all of its subkeys.

There is also an additional, implicit state called Merge-Unbacked, used to describe keys that don't exist in a hive at a given level, and so they simply fall through to the state represented by keys in the lower layers. Overall, layer semantics play a crucial role in the functionality of layered keys and differencing hives, and their correct handling in the registry implementation is paramount to system security and stability. Unfortunately, the feature is too complex to thoroughly discuss here, but there are some excellent resources on the subject: Microsoft's Containerized Configuration (US20170279678A1) patent, Maxim Suhanov's Containerized registry hives in Windows blog post, and the "Registry virtualization" section in Chapter 10 of the Windows Internals 7 (Part 2) book.

InheritClass

This bit is also related to layered keys, and it indicates whether the key inherits the class value from its counterparts lower in the key stack, or defines its own (or lack thereof).

Parent

The field identifies the key node that acts as this key's parent within the registry's hierarchical structure. Except for root keys, which exist at the topmost level of a hive, every key must have a valid Parent field. This index plays a vital role in navigating the registry and modifying key relationships. For example, it's essential for determining a key's full path or ensuring correct alphabetical order when renaming a key within its parent's subkey list.

SubKeyCounts

This two-element array of DWORDs stores the number of the key's stable and volatile subkeys, respectively. Even though the integers are 32 bits wide, the actual number of subkeys is limited by the upper bound of all keys in a hive in a specific storage space, which is roughly 2 GiB (storage space size) ÷ 84 bytes (minimum key node size) ≈ 25.5 million keys.

The data in this field is somewhat redundant, as the same information is also stored in the subkey indexes themselves. Nevertheless, the cached numbers stored directly in the key node make it possible to efficiently query the numbers of subkeys with API such as RegQueryInfoKey. The kernel does its best to keep the two copies of the information in sync, and any discrepancies between them may lead to memory corruption vulnerabilities.

SubKeyLists

This is another two-element array, which complements the previous SubKeyCounts member by providing cell indexes to the corresponding subkey lists for each storage type. The format of these lists is discussed in detail in the "Subkey indexes" section below; for now, it's only important to know that if SubKeyCounts[x] > 0, then SubKeyLists[x] is expected to be a valid cell index, otherwise it should be equal to HCELL_NIL (-1). Furthermore, because the volatile space is a strictly in-memory concept that doesn't exist on disk, newly loaded hives are always expected to have SubKeyCounts[1] set to 0 and SubKeyLists[1] set to HCELL_NIL.

ValueList

This is a structure of type _CHILD_LIST, which consists of two 32-bit integers: the number of values associated with the key, and a cell index of the actual value list. Here, there is no distinction between stable and volatile values: for any given key, the values always inherit the storage type of the key, so either all of them are stable, or all of them are volatile. Similarly to subkey lists, though, if ValueList.Count is 0, then ValueList.List must be HCELL_NIL.

As mentioned earlier, this field also had a second meaning if the key was a predefined key: in that case, ValueList.Count contained an arbitrary value with the highest bit set, which indicated the top-level HKEY to redirect to, and ValueList.List was completely unused and could contain arbitrary data. As you can imagine, whenever an internal system function started to use such a value list with the assumption it was a normal key, it would operate on an inadequately huge count and an invalid cell index, wrecking havoc in the kernel. Thankfully, this is no longer a possibility due to the deprecation of predefined keys in 2023.

ChildHiveReference

You may have noticed that ChildHiveReference is part of a union, as it resides at the same offset as the SubKeyLists member (offset 0x1C). It is a special object that is used to implement hive mounting under the \Registry tree, and is unique to keys that have the KEY_HIVE_EXIT flag set (i.e. exit nodes). It is only ever used in memory, and is therefore not applicable to regular hives stored on disk. Its two fields specify the root key of the mounted hive, as a pair of a kernel pointer to the _HHIVE descriptor structure and the cell index of the root key. This breaks the fundamental invariant that hives are self contained and don't store any virtual address pointers, only cell indexes. It is the only exception to the rule, and only because it is a necessary hack/workaround to implement a feature that hives normally don't support: attaching one hive to another in the global system view.

The field and its usage are discussed in more detail in the "Link nodes" section below.

Security

This is the cell index of the security descriptor cell corresponding to the key. It is a mandatory field for every type of key in the registry (symbolic links, previously predefined keys etc.), with the only exception being system-managed exit nodes. For every key that has an invalid security descriptor during hive load (e.g. set to HCELL_NIL or just an invalid cell index), it is automatically fixed up to inherit the security descriptor of its parent key. If the root key of a hive has invalid security, the whole hive is rejected with the STATUS_REGISTRY_CORRUPT error code.

The security descriptor cell always has the same storage type as the key(s) that it is associated with. So for example, if there are two keys in a hive with the same security properties, one in the stable and the other in the volatile space, then they will reference two different stable/volatile security cells with equivalent data.

For obvious reasons, the correct handling of this field is crucial to overall system security. In the course of my research, I have discovered 9 vulnerabilities directly involving the handling of security descriptors, and a further 4 reported to Microsoft outside of the tracker (WinRegLowSeverityBugs #1, #10, #13, #20). They generally didn't have much to do with the _CM_KEY_NODE.Security field specifically, but rather the formatting of the security cells and higher-level logic related to them:

- Binary formatting of the SECURITY_DESCRIPTOR_RELATIVE structure
- Maintaining the consistency of the doubly-linked list of security descriptors in the hive
- Reference counting security descriptors when operating on keys
- Enforcing proper access checks when opening and creating keys

Overall, this is probably the most interesting field in the structure from a security research perspective.

Class and ClassLength

In technical terms, a key class is an optional, immutable blob of 1-65535 bytes associated with a key. It can only be set once, during the creation of a key, through the lpClass argument of the RegCreateKeyExW API (or the equivalent Class parameter of the NtCreateKey system call). It can be then queried with functions such as RegQueryInfoKey, but cannot be modified without deleting and re-creating the key. If the class exists, then the ClassLength field is set accordingly, and Class is a cell index that points to its backing buffer. Otherwise, ClassLength is set to 0 and Class is HCELL_NIL (-1).

Conceptually, a class can be viewed as an extra, hidden value of a key, existing alongside the normal value list. It is not displayed anywhere in the Regedit GUI, but if it exists for a given key, it can be retrieved by using the "Export" option in Regedit to save the key to a .txt file, which also exports the class data. It has existed since the earliest version 1.0 of the regf format – perhaps as a way to store the "type" of a key similar to how every value has a defined type. Today, it seems to be a mostly obsolete mechanism that doesn't see much use; even Raymond Chen wrote in his What is the terminology for describing the various parts of the registry? blog in 2009:

> Bonus chatter: There's also this thing called a class. I have no idea what it's for, so don't ask.

When I ran a quick scan of the Windows 11 registry, I found the following unique strings being used at least once as a key class:

- "DynDRootClass"
- "GenericClass"
- "Network ComputerName"
- "REG_SZ"
- "Shell"

The Windows NT Registry File (REGF) format specification lists several other values that have been observed in the past, such as "activeds.dll ", "Cygwin", "OS2SS" or "TCPMon". It is worth noting that the class was also used to store the encryption keys for the now-deprecated SAM database encryption mechanism known as SysKey. Overall, due to its simplistic nature, key classes are not particularly security-relevant, but may be of interest in the context of obfuscation and hiding data, as they are easily accessible and yet a largely overlooked part of the registry.

MaxNameLen, MaxClassLen, MaxValueNameLen and MaxValueDataLen

These four fields store cached information about the maximum lengths of several properties of the key or its subkeys:

- MaxNameLen: the maximum length of a subkey's name,
- MaxClassLen: the maximum length of a subkey's class information,

- MaxValueNameLen: the maximum length of a value name associated with the key,
- MaxValueDataLen: the maximum length of a value data associated with the key.

The presumed purpose of these members is to facilitate a quick lookup of the per-key limits, such that when a client application wants to enumerate/query subkeys or values, it can simply allocate a single buffer guaranteed to accommodate every possible key name, value name, etc. And so, their exact values can be retrieved with the RegQueryInfoKey API via the lpcbMaxSubKeyLen, lpcbMaxClassLen, lpcbMaxValueNameLen and lpcbMaxValueLen arguments.

Although querying these limits seems convenient, there are some caveats that are important to keep in mind:

- The fields are intended to represent the lower bound of the number of bytes required to store the given property, but not necessarily to be optimal (i.e. to be the smallest sufficient length). For example, when a key with formerly the longest name is deleted, the MaxNameLen field of the parent is not updated with the value of the second-largest length, as that would require the lengthy process of iterating through all of the subkeys again. Therefore, relying on those values may incur some unwanted memory overhead.
- When operating on registry keys that are globally visible in the registry tree, it is possible that a race condition with another application causes one of the maxima to change in between the RegQueryInfoKey call and the actual data query. To address this, applications should include fallback logic to allocate more memory in the rare case when the obtained maximum proves insufficient.
- To add to the previous point, after having reverse-engineered and reviewed most of the Configuration Manager code, it is my instinct that these fields continue to be supported throughout the development of new registry features (e.g. differencing hives), but it is mostly on a best-effort basis. For example, during hive load, only MaxValueNameLen and MaxValueDataLen are enforced to have the correct values, while MaxNameLen and MaxClassLen remain unchecked. For this reason, I would personally not rely on the consistency of those values for the security of any client code, and would treat them more as a guidance/supplementary information than the sole source of truth about the key limits.

UserFlags

This is a field whose name, offset and function (so basically every aspect) has been subject to change over the years. Its current form has existed since Windows Vista, and occupies bits 20-23 of MaxNameLen, which had been previously a 32-bit integer, but was later reduced to 16 bits to make room for these extra flags. In theory, its name may suggest that this member is meant to store user-defined data, but in practice, Microsoft developers quickly found their own use for the bitmask: storing flags related to the Registry Reflection mechanism for providing interoperability between 32-bit and 64-bit applications.

You can read more about the meaning of each specific flag here, but in short, this was where reflection-specific configuration was internally saved by API functions such as RegEnableReflectionKey and RegDisableReflectionKey, and retrieved by RegQueryReflectionKey.

However, this specific use seems to have been short-lived, as Registry Reflection was soon deprecated in Windows 7. Since then, it could indeed be considered as four extra bits of user-controlled storage per key, accessible for reading via NtQueryKey(KeyFlagsInformation) and for writing via NtSetInformationKey(KeyWow64FlagsInformation). Beyond being interesting for historical reasons, the field doesn't play any important role in security.

VirtControlFlags

This field is another one introduced around Windows XP SP3 / Windows Vista that took over some of the space from MaxNameLen. It is related to Registry Virtualization and takes up four bits in the _CM_KEY_NODE structure definition, but there are only three flags that it can really store:

Mask	Name	Description
0x1	REG_KEY_DONT_VIRTUALIZE	Disables virtualization for the specific key.
0x2	REG_KEY_DONT_SILENT_FAIL	Prevents the system from re-opening a virtualized key with MAXIMUM_ACCESS if the initial Open operation with the desired access rights has failed.
0x4	REG_KEY_RECURSE_FLAG	Causes new subkeys of the key to inherit its virtualization-related configuration.

The flags are not sanitized in any way during hive load and so may be set to arbitrary values. They can also be modified programmatically by using the NtSetInformationKey(KeySetVirtualizationInformation) system call, or even from the Windows command line, by using the REG FLAGS command:

```
C:\>reg flags /?

REG FLAGS KeyName [QUERY |

                SET [DONT_VIRTUALIZE] [DONT_SILENT_FAIL] [RECURSE_FLAG]]

               [/reg:32 | /reg:64]

 Keyname    "HKLM\Software"[\SubKey] (Restricted to these keys on
     local machine only).

  SubKey   The full name of a registry key under HKLM\Software.

 DONT_VIRTUALIZE DONT_SILENT_FAIL RECURSE_FLAG

  Used with SET; flags specified on the command line will be set,

     while those not specified will be cleared.

 /reg:32  Specifies the key should be accessed using the 32-bit registry view.

 /reg:64  Specifies the key should be accessed using the 64-bit registry view.
```

More information about these flags can be found in the documentation of the ORSetVirtualFlags API function, a part of the Offline Registry Library. In the context of registry security research, I haven't found them particularly interesting – the other virtualization-related flags in the "Flags" field have proved to be much more useful in that regard.

Debug

In Debug/Checked builds of Windows, it used to be possible to have the kernel trigger a breakpoint when performing a specific operation on a specific registry key. To enable the option, an administrator would have to set the HKLM\System\CurrentControlSet\Control\Session Manager\Configuration Manager\RegDebugBreaksEnabled value to 1, which would propagate to the global kernel CmpRegDebugBreakEnabled variable. Then, the "Debug" field of each key would store a bitmask indicating which subset of eight possible operations should be interrupted for the given key:

Mask	Name
0x01	BREAK_ON_OPEN
0x02	BREAK_ON_DELETE
0x04	BREAK_ON_SECURITY_CHANGE
0x08	BREAK_ON_CREATE_SUBKEY
0x10	BREAK_ON_DELETE_SUBKEY

0x20	BREAK_ON_SET_VALUE
0x40	BREAK_ON_DELETE_VALUE
0x80	BREAK_ON_KEY_VIRTUALIZE

Whenever a breakpoint was triggered by this mechanism, the kernel would also print out a corresponding message for the attached debugger, for instance:

DbgPrint("\n\n Current process is deleting a key tagged as BREAK ON DELETE");

DbgPrint(" or deleting a subkey under a key tagged as BREAK_ON_DELETE_SUBKEY\n");

DbgPrint("\nPlease type the following in the debugger window: !reg kcb %p\n\n\n", Kcb);

Now that the Debug/Checked builds have been discontinued – or at least not released publicly anymore for the latest versions of Windows 10/11 – the "Debug" field is just an unused byte in the key node structure.

WorkVar

According to an unofficial format specification, WorkVar used to be an internal-use member meant to be only ever accessed by the kernel in order to optimize key lookups. The last version of Windows where WorkVar was still in active use was Windows 2000; since Windows XP, it has simply been another four bytes of unused memory in the key node data layout.

NameLength and Name

The combination of these two fields specifies the name of the key: NameLength indicates the length of the string in bytes, and Name is an inline, variable-length buffer at the end of the structure that stores the name itself. There are a number of considerations and consistency requirements related to registry key names, enforced when loading a hive and later at runtime:

- Compression: If the KEY_COMP_NAME (0x20) flag is clear in _CM_KEY_NODE.Flags, the name is formatted as a wide string of 16-bit characters. If it is set, which is the common scenario, then "Name" represents a more tightly packed ASCII string of 8-bit characters. Considering that a majority of keys in the registry are alphanumeric, this optimization saves a non-trivial amount of memory and disk space. It is interesting to note that it is still possible to load a hive with a non-optimally formatted key name (non-compressed ASCII string), but such a key node would never be generated by Windows itself.

- Length: The key name mustn't be empty (i.e. it should be at least one character long), and it cannot exceed 256 characters in length (even though Registry element size limits incorrectly claims that the limit is 255). The NameLength field value is expressed in bytes, so it must be between 1-256 for compressed names, and 2-512 for wide strings (and divisible by two). Up until October 2022, this limit was not correctly enforced, making it possible to load hives with key names up to 1040 characters, which would then be mishandled or outright rejected by other parts of the registry code.
- Charset: All characters in the 0x0000 – 0xFFFF range are allowed in a key name with the exception of backslash ('\', 0x005C). The backslash is singled out because it plays a special role in the registry, separating distinct elements of the registry paths. Since the kernel must always be able to distinguish parts of key names from the separator, a decision was made to exclude this one character from the key name charset, similar to how backslashes are not allowed in file names. Furthermore, there is a second minor requirement that the key name must not start with a null character, but it may be present at any other position in the name (this only started to be properly enforced in NtRenameKey after the fix for CVE-2024-26178 in March 2024). Overall, this means that key names aren't truly textual strings in the conventional sense of the word: they don't use a terminator, and may contain all sorts of non-printable characters. It would be more appropriate to think of them as binary blobs used to reference registry keys, which doesn't have any consequences for the kernel, as it universally uses the UNICODE_STRING structure that includes both the length and the backing buffer of the string anyway. But if a potentially malicious program were to create a key with an unusual name (e.g. including a null character), it could prove difficult for an administrator to operate on it with the built-in registry utilities (reg.exe, Regedit), or even with third-party tools that use the high-level API (such as RegOpenKeyEx). In such cases, it might be required to use specialized tools that interact with the Windows registry directly through the system call interface as the only way to examine/modify such keys.
- Uniqueness: One of the most important invariants of the Windows registry implementation is the uniqueness of key names: there may be only one key with a specific path, or in other words, for every key, there mustn't be any duplicates in the list of its subkeys. Given that registry key names are case-insensitive, any two names are always compared in their uppercase form to determine if they are equal or not. This uniqueness requirement is enforced both during hive load and subsequent operations, and failure to do it correctly could lead to both logic bugs and memory corruption. For some examples of the potential outcomes of allowing duplicate key names in registry, see Maxim Suhanov's The uppercased hell blog or my CVE-2023-21748 / CVE-2023-23420 bug reports.

Another intriguing aspect of the key names are the names associated with the root keys of default system hives. In general, every registry key in Windows is referenced by its name specified in the key node, except for root keys, which are known by the name of their mount points. As a result, the "real" underlying names of root keys are never visible to users or applications, but they are nevertheless present in the hive file as a mandatory part of every key node, and could be potentially used to learn something about how these fundamental system hives (SOFTWARE, SYSTEM etc.) are generated.

I have examined hives from various Windows versions ranging from Windows NT 3.1 to Windows 11, and arrived at the following list of per-version root key names:

Version	Root key name
NT 3.1 - NT 4.0	Same as the hive name (e.g., "SYSTEM")
2000 - XP	$$$PROTO.HIV
Vista - 7	CMI-CreateHive{RANDOM GUID}
8	CsiTool-CreateHive-{00000000-0000-0000-0000-000000000000}
10 - 11	ROOT

In early NT versions, the root key name simply mirrored the hive's file name. In Windows 2000 and XP, the name stemmed from the fact that system hives were created during system installation by temporarily creating the tree root under \Registry\Machine\SYSTEM\$$$PROTO.HIV, pre-initializing it with the default data for the given hive, and saving it to a file with an API like RegSaveKeyEx.

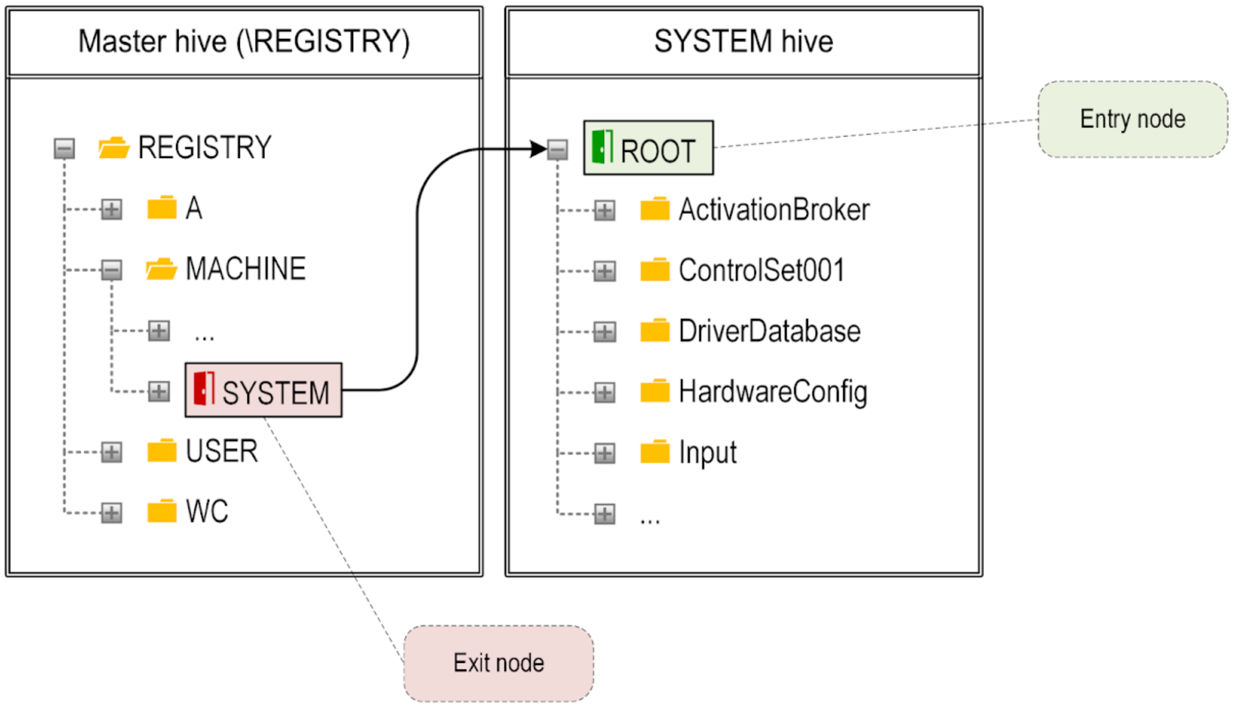
In Windows 10 and 11, the name is simply "ROOT", which, along with the "OfRg" magic bytes at offset 0xB0 in the file header, hints that the hives are created with the Offline Registry Library. This leaves versions between Windows Vista and Windows 8 as the big unknown: neither "CMI-CreateHive" nor "CsiTool-CreateHive" sound particularly familiar, and I haven't been able to find any information about them in any public resources. It is probably safe to assume that these strings are indicative of some internal Microsoft tooling that was used to generate hives for these systems, but not much is known beyond it. Nevertheless, I find it fascinating that such little tidbits of information can be found in obscure corners of file formats. You never know when some other missing part of the puzzle becomes known publicly, making it possible to finally connect the dots and see the bigger picture, sometimes years or decades after the initial release of the software.

Link nodes

As mentioned above, link nodes are a special type of key node designed to facilitate the mounting of arbitrary hives from disk into the global registry view. They are managed by the Windows kernel and only ever exist in memory. They are represented by the _CM_KEY_NODE structure, but with the following differences compared to regular keys:

- The Signature field is set to 0x6B6C ('lk') instead of 0x6B6E ('nk'),
- The KEY_HIVE_EXIT (0x0002) flag is set in Flags,
- The key doesn't have any of the standard key properties, such as the security descriptor, class, subkeys or values. The only cell reference it contains is to its parent cell, which is one of \Registry\A, \Registry\Machine, \Registry\User or \Registry\WC.
- Instead of the SubKeyLists member at offset 0x1C, the link node uses the ChildHiveReference field of type _CM_KEY_REFERENCE, which stores a kernel-mode pointer to the destination hive descriptor (_HHIVE*), and the cell index of the root key within that hive.

So, whenever you see a hive root key (e.g. any key within HKLM or HKCU), you are actually looking at a pair of a link node (also known as exit node) + root key (a.k.a. entry node – these terms are used interchangeably). The mount point assumes the key name of the link node (so that it is easily enumerable with the existing kernel logic), and all of the characteristics of the entry node. This is illustrated in the following diagram, where the key marked in red is the link node of the SYSTEM hive, and the green one is the root key:



The existence of link nodes seems to be very little known and scarcely documented in public resources, which is likely caused by the fact that the Windows kernel makes them virtually invisible, and not just for users and high-level API clients, but even for administrators and kernel driver developers. The way the registry tree traversing code is structured, whenever it encounters a link node, it always makes sure to skip over it and reference the corresponding entry node. This means that it is impossible to open or otherwise observe the link node itself from the context of user-mode, but if we put in some effort, we should be able to see it in WinDbg attached as a kernel debugger. We can approach the link node from two sides: either try to find it top-down starting from the master hive, or by locating a key in a mounted hive and traversing the registry tree upwards.

In this post, we will proceed with the first idea and enumerate the keys within \Registry\Machine (i.e. HKLM):

```
0: kd> !reg querykey \registry\machine

Found KCB = ffff800f88ad96e0 :: \REGISTRY\MACHINE

Hive        ffff800f88a88000

KeyNode     ffff800f88ada16c

[SubKeyAddr]        [SubKeyName]

ffff800f88ada44c    BCD00000000

ffff800f88ada3cc    HARDWARE

ffff800f88ada59c    SAM

ffff800f88ada504    SECURITY

ffff800f88ada374    SOFTWARE

ffff800f88ada31c    SYSTEM

 Use '!reg keyinfo ffff800f88a88000 <SubKeyAddr>' to dump the subkey details

[ValueType]        [ValueName]                 [ValueData]

REG_DWORD          ServiceLastKnownStatus      2
```

Here, we can see all the system hive mount points together with their corresponding link node addresses. In case of normal, stable keys, these would be user-mode addresses within the address space of the Registry process, but since the master hive is a volatile one, all of its structures are stored on the kernel pools. We can then use a command such as !reg knode to query any of the specific subkeys, e.g. SYSTEM:

```
0: kd> !reg knode ffff800f88ada31c
```

```
Signature: CM_LINK_NODE_SIGNATURE (kl)

Name                : SYSTEM

ParentCell          : 0x168

Security            : 0xffffffff [cell index]

Class               : 0xffffffff [cell index]

Flags               : 0x2a

MaxNameLen          : 0x0

MaxClassLen         : 0x0

MaxValueNameLen     : 0x0

MaxValueDataLen     : 0x0

LastWriteTime       : 0x 1db2b94:0xe031a530

SubKeyCount[Stable  ]: 0x0

SubKeyLists[Stable  ]: 0x20

SubKeyCount[Volatile]: 0x0

SubKeyLists[Volatile]: 0xffffffff

ValueList.Count     : 0x88a8e000

ValueList.List      : 0xffff800f
```

As expected, the key node has the special link node signature ('kl'), and the 0x2 flag set within the 0x2a Flags bitmask (the other two flags set are KEY_NO_DELETE and KEY_COMP_NAME). The command gets a little confused, because it expects to operate on a regular key node and display its subkey/value counts and lists, but as mentioned above, this space is taken up by the _CM_KEY_REFERENCE structure in the link node. If we line up the offsets correctly, we can decode that the exit node points at cell index 0x20 in hive 0xffff800f88a8e000, which is consistent with the outcome of displaying the structure data directly:

```
0: kd> dx -id 0,0,ffffbd044acf6040 -r1 (*
((ntkrnlmp!_CM_KEY_REFERENCE *)0xffff800f88ada338))
```

```
(*((ntkrnlmp!_CM_KEY_REFERENCE *)0xffff800f88ada338))
 [Type: _CM_KEY_REFERENCE]

    [+0x000] KeyCell          : 0x20 [Type: unsigned long]

    [+0x008] KeyHive          : 0xffff800f88a8e000 [Type: _HHIVE *]
```

We can now translate this information into the cell's virtual address, and take a peek into it with !reg knode and !reg keyinfo:

```
0: kd> !reg cellindex 0xffff800f88a8e000 0x20

Map = ffff800f88adc000 Type = 0 Table = 0 Block = 0 Offset = 20

MapTable     = ffff800f88ade000

MapEntry     = ffff800f88ade000

BinAddress = ffff800f896e8009, BlockOffset = 0000000000000000

BlockAddress = ffff800f896e8000

pcell:  ffff800f896e8024

0: kd> !reg knode ffff800f896e8024

Signature: CM_KEY_NODE_SIGNATURE (kn)

Name                 : ROOT

ParentCell           : 0x318

Security             : 0x78 [cell index]

Class                : 0xffffffff [cell index]

Flags                : 0x2c

MaxNameLen           : 0x26

MaxClassLen          : 0x0

MaxValueNameLen      : 0x0

MaxValueDataLen      : 0x0

LastWriteTime        : 0x 1db2b94:0xe031a530
```

```
0: kd> !reg keyinfo 0xffff800f88a8e000 ffff800f896e8024

KeyPath         \REGISTRY\MACHINE\SYSTEM

[SubKeyAddr]         [SubKeyName]

ffff800f896e8174     ActivationBroker

ffff800f896e964c     ControlSet001

ffff800f89f0e8a4     DriverDatabase

ffff800f89f999c4     HardwareConfig

ffff800f89f9a314     Input

ffff800f89f9a3dc     Keyboard Layout

ffff800f89f9a43c     Maps

ffff800f89f9a674     MountedDevices

ffff800f89f9ab64     ResourceManager

ffff800f89f9abc4     ResourcePolicyStore

ffff800f89f9ac2c     RNG

ffff800f89f9addc     Select

ffff800f89f9aed4     Setup

ffff800f89f9b7d4     Software

ffff800f89f9d1f4     State

ffff800f89f9d24c     WaaS

ffff800f89fabc8c     WPA

[SubKeyAddr]         [VolatileSubKeyName]

ffff800f88b91024     CurrentControlSet

 Use '!reg keyinfo ffff800f88a8e000 <SubKeyAddr>' to dump the subkey details

[ValueType]         [ValueName]                  [ValueData]

 Key has no Values
```

We have indeed ended up at the root key of the SYSTEM hive, which has a standard key node signature ('nk'), the predefined "ROOT" name, a valid security descriptor, a list of subkeys, and so on.

Overall, link nodes are an interesting implementation detail of the registry that are worth keeping in mind. However, considering their relative simplicity and the fact that they are hidden away even from very low-level mechanisms like Registry Callbacks, they are of limited significance to system security. The lone vulnerability I found related to them, CVE-2023-21747, resulted in a use-after-free due to improper cleanup of the exit node when faced with an out-of-memory condition.

Subkey indexes

Operations performed on subkey lists are some of the most common ones – they are involved whenever a key is opened, created, deleted, renamed or enumerated, which constitutes a majority of actions involving the registry at runtime. It is for this reason that subkey lists have seen the most evolution throughout the subsequent versions of the regf format. As the interface was getting adopted by more and more applications in Windows NT and later systems, Microsoft developers could collect data on the typical usage patterns and devise adequate optimizations to speed these operations up. In this section, we will have a deeper look into how subkey indexes are formatted in the hives, and how the different types of operations affect them.

By way of introduction, subkey indexes are data structures storing lists of descendant keys relative to a parent key, referenced through the _CM_KEY_NODE.SubKeyLists[...] cell indexes. During hive load, the value at index 0 of the array may either be a subkey index, or HCELL_NIL if there are no subkeys; index 1 must always be equal to HCELL_NIL, as by definition there are no volatile subkeys on disk. The high-level concept behind the subkey index is that it is a linear list of key node cell indexes, which must efficiently support the following operations (from most to least commonly used, in my subjective opinion):

1. Finding a key by name,
2. Finding a key by index on the list,
3. Adding a new key to the list,
4. Deleting a key from the list.

Regardless of the underlying representation of the list, it is always stored in a lexicographical order, reducing the lookup-by-name time from linear to logarithmic by using binary search. Let's now look into the specific structures used in registry hives to implement this functionality.

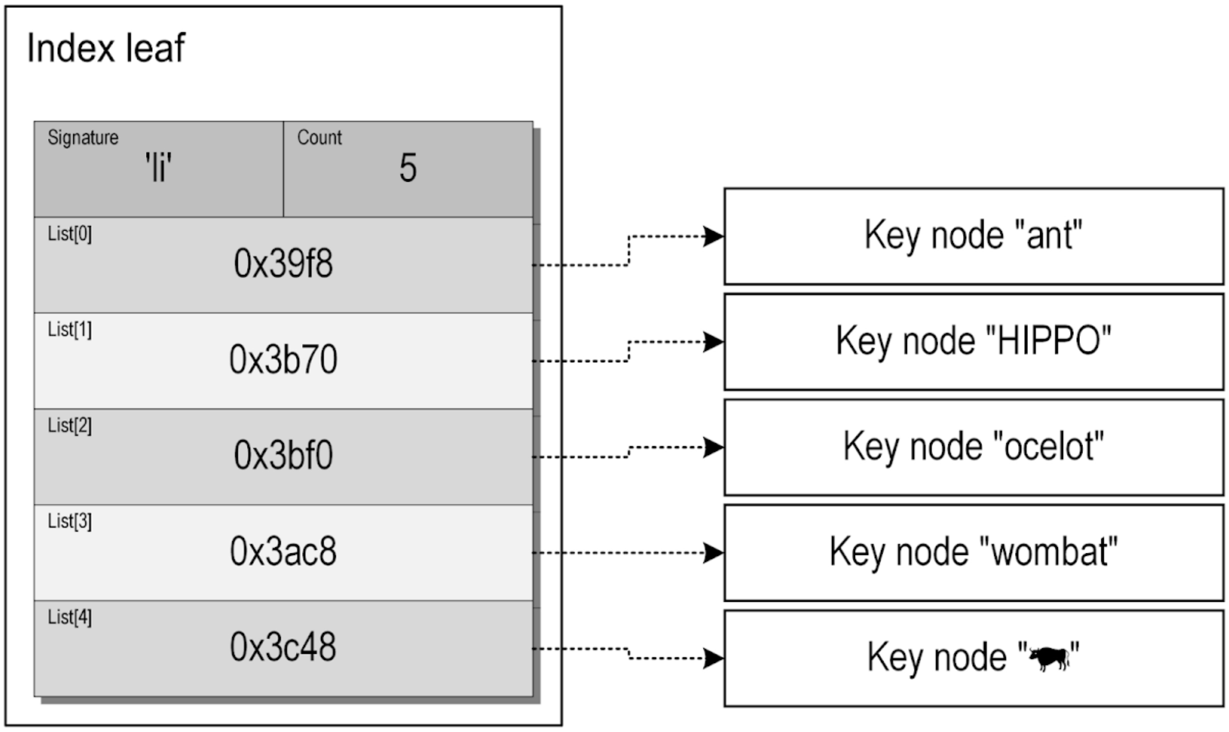
Index leaves

Index leaves are the most basic type of a subkey list, which has been supported since the first iteration of the regf format and consists of just three members: the signature (0x696C, 'li'), number of entries (16-bit), and an inline, variable-length list of the cell indexes. The corresponding Windows kernel structure is _CM_KEY_INDEX:

```
0: kd> dt _CM_KEY_INDEX

nt!_CM_KEY_INDEX

   +0x000 Signature        : Uint2B

   +0x002 Count            : Uint2B

   +0x004 List             : [1] Uint4B
```

Given the Count field range, the index leaf can store up to 65535 subkeys. It is the most compact one in terms of disk/memory consumption, but it provides somewhat poor cache locality, because every key referenced during the lookup must be accessed in memory in order to read its name from _CM_KEY_NODE.Name. Nevertheless, index leaves are still commonly used in all versions of Windows up to this day.

As an example, let's consider a key with five subkeys named "wombat", "🐂", "HIPPO", "ant", and "ocelot". An index leaf of such a key could look like this:



This illustrates that entries in the list are indeed stored in a sorted manner, and in a case-insensitive way – "ant" goes before "HIPPO" even though 'H' (0x48) < 'a' (0x61). However, this logic applies to comparisons only, and otherwise the letter casing specified during key creation is preserved and visible to registry users. Finally, the unicode ox symbol is placed last on the list, because it is encoded as U+D83D U+DC02, and 0xD83D is greater than any of the ASCII characters in the other names.

Fast leaves

Fast leaves are slightly younger than subkey indexes, introduced in regf version 1.3 in 1995 (Windows NT 4.0). As hive versions 1.2 and below have been long obsolete, that means that fast leaves are universally supported in every modern version of Windows at the time of this writing. As the name suggests, they are meant to be faster than their predecessors, by including up to four initial characters of each subkey in the list as a "hint" next to the cell index of the key. This allows the kernel to execute the first four iterations of the string comparison loop using data only from the fast leaf and without referring to the corresponding node, which addresses the aforementioned issue of poor cache locality in index leaves. We expect this optimization to be effective in most real-life scenarios, as most keys consist of ASCII-only characters and differ from each other within the first four symbols.

The specific logic of generating the 32-bit hint from a string can be found in the internal CmpGenerateFastLeafHintForUnicodeString kernel function, but is boils down to the following steps:

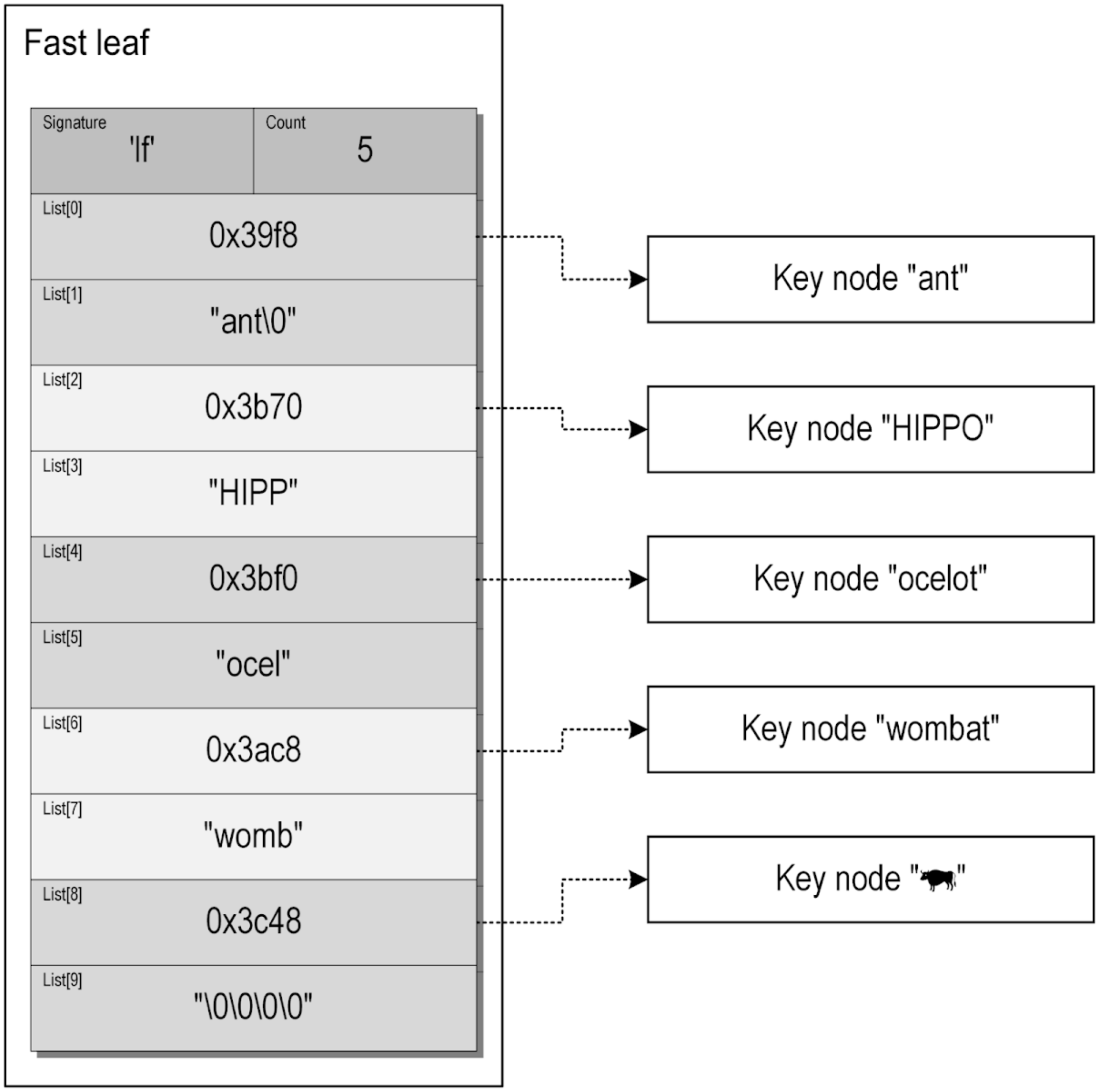
1. Set the initial hint variable to 0
2. In a loop of min(4, length) iterations:

   1. If the n-th character is greater than 0xFF, break
   2. Otherwise add the character (with its original case) to the hint

3. Return the hint to the caller

For example, the hint for "ant" is "ant\0", the hint for "HIPPO" is "HIPP", and the hint for "🐂" is "\0\0\0\0" (the first character is non-ASCII, so the whole hint is simply zero).

When it comes to the structure layout of the fast leaf, it is basically the same as the index leaf, but it has a different signature ('lf') and twice as many entries in the List array due to the addition of hints. There doesn't seem to be any structure definition corresponding specifically to fast leaves in the public symbols, which either means that the structure is a non-public one, or it is also accessed via _CM_KEY_INDEX in the source code, but through references such as Index.List[2*n] instead of Index.List[n]. An illustration of a fast leaf containing the five example subkeys is shown below:



Hash leaves

Hash leaves are the third and last (for now) iteration of the subkey index format, introduced in Windows XP in 2001 (regf version 1.5). They have exactly the same data layout as fast leaves, but are characterized by the 'lh' signature, and the 32-bit hint is a simple hash of the entire string instead of an inline representation of the first four characters. The specific hashing algorithm is implemented in the internal CmpHashUnicodeComponent function, and can be summarized with the following steps:

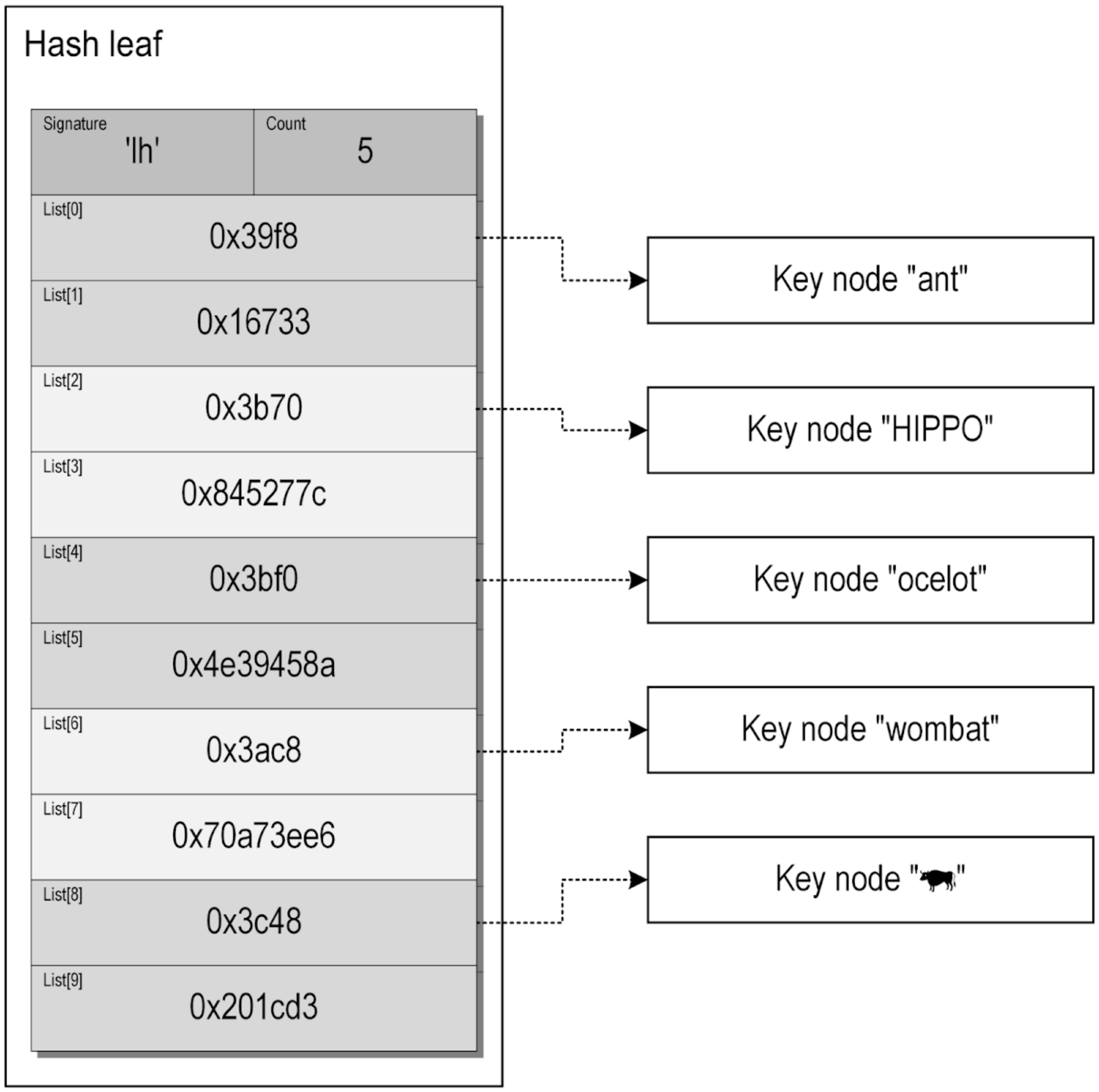
1. Start with a hash equal to 0
2. For every character in the string:

1. Hash = (Uppercase(Character) + 37 * Hash) % 0x100000000

3. Return the hash to the caller

The main benefit of this approach is that it works equally well with ASCII and non-ASCII strings, and it covers the entire name and not just a prefix, further limiting the number of necessary references to the subkey nodes during key lookup. However, you may notice that a full-string hash isn't really compatible with the concept of binary search, and indeed, whenever a hash leaf is used, the kernel performs a linear search instead of a binary one, as can be seen in the corresponding CmpFindSubKeyByHashWithStatus function. In theory, this could lead to iterating through 65535 keys (the maximum number of entries in a hash leaf), but in practice, the kernel makes sure that a hash leaf is never longer than 1012 elements. This is okay for performance, because when more subkeys are associated with a key, a second-level data structure comes into play (the root index, see the next section), and that one is always traversed with a binary search. Overall, it seems possible that the cache friendliness of the hash leaf makes up for its theoretically worse lookup complexity, especially in the average case.

A corresponding diagram of a hash leaf data layout is shown below:

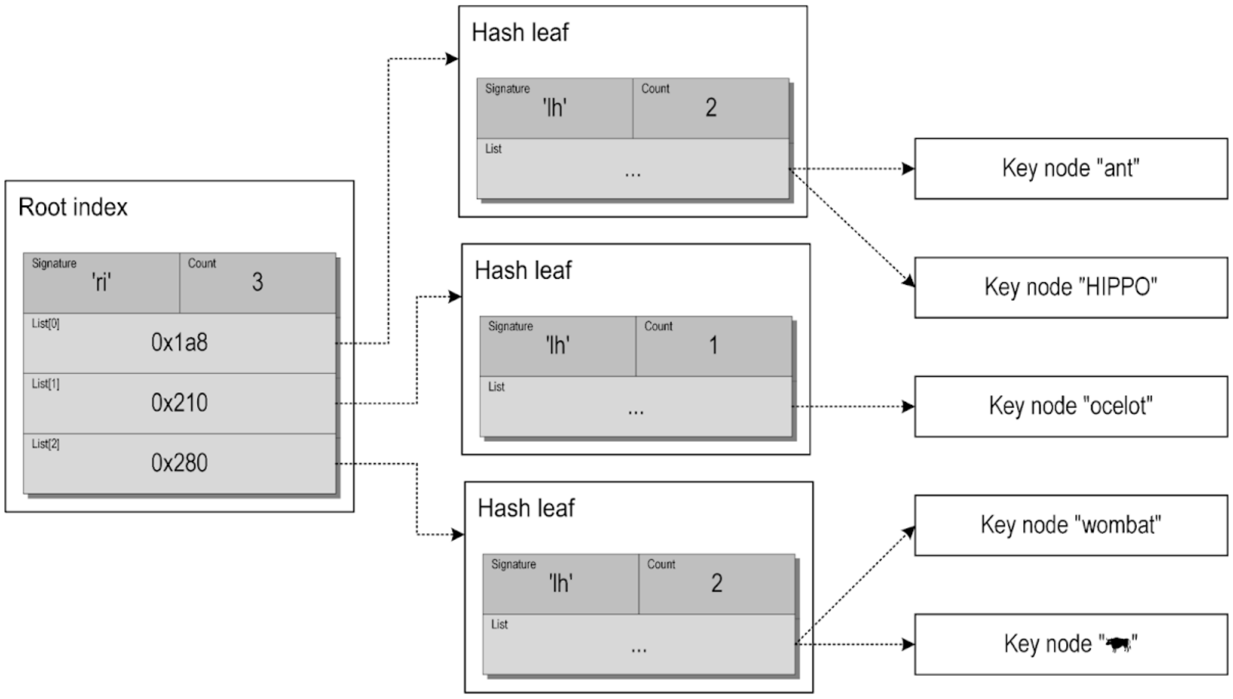


Root indexes

Each key in the registry can potentially have many thousands of subkeys, but having them stored in one very long list (such as a single index, fast or hash leaf) could lead to poor performance for some operations. For example, whenever a new key is inserted into the alphabetically sorted list, the portion of the list after the new key has to be moved in memory to make room for the new item. Similar CPU-heavy situations could arise when extending the dynamically sized array in the hive, and potentially having to copy its entire contents to a new cell if the existing one doesn't have any free space behind it. In the worst case scenario, this would have a complexity of O(n) per operation, which is too slow for such an important system mechanism as the Windows registry.

It is likely for this reason that whenever the subkey list becomes longer than 1012 elements for the first time, a second-level index called the root index is inserted into the data structure. This has the goal of splitting a single long list into several shorter ones, which are easier to manage in memory. Root indexes cannot be nested or referenced recursively by one another: a subkey list may either be non-existent, a single leaf-type list, or a single root index pointing at leaf-type lists (in other words, the list may be 0, 1 or 2 levels deep).

The root index has existed for as long as the index leaves have: since the very first regf version 1.0 in Windows NT 3.1 Pre-Release. It also has the same layout represented by the _CM_KEY_INDEX structure, which consists of a signature ('ri' in this case), a 16-bit count and an array of cell indexes pointing at leaf-type lists, without any additional hints. An example diagram of a two-level subkey index containing five keys is shown below:



Fundamental subkey list consistency requirements

There is a set of some very basic format consistency requirements concerning subkey indexes, which must be always met for any active hive in the system, regardless of whether it has been loaded from disk or created from scratch at runtime. These are the minimum set of rules for this data structure to be considered as "valid", and they are tightly connected to the memory safety guarantees of the kernel functions that operate on them. They are as follows:

- The signature of each subkey list cell must be correctly set to its corresponding type, one of 'li', 'lf', 'lh' or 'ri'.
- The size of the cell must be greater or equal to the number of bytes required to store all of the elements in the "List" array, according to the value of the "Count" member.
- A subkey list cell may never be empty, i.e. _CM_KEY_INDEX.Count mustn't be zero (whenever it becomes zero, it should be freed and un-referenced in any of the other hive cells).

- The number of subkeys cached in the key node (_CM_KEY_NODE.SubKeyCounts[x]) must be equal to the number of subkeys defined in the subkey index (i.e. the sum of _CM_KEY_INDEX.Count of its index leaves).
- The cell indexes stored in _CM_KEY_NODE.SubKeyLists[x] must either be HCELL_NIL (if SubKeyCounts[x] is zero), or point to a root index or one of the three leaf types. Additionally, SubKeyCounts[1] must be zero and SubKeyLists[1] must be HCELL_NIL on hive load.
- All cell indexes stored in a root index must point at valid leaf indexes.
- All cell indexes stored in leaf indexes must point at valid key nodes.
- All hints specified in the fast leaves and hash leaves must be consistent with the names of their corresponding keys.
- The overall subkey list must be sorted lexicographically, i.e. the name of each n+1th subkey must be strictly greater than the name of the nth subkey. This also entails that there mustn't be any duplicates in the subkey list, neither with regards to the cell index or the subkey name.

Notably, there are also some constraints that seem very natural, but are in fact not enforced by the Windows kernel:

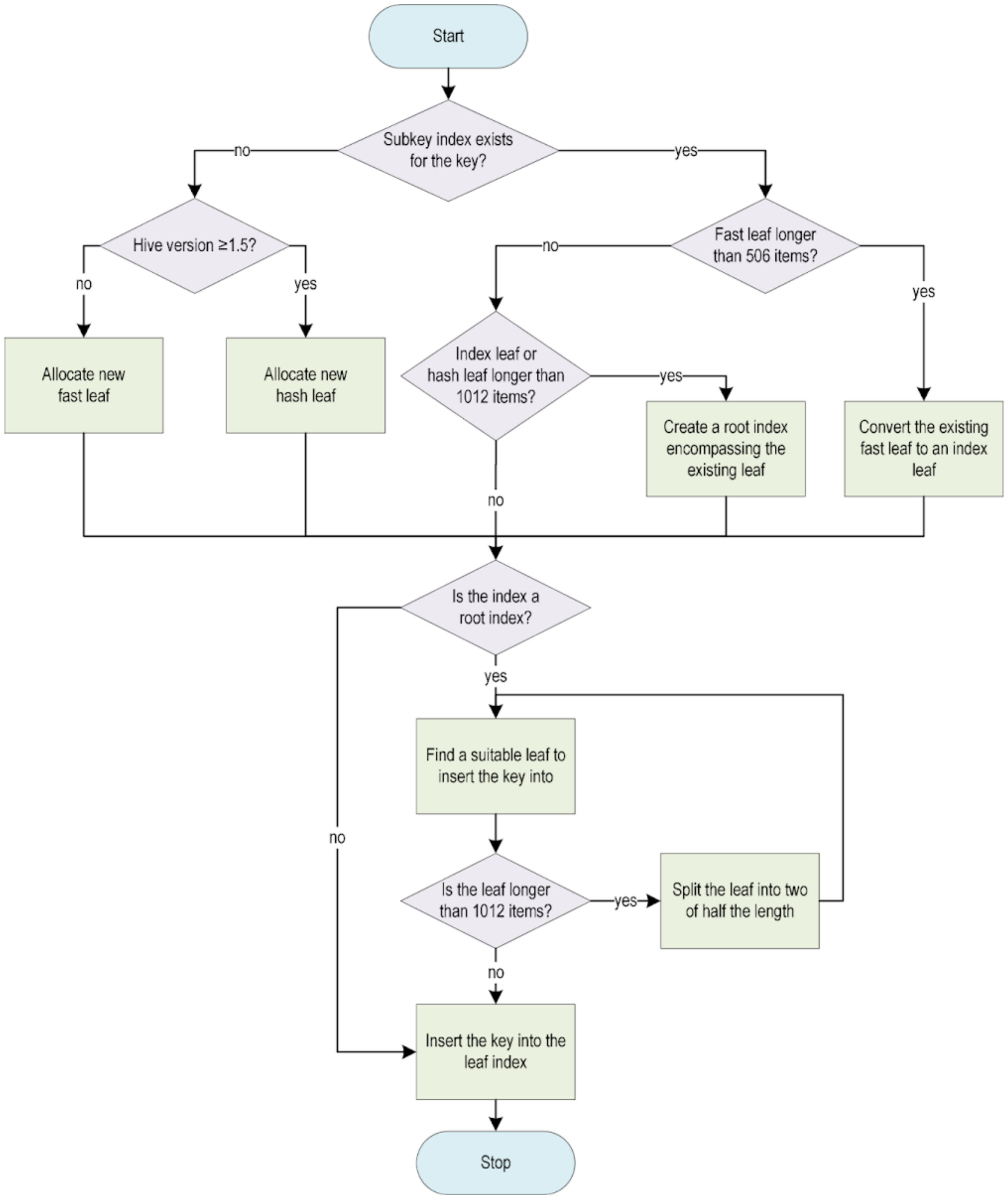
- There is no requirement that the format of a leaf-type index must be consistent with the version of the hive: instead, every one of li/lf/lh types are accepted for every hive version 1.3 – 1.6. The most glaring example of this behavior is that hash leaves are allowed in hive versions 1.3 and 1.4, even though they were historically only introduced in version 1.5 of the format.
- There is no requirement that all the leaf indexes referenced by a root index are all of the same type. In fact, a single subkey list may consist of an arbitrary combination of index leaves, fast leaves and hash leaves, and the kernel must handle such situations gracefully.
- Beyond the fact that none of the actively used subkey indexes may be empty, there are no limitations with regards to how the subkeys are laid out in the data structure. For example, the existence of a root index doesn't automatically indicate that there are many subkeys on the list: there may as well be a single root index, pointing to a single leaf, containing a single subkey. It is also allowed for several leafs being part of a single root index to have wildly different counts, with some single-digit ones coexisting with others around the 64K mark. The kernel doesn't ensure any advanced "balancing" of the subkey index by default – it does split large leafs into smaller ones, but only while adding a new subkey, and not during the loading of an existing hive.

Three examples of kernel vulnerabilities that were directly related to the handling of subkey lists are: CVE-2022-37956 (integer overflows in registry subkey lists leading to memory corruption), CVE-2022-38037 (memory corruption due to type confusion of subkey index leaves in registry hives) and CVE-2024-26182 (subkey list use-after-free due to mishandling of partial success in CmpAddSubKeyEx). I personally find the first one (CVE-2022-37956) particularly interesting, because the hive memory corruption could be triggered with the right sequence of API calls, or even just command-line reg.exe tool invocations. Granted that the number of required operations was quite high (around 66 million), but it still goes to show that being intimately familiar with the inner workings of the target software may open new avenues of exploitation that would otherwise not be available. For a detailed explanation of the subkey list management logic, see the next section.

Internal Windows logic of handling subkey lists

On top of the requirements and restrictions imposed by the regf format itself, there are some further characteristics of most registry hives found on real systems, caused by some decisions implemented in the logic of the Windows kernel. The most important thing to note is that, as mentioned above, the kernel operates on any subkey list lazily, only when there is a need to do so due to a key being added/deleted in the registry. Therefore, a weirdly formatted (but adhering to the bare regf requirements) subkey index will remain in this state after loading, for as long as a client application doesn't decide to change it.

Most of the relevant high-level logic of handling subkey lists takes place when adding new keys, and is illustrated in the flow chart below:


Start
Subkey index exists for the key?
no
yes
Hive version ≥1.5?
no
yes
Allocate new fast leaf
Allocate new hash leaf
Fast leaf longer than 506 items?
no
yes
Index leaf or hash leaf longer than 1012 items?
yes
no
Create a root index encompassing the existing leaf
Convert the existing fast leaf to an index leaf
Is the index a root index?
yes
no
Find a suitable leaf to insert the key into
Is the leaf longer than 1012 items?
yes
Split the leaf into two of half the length
no
Insert the key into the leaf index
Stop

The general high-level function that implements the above logic in the Windows kernel is CmpAddSubKeyEx, which then calls a few helper routines with mostly self-descriptive names: CmpAddSubKeyToList, CmpSelectLeaf, CmpSplitLeaf and CmpAddToLeaf. Compared to addition, the process of deleting a key from the list is very straightforward, and is achieved by removing it from the respective leaf index, freeing the leaf if it was the last remaining element, and freeing the root index if it was present and the freed leaf was its last remaining element. There are no special steps being taken other than the strictly necessary ones to implement the functionality.

Given the above, we can conclude that registry hives created organically by Windows generally adhere to the following set of extra rules:

- The leaf types being used are in line with the version of the hive: index and fast leaves for versions ≤1.4, and hash leaves for versions ≥1.5.
- All leaves within a single index root have the same type.
- Index leaves never contain more than 1012 elements.
- Once a root index is created for a key, it is never downgraded back to a single leaf index other than through the deletion of all subkeys, and creating a new one starting from an empty subkey list.

Security descriptors

Security descriptors play a central role in enforcing access control to the information stored in the registry. Their significance is apparent through the fact that they are the only mandatory property of registry keys, as opposed to classes, values and subkeys which are all optional. At the same time, large groups of keys typically share the same security settings, so it would make little sense to store a separate copy of the data for every one of them. For example, in a default installation of Windows 11, the SOFTWARE hive includes around 250,000 keys but only around 500 unique security descriptors. This is why they are the only type of cell in the hive that can be associated with multiple keys at the same time. By only storing a single instance of each unique descriptor in the hive, the system saves significant disk and memory space. However, this efficiency requires careful management of each descriptor's usage through reference counting, which ensures they can be safely freed when no longer needed.

When loading a hive, the kernel enumerates all of its security descriptors without having to traverse the entire key tree first. In order to make this possible, security descriptors in the stable space are organized into a doubly-linked list, starting at the descriptor of the root key. Internal consistency of this list is mandatory – if any inconsistencies are found, it is reset to become a single-entry list with just the root security descriptor and nothing else. If the root security descriptor itself is corrupted, the hive is deemed to be in an unrecoverable state and rejected completely.

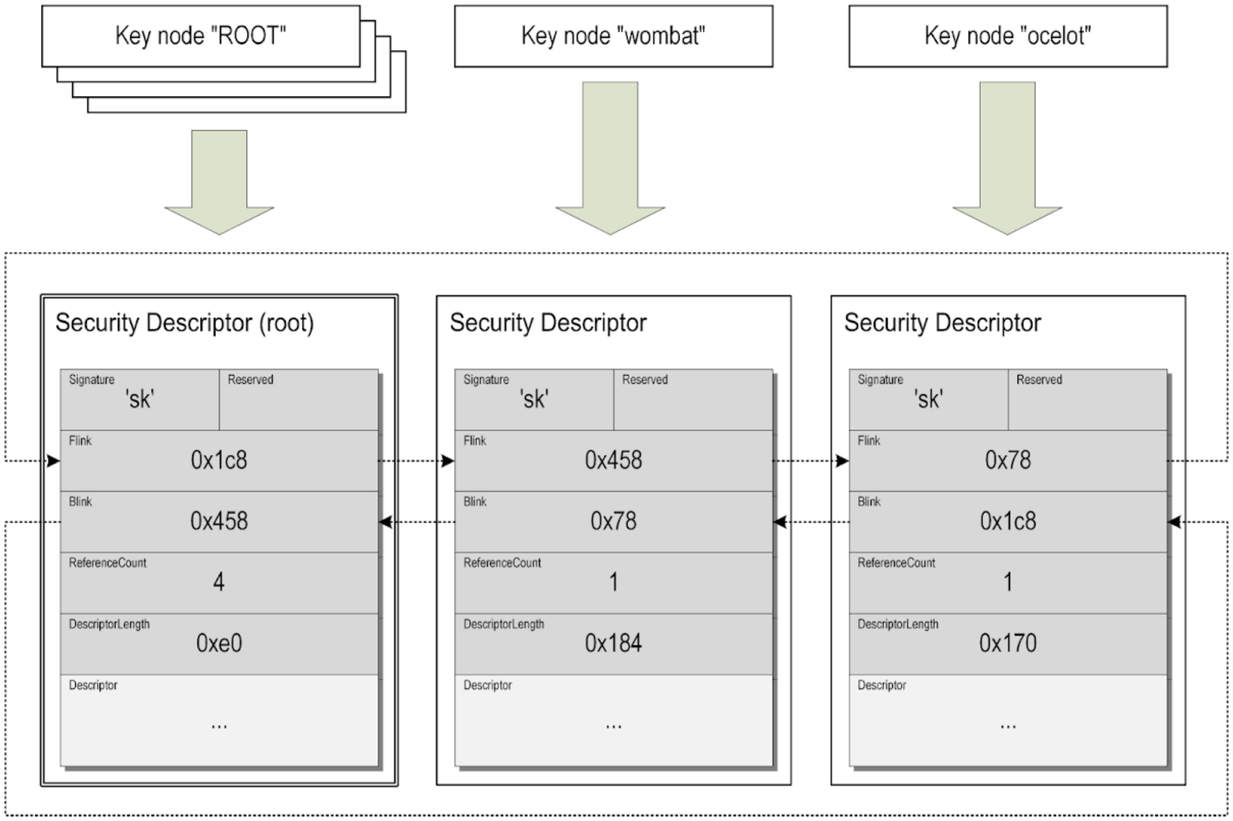
While traversing the global list, the kernel also verifies that the binary encoding of the security descriptors is valid and safe to pass to internal security-related functions later in time. In the hives, descriptors are formatted as self-contained blobs of bytes adhering to the SECURITY_DESCRIPTOR_RELATIVE structure layout. Compared to other hive cells (key nodes etc.), the internal format of security cells is relatively complex: it is variable in size and contains multiple sub-structures (SIDs, ACLs, ACEs), length indicators and internal offsets. To detect any potential corruption early, the RtlValidRelativeSecurityDescriptor function must succeed for every descriptor in a newly loaded hive, otherwise the previously discussed fallback logic takes place.

The last step in the security descriptor validation process is to make sure that the reference counts specified in the hive are equal to the actual number of references from registry keys. This is achieved by re-counting the references when traversing the key tree structure of the hive, and later checking if the values found in _CM_KEY_SECURITY.ReferenceCount are in line with the regenerated counts. If the two values are unequal, the refcount in the security cell is adjusted to reflect the correct number of references. This is critical for system security, because operating on an invalid refcount – especially an inadequately small one – may directly lead to exploitable memory corruption conditions.

Some examples of historical vulnerabilities related to the three fundamental aspects of security descriptor consistency are as follows:

Security descriptor binary format validity: CVE-2022-35768

A high-level illustration of a security descriptor linked list consisting of three elements is shown in the diagram below:



Security cell format

Let's now have a look at the specific layout of the security cells. They are represented by the _CM_KEY_SECURITY structure, whose definition is shown in the WinDbg format below:

```
0: kd> dt _CM_KEY_SECURITY

nt!_CM_KEY_SECURITY
```

```
+0x000 Signature       : Uint2B
+0x002 Reserved        : Uint2B
+0x004 Flink           : Uint4B
+0x008 Blink           : Uint4B
+0x00c ReferenceCount   : Uint4B
+0x010 DescriptorLength : Uint4B
+0x014 Descriptor      : _SECURITY_DESCRIPTOR_RELATIVE
```

Each of its fields is discussed in more detail in the following subsections.

Signature

The magic bytes of this cell type, equal to 0x6B73 ('sk'). It exists for informational purposes only, but isn't used for anything at runtime – it isn't even verified on hive load, and can therefore be anything in a binary-controlled hive.

Reserved

An unused field that may contain arbitrary data; never accessed by the kernel.

Flink and Blink

As discussed earlier, these are the forward and backward links in the security descriptor list. They must always be kept in a valid state. In a single-element list, Flink/Blink point at themselves – that is, at the security descriptor they are both part of.

ReferenceCount

This single field was arguably responsible for the most registry-related vulnerabilities out of all of the hive structures. It is a 32-bit unsigned integer that expresses the number of objects that actively rely on this security descriptor, which mostly means the key nodes associated with it, but not only. Whenever this member gets out of sync with the real number of references, it may lead to serious memory corruption primitives, so it is very important that the kernel ensures its correct value both on hive load and during any subsequent operations. The two prevalent risks are that:

- The refcount gets too small: when this happens, it is possible that the cell gets freed while some objects still hold active references to it. This leads to a straightforward use-after-free scenario, and in my experience, it is easily exploitable by a local attacker.

- The refcount gets too large: this situation doesn't immediately lead to memory corruption, but let's remember that the structure member has a limited, 32-bit width. If an attacker were able to indiscriminately increment the counter without real references to back it up, they could eventually get it to the maximum uint32 value, 0xFFFFFFFF. For many years, the Windows kernel hasn't implemented any protection against registry refcount integer overflows, so another incrementation of the field after 0xFFFFFFFF would wrap it back to zero, which brings us to the previous scenario of an inadequately small count. However, following some bug reports and discussions, Microsoft has gradually added overflow protection in the relevant, internal functions, starting in April 2023 and eventually landing the last missing check in November 2024. Thanks to this effort, I believe that as I am writing this, security descriptor refcount leaks should no longer be an exploitable condition.

Under most circumstances, the value of the refcount is somewhere between 1 and ~24.4 million (the maximum number of keys in a hive given the space constraints). However, it is interesting to note that it might be legitimately set to a greater value. Consider the following: immediately after loading a hive, all security refcounts are exactly equal to the number of keys associated with them. But, key nodes globally visible in the registry tree are not the only ones that can reference security cells; there may be also keys that have been created in the scope of a transaction and not committed yet, as well as pending, transacted operations of changing the security properties of a key (marked by the UoWAddThisKey and UoWSetSecurityDescriptor enums of type UoWActionType). They too may increase the refcount value beyond what would normally be possible with just regular, non-transacted keys. This phenomenon has been discussed in detail in the CVE-2024-43641 bug report.

Overall, reference counts are of great importance to system security, and every registry operation that involves it deserves a thorough security assessment.

DescriptorLength

This is the length of the security descriptor data (i.e. the size of the Descriptor array) expressed in bytes. It's worth noting that the format doesn't force it to be the minimum length sufficient to store the binary blob. This means that the overall cell length must be greater than DescriptorLength + 20 (i.e. the declared length of the descriptor plus the _CM_KEY_SECURITY header), and in turn DescriptorLength must be greater than the actual size of the descriptor. Both cases of the cell size or the DescriptorLength having non-optimal values are accepted by the kernel, and the extra bytes are ignored.

Descriptor

This variable-length array stores the actual security descriptor in the form of the SECURITY_DESCRIPTOR_RELATIVE structure. It doesn't necessarily have to be formatted in the most natural way, and the only requirement is that it successfully passes the RtlValidRelativeSecurityDescriptor check with the RequiredInformation argument set to zero. This means, for example, that the Owner/Group/Sacl/Dacl components may be spread out in memory and have gaps in between them, or conversely, that their representations may overlap. This was one of the main contributing factors in CVE-2022-35768, but the fix was to more accurately calculate the length of irregularly-encoded descriptors, and the freedom to structure them in non-standard ways has remained. It is even possible to use a completely empty descriptor without any owner or access control entries, and such a construct will be acknowledged by the system, too.

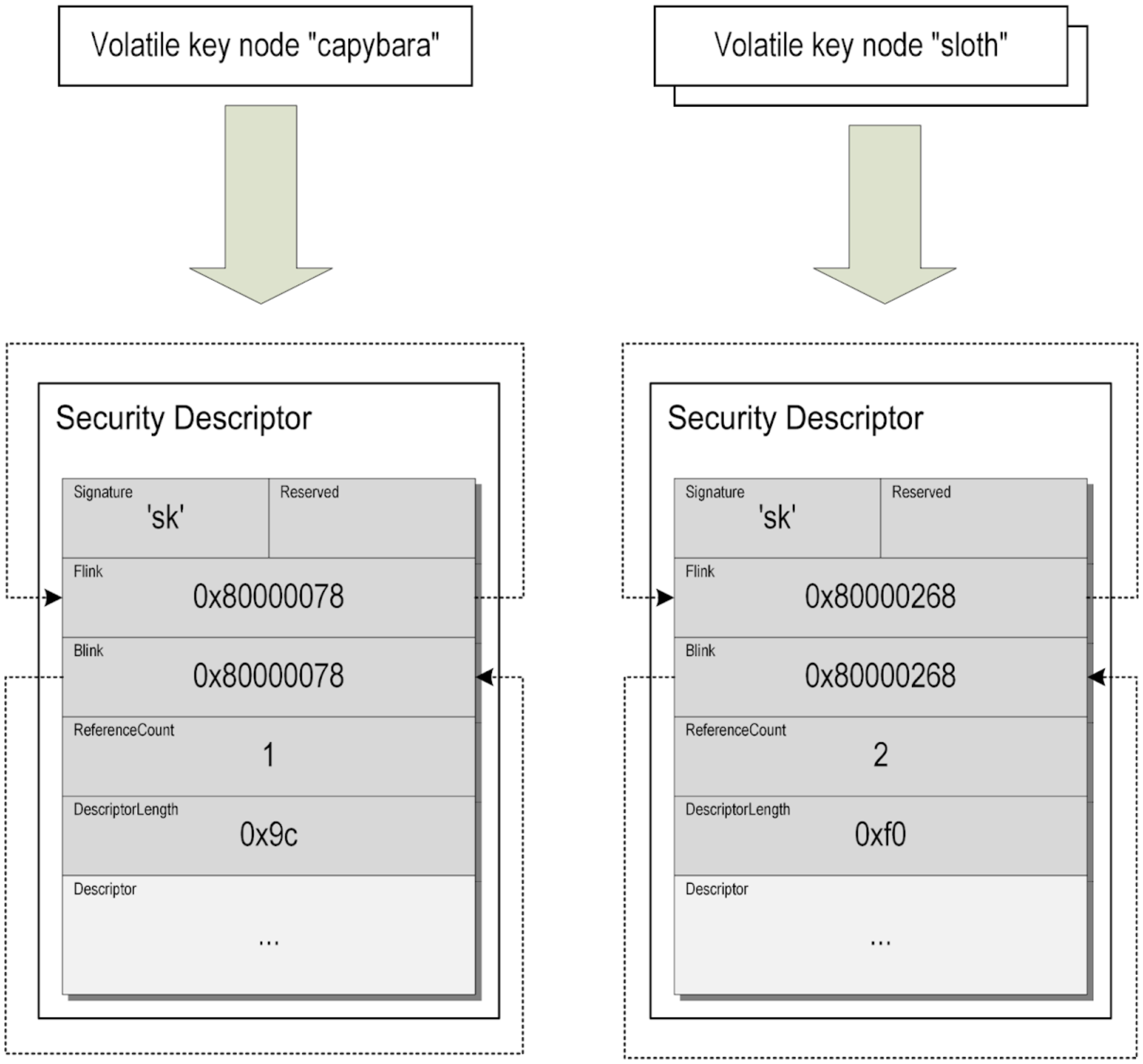
Another somewhat interesting fact is that security descriptors are meant to be deduplicated, so naturally whenever a user assigns a security descriptor that already exists in the hive, it is simply reused and its reference count is incremented. However, again, the format (or rather its canonical implementation in Windows) doesn't force the uniqueness requirement upon the security descriptors in hives loaded from disk. So, even though they would be never created by the OS itself, multiple identical copies of a descriptor are allowed in specially crafted hives and may co-exist without (seemingly) causing any issues for the kernel.

The access rights defined by the security descriptors are based on permissions specific to the registry and its operations, so there is an access mask dedicated to creating keys (KEY_CREATE_SUB_KEY), reading values (KEY_QUERY_VALUE), writing values (KEY_SET_VALUE), and so on. They all have self-descriptive names and are well-documented in Registry Key Security and Access Rights, so we won't spend more time discussing them here.

Security descriptors of volatile keys

Similarly to every other property of a registry key, the storage type of a security descriptor always matches the type of its associated key(s). This means that a stable key will always use a stable descriptor, and a volatile key – a volatile descriptor. It is the only "exception" to the rule that security descriptors are deduplicated and unique within the scope of the hive. If there are two keys with identical security settings but different storage types, they will reference two distinct security descriptor cells via their _CM_KEY_NODE.Security fields, one with the highest bit set and the other with the bit clear. The descriptors stored on both sides are subject to the same rules with regards to reference counting, allocating and freeing.

Furthermore, we have previously discussed how all security descriptors in a hive are connected in one global doubly-linked list, but this only applies to the descriptors in the stable space. The functionality is needed so that the descriptors can be enumerated by the kernel when loading a hive from disk, and since volatile descriptors are in-memory only and disappear together with their corresponding keys on hive unload or a system shutdown, there is no need to link them together. The internal CmpInsertSecurityCellList function takes this into account, and points the Flink/Blink fields at themselves, making each volatile descriptor a single-entry list in order to keep it compatible with the list linking/unlinking code. This behavior is illustrated in the diagram below, with two volatile security descriptors each being in their own pseudo-list:



This slight quirk is the reason why the ability to create stable keys under volatile ones, which should normally not be possible, may be an exploitable condition with security impact. For details, see the "Creation of stable subkeys under volatile keys" section in the CVE-2023-21748 bug report, or the CVE-2024-26173 bug report.

Security descriptors in app hives

In normal registry hives, there are no artificial restrictions with regards to security descriptors. There may be an arbitrary number of them, and they may contain any type of settings the user wishes, as long as they have binary control over the hive file and/or the existing security descriptors grant them the access to change them to whatever they want. However, there are some limitations concerning security descriptors in application hives, as documented in the MSDN page of the RegLoadAppKeyA function:

All keys inside the hive must have the same security descriptor, otherwise the function will fail. This security descriptor must grant the caller the access specified by the samDesired parameter or the function will fail. You cannot use the RegSetKeySecurity function on any key inside the hive.

The intent behind the quote seems to be that the security settings within an app hive should be uniform and immutable; that is, remain identical to their initial state at hive creation, and consistent across all keys. There is indeed some truth to the documentation, as trying to change the security of a key within an app hive with RegSetKeySecurity, or to create a new key with a custom descriptor both result in a failure with STATUS_ACCESS_DENIED. However, the part about all keys having the same security descriptor is not actually enforced, and a user can freely load an app hive with any number of different security descriptors associated with the keys. This was reported to Microsoft as WinRegLowSeverityBugs issue #20, but wasn't deemed severe enough to be addressed in a security bulletin (which I agree with), so for now, it remains an interesting discrepancy between the documentation and implementation.

Key values and value lists

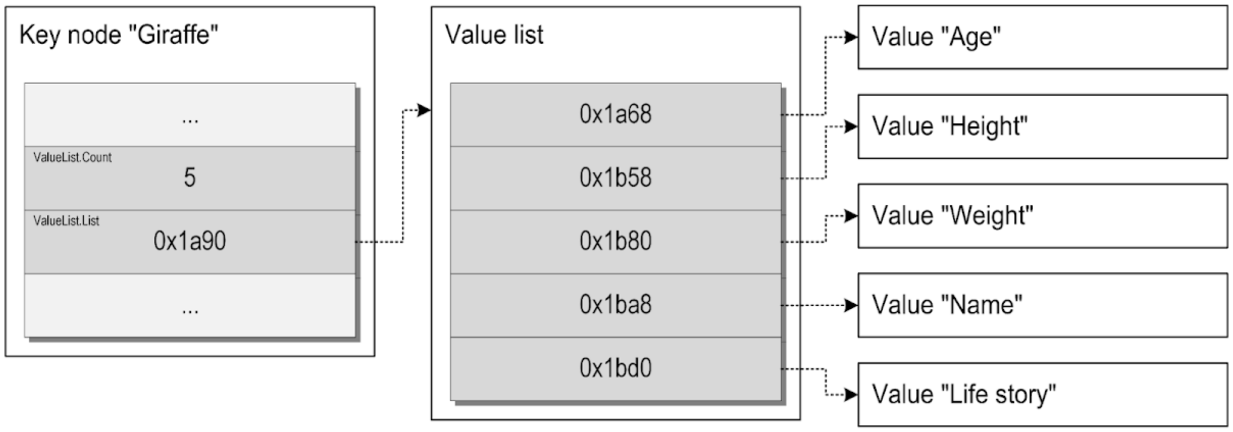
While keys allow software to create a data organization hierarchy, values are the means of actually storing the data. Each value is associated with one specific key, and is characterized by the following properties:

- Name
- Type
- Data

In general, values are much simpler than keys. To begin with, they are not a full-fledged object in the NT Object Manager sense: you cannot open a handle to a value, and thus you may only access them through the handle of its associated key and its name. They also don't have dedicated security descriptors, so a client with a key handle with the KEY_QUERY_VALUE access can enumerate and read all values of the key, and the KEY_SET_VALUE rights allows the caller to create/modify/delete all values within a key. For these reasons, values are best thought of as elaborate attributes of a key, not as an independent entity.

There is no fixed limit on the number of values associated with a key other than the available hive space, which places the number at around 67 million (0x80000000 ÷ 0x20, the hive space divided by the minimum value cell size). The value list format is also not as optimized as the subkey index is: it is a linear, single-level list with just the raw value cell indexes, without any additional metadata like a header or hints. The list is not sorted either, and their order is defined by when they were added to the key. Finally, value name uniqueness is guaranteed on output, but not enforced on input: it is possible to load a specially crafted hive with several values with the same name, and contrary to duplicate keys, this doesn't seem to pose any fundamental problems for the registry implementation.

A high-level overview of the hive cells related to a key's value list is shown below:



In the next section, we will examine the internal layout and semantics of the _CM_KEY_VALUE structure, which describes each unique value in the registry.

The key value cell

As usual, we can print out the structure definition in WinDbg:

```
0: kd> dt _CM_KEY_VALUE

nt!_CM_KEY_VALUE

   +0x000 Signature        : Uint2B

   +0x002 NameLength       : Uint2B

   +0x004 DataLength       : Uint4B

   +0x008 Data             : Uint4B

   +0x00c Type             : Uint4B

   +0x010 Flags            : Uint2B

   +0x012 Spare            : Uint2B

   +0x014 Name             : [1] Wchar
```

Let's examine each field more closely.

Signature

It identifies the cell as a key value, and must be equal to 0x6B76 ('vk'). It is verified during hive load, but isn't used for anything else later on.

NameLength and Name

The combination of these two fields specifies the name of the value: NameLength indicates the length of the string in bytes, and Name is an inline, variable-length buffer that stores the name itself. Let's consider the same criteria of the name that we have previously discussed in the context of registry keys:

- Compression: Similarly to keys, value names may be compressed if the VALUE_COMP_NAME (0x1) flag is set in _CM_KEY_VALUE.Flags. In that case, the string is stored as 8-bit ASCII characters, otherwise the normal wide-character encoding is used.
- Length: The length of the name can be between 0 and 16,383 characters. A length of zero indicates an alias for the value displayed by Regedit as "(Default)", a remnant of the design from Windows 3.1 where data was assigned directly to keys. As a sidenote, the correct enforcement of the upper limit was only introduced in October 2022 as a fix for CVE-2022-37991.
- Charset: All characters in the 0x0000 – 0xFFFF range are allowed in a value name, with no exceptions. Since values are not part of the same namespace as keys, this even includes backslashes. The only constraint is that if the corresponding key is a symbolic link, then the value must be named "SymbolicLinkValue", as it has a special meaning and stores the link's target path. An example of a bug related to sanitizing value names was CVE-2024-26176.
- Uniqueness: Value name uniqueness is not enforced on input, but it is maintained by the kernel at runtime on a best-effort basis. That means that whenever setting a value, the system will always try to reuse an existing one with the same name before creating a new one. Similarly to keys, value lookup is performed in a case-insensitive manner, but the original casing is preserved and visible to the clients.

DataLength

Specifies the length of the data stored in the value. The various ranges of the 32-bit space that the field can fall into are explained below:

DataLength	Hive versions < 1.4	Hive versions ≥ 1.4
0x0	Empty value, `Data` must be set to HCELL_NIL.	
0x1 – 0x3FD8	Data stored directly in a backing cell pointed to by `Data`.	

0x3FD9 – 0xFFFFC	Data stored directly in a backing cell pointed to by `Data`.	Data split into 16344-byte chunks and saved in a big data object pointed to by `Data`.
0xFFFFD – 0x3FD7C028	Invalid.	
0x3FD7C029 – 0x7FFFF000		Not accepted on input due to a 16-bit integer overflow in the big data chunk count. Feasible to set at runtime, but the saved data will be truncated due to the same bug / design limitation.
0x7FFFF001 – 0x7FFFFFFF	Invalid	
0x80000000 – 0x80000004	Between 0–4 bytes stored inline in the `Data` field.	
0x80000005 – 0xFFFFFFFF	Invalid.	

Data

Responsible for storing or pointing to the data associated with the value. To summarize the table above, it can be in one or four states, depending on the data length and hive version:

1. Empty – equal to HCELL_NIL, if DataLength is 0.
2. Inline – stores up to four bytes in the Data member of the value cell itself, as indicated by DataLength & 0x7FFFFFFF, if the highest bit of DataLength is set. As a side effect, an empty value can be represented in two ways: either as DataLength=0 or DataLength=0x80000000.
3. Raw data – points to a raw backing cell if Hive.Version < 1.4 or DataLength ≤ 0x3FD8.
4. Big data – points to a big data structure introduced in hive version 1.4, which is capable of storing 0xFFFF × 0x3FD8 = 0x3FD7C028 bytes (a little under 1 GiB). More on big data cells in the section below.

Type

This field is supposed to store one of the supported value types, such as REG_DWORD, REG_BINARY, etc. We'll omit a thorough discussion of the official types, as we feel they are already well documented and understood. From a strictly technical point of view, though, it's important to note that the type is simply a hint, an extra piece of metadata that is available to a registry client with the intended purpose of indicating the nature of the value. However, Windows provides no guarantees with regards to the consistency between the value type and its data. For instance, a REG_DWORD value doesn't have to be four-bytes long (even though it conventionally is), a REG_SZ unicode string can have an odd length, and so on. Any client application that operates on user-controlled data from the registry should always check the specific properties it relies on, instead of unconditionally trusting the value type.

Beyond this flexibility in data interpretation, there's another aspect of the Type field to consider: its potential for misuse due to its 32-bit width. The kernel generally doesn't perform any verification that its numerical value is one of the small, predefined enums (other than to ensure REG_LINK for symbolic links and REG_NONE for tombstone values), so it is possible to set it to any arbitrary 32-bit value, and have it returned in exactly the same form by system APIs such as RegQueryValueEx. If a program or driver happens to use the value type returned by the system as a direct index into an array without any prior bounds checking, this could lead to out-of-bounds reads or memory corruption. In some sense, it would probably be safest for the most critical/privileged software in the system (e.g. antivirus engines) not to use the value type at all, or only within a very limited scope.

Flags

There are currently two supported flags that can be set on registry values:

- VALUE_COMP_NAME (0x1) – equivalent to KEY_COMP_NAME, indicates that the value name representation is a tightly packed string of ASCII characters.
- VALUE_TOMBSTONE (0x2) – used exclusively in differencing hives (version 1.6) to indicate that a value with the given name has been explicitly deleted and doesn't exist on this key layer. It requires that the value type is REG_NONE and it doesn't contain any data. It is equivalent to the Tombstone (1) property of a key set in the LayerSemantics field of a key node.

Spare

Unused member, never accessed by the kernel.

Big data value storage

Prior to hive version 1.4, the maximum length of a value in the registry was 1 MB, which was directly related to the maximum length of the single backing cell that would store the raw data. This limitation is documented in the Registry element size limits article:

Registry element	Size limit
Value	• Available memory (latest format) [editor's note: this is not fully accurate] • 1 MB (standard format)

Here, "standard format" refers to regf v1.3. On some level, 1 MB could be considered a reasonable limit, as the registry was not designed to serve as storage for large quantities of data – at least not initially. One example of a public resource which vocalized this design decision was the old Windows registry information for advanced users article from around 2002-2003, which stated:

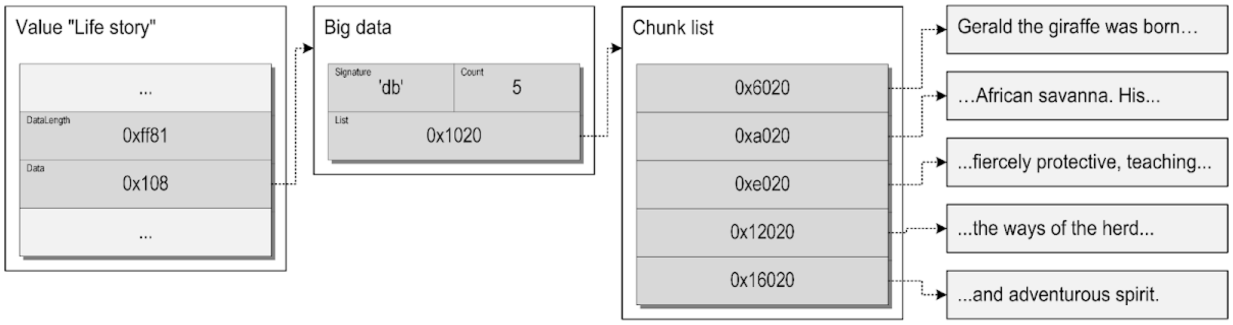
> Long values (more than 2,048 bytes) must be stored as files with the file names stored in the registry.

Nevertheless, it seems that at some point during the development of Windows XP, Microsoft decided to provide the registry clients with the ability to store larger chunks of data, not bound by the somewhat arbitrary limits of the regf format. In order to facilitate this use case, a new cell type was added, called the "big data". Conceptually, it is simply a means of dividing one long data blob into smaller portions of 16344 bytes, each stored in a separate cell. It replaces the single backing cell with a _CM_BIG_DATA structure defined as follows:

```
0: kd> dt _CM_BIG_DATA

nt!_CM_BIG_DATA

   +0x000 Signature        : Uint2B

   +0x002 Count            : Uint2B

   +0x004 List             : Uint4B
```

The signature is set to 0x6264 ('db') and verified on hive load, but otherwise not used. The count represents the number of 16344-byte chunks making up the overall value, and is generally supposed to be set to an integer between 2–65535. Otherwise, if it was set to 0, that would mean that the value is empty so the big data object shouldn't be present at all. If it was equal to 1, a direct backing buffer should have been used instead, so such a construct would also be invalid. Neither zero nor one are thus accepted by the hive loader, but it is technically possible to set these values at runtime by abusing the aforementioned integer overflow bug. We haven't found any security impact of this behavior other than it being a correctness error, though.

The last element of the structure, List, is a cell index to a basic array of cell indexes making up the value chunks. Its format is equivalent to that of the value list, which also stores just the HCELL_INDEX values without any headers or additional information. Furthermore, every chunk other than the last one must contain exactly 16344 bytes. If the length of the overall value is not divisible by 16344, the final chunk contains the remaining 1–16343 bytes. The layout of the big data object and its associated cells is shown in the diagram below:

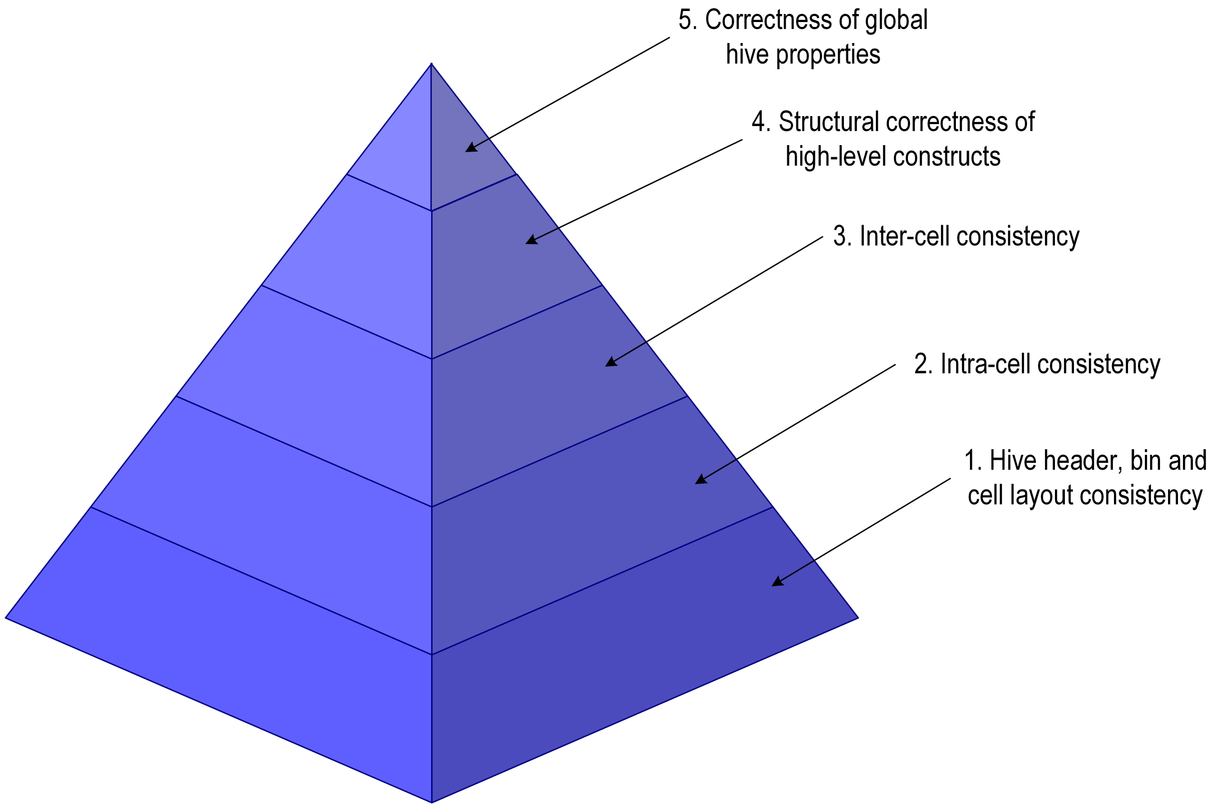


This concludes the part about the internal format of registry hives.

The hive loading and sanitization process

The hive loading process implemented by the NtLoadKey* family of system calls is a long and complex operation. It involves opening the hive file, loading it in memory, verifying its integrity, optionally recovering state from transactional log files, allocating any related kernel objects, attaching the hive to the global registry tree, and optionally opening a handle to the hive root and returning it to the caller. In this blog post, we are particularly interested in the hive sanitization part. Understanding this portion of the registry code is like consulting the official specification – or even better, as the code doesn't lie and is essentially the ground truth of what is and isn't accepted as valid data. Furthermore, it provides us with a number of hints as to which properties of the format are imperative to the correct functioning of the database, and which ones are more conventional, and don't have any serious consequences even if broken. The goal of this section is to discuss the overall control flow of loading a hive and performing the initial pass of sanitization. By documenting which internal routines are responsible for which checks, we hope to make it easier for other security researchers to navigate the hive loading code, providing a good starting point for their own investigations.

The registry, as a logical structure, is built on top of several lower-layer abstractions, each of which has a number of invariants that must hold in order for the hive to be considered valid, and in order for operations being performed on the hive to be safe. This is illustrated in the pyramid below, with the most foundational requirements placed at the bottom, and the increasingly more general aspects of hive integrity towards the top:


5. Correctness of global hive properties
4. Structural correctness of high-level constructs
3. Inter-cell consistency
2. Intra-cell consistency
1. Hive header, bin and cell layout consistency

Let's consider some examples of validity checks at each level, starting with the most fundamental ones:

1. Hive header, bin and cell layout consistency
    - Validity of the hive version, length, root cell index, flags in the header.
    - Existence of at least one bin in the hive.
    - Validity of each bin's header, particularly the file offset and size.
    - Validity of cells: aligned to eight bytes, within the bounds of the bin, completely filling out the bin.
2. Intra-cell consistency
    - Sufficient size of each cell with regards to the data it stores: at least the minimum size for the cell type (e.g. 0x4e for the key node), plus adequate to any variable-length internal arrays, such as the key name or value name.
    - Correct signatures being set for every kind of cell depending on its function.
    - Valid combinations of flags being set in key nodes and values.
    - Strings (key names, value names) adhering to the format requirements regarding minimum and maximum lengths, charset, etc.

3. Inter-cell consistency
    - Valid references to cells in cell indexes, and each allocated cell only being used for one specific purpose.
    - Consistency between copies of redundant data in separate cells: e.g. _CM_KEY_NODE.SubKeyCounts[...] vs. the length of the subkey index.
    - Consistency between length markers in one cell vs. the amount of data stored in the corresponding backing buffer (e.g. _CM_KEY_VALUE.DataLength vs. length of the data stored in the raw data cell / big data cell).
    - Correct hints in subkey indexes (fast leaves, hash leaves).
    - Correct reference counts in the security descriptors.
4. Structural correctness of high-level constructs
    - Consistency of the linked list of security descriptors.
    - Subkeys being laid out in a lexicographical order in all subkey indexes.
    - Symbolic link keys having a single value named "SymbolicLinkValue" of type REG_LINK.
    - Subkeys in the stable space always having a non-volatile parent.
5. Correctness of global hive properties
    - Each hive always containing at least one key (the root key) and at least one security descriptor.
    - Only the root of the hive, and no other key having the KEY_HIVE_ENTRY flag set.
    - The depth of the hive's tree structure being a maximum of 512 levels.

As we can see, there are a variety of constraints that require verification when loading a hive, with the more abstract ones relying on the lower-layer ones to be confirmed first. It explains why the process is by far the most complex operation one can perform on the registry, spanning across thousands of lines of code and dozens of functions. To better illustrate this process, I've outlined the most important hive validation functions below, indented to show their hierarchical relationships as they execute in the kernel:

- NtLoadKey* → CmLoadDifferencingKey → CmLoad(App)Key
  - CmpCmdHiveOpen → CmpInitHiveFromFile → CmpCreateHive
    - HvHiveStartFileBacked → HvLoadHive
      - HvpGetHiveHeader
      - HvAnalyzeLogFiles
      - HvpPerformLogFileRecovery
      - HvpRemapAndEnlistHiveBins
        - HvpValidateLoadedBin
        - HvpEnlistFreeCells
    - CmCheckRegistry
      - HvCheckHive

        HvCheckBin
      - CmpValidateHiveSecurityDescriptors
      - CmpCheckRegistry2
        - CmpCheckKey
          - CmpCheckValueList
          - CmpCheckLeaf
        - CmpCheckLexicographicalOrder
        - CmpCheckAndFixSecurityCellsRefcount
  - CmpLoadKeyCommon

    CmpLinkHiveToMaster

    ObOpenObjectByName → ... <NT Object Manager> ... →

    CmpParseKey → CmpDoParseKey

    CmpUpdateHiveRootCellFlags

Here is a short summary of each of the above functions, according to my own analysis and understanding:

Function name(s)	Description
NtLoadKey*	The four syscall entry points for loading registry hives, as discussed in the previous post: NtLoadKey, NtLoadKey2, NtLoadKeyEx, NtLoadKey3.

CmLoadDifferencingKey	A generic function for loading hives – not just differencing ones but every kind, contrary to what the name might suggest. Other than the syscall handlers, it is also called by VrpPreLoadKey and VrpLoadDifferencingHive, which are parts of the VRegDriver. It is responsible for sanitizing the input flags, checking the privileges of the caller, calling registry callbacks, invoking specialized functions to actually load the hive, and opening a handle to the root of the hive if the caller requested it.
CmLoadKey, CmLoadAppKey	Functions implementing the core functionality of loading normal and app hives, respectively. They are responsible for coordinating lower-layer loading functions, resolving any conflicts related to the hive file / registry mount path, and inserting the hive-related objects into the corresponding kernel data structures. In terms of opening and validating the binary hive representation, they are virtually equivalent.
CmpCmdHiveOpen, CmpInitHiveFromFile, CmpCreateHive	Functions dedicated to opening the hive file on disk, loading it in memory, validating its integrity and allocating the internal kernel structures (_CMHIVE and other objects representing the hive).
HvHiveStartFileBacked, HvLoadHive	Common functions for loading and sanitizing the hive on the level of header, bins and cells (the lowest level of the pyramid).
HvpGetHiveHeader	Reads and validates the hive header, trying to determine if it is valid or corrupted, and whether the header or hive data need to be recovered from a log file.
HvAnalyzeLogFiles, HvpPerformLogFileRecovery	Two most important functions related to data recovery from log files: the first one determines which of the two files (.LOG1/LOG2) to use, and the second one actually applies the log file entries to the hive mapping in memory.

HvpRemapAndEnlistHiveBins, HvpValidateLoadedBin, HvpEnlistFreeCells	Functions responsible for re-mapping the hive after log file recovery, in order to ensure that every bin is mapped as a continuous block of memory. During the process, the validity of all bins and the layout of their cells is verified.
CmCheckRegistry	A generic function encompassing the verification of levels ≥ 2 of the pyramid, i.e. everything about the hive that defines its logical structure and is not related to memory management. If any self-healing occurs during the process, the function restarts its logic, so it may potentially take multiple iterations before a corrupted hive is fixed up and accepted as valid.
HvCheckHive, HvCheckBin	Two functions responsible for validating the bin headers and layout of their cells. As you may have noticed, this part of their functionality is redundant with HvpValidateLoadedBin and HvpEnlistFreeCells. The difference is that the earlier functions are used to cache information about the positions of free cells in the hive, to optimize the allocation process later on. On the other hand, the underlying purpose of HvCheckHive and HvCheckBin is to generate a bitmap object (RTL_BITMAP) that indicates the positions of allocated cells, in order to ensure the validity of cell indexes when sanitizing the hive, and to make sure that every cell is only used for a single purpose in the hive. As a side note, there is an amusing bug in HvCheckBin related to verifying cell size correctness, but it seems to be non-exploitable precisely because the same sanitization is correctly performed earlier in HvpEnlistFreeCells.
CmpValidateHiveSecurityDescriptors	The function traverses the linked list of security descriptor cells, and verifies its consistency (the correctness of the Flink/Blink indexes) and the validity of the security descriptor blobs. At the same time, it also caches information about the descriptors in internal kernel structures, so that they can be quickly looked up when verifying the _CM_KEY_NODE.Security fields, and later at system run time.

CmpCheckRegistry2	A function responsible for performing a single attempt at validating the entire key structure. There are several possible return codes: • STATUS_SUCCESS if the hive validation passes without problems, • STATUS_REGISTRY_HIVE_RECOVERED if minor corruption was encountered, but it was successfully fixed in-place, • STATUS_RETRY if a badly corrupted key was encountered and removed from its parent's subkey index. This causes CmCheckRegistry to restart the validation process from scratch. • STATUS_REGISTRY_CORRUPT if the hive was found to be corrupted beyond repair. • Other problem-specific error codes such as STATUS_NO_LOG_SPACE or STATUS_INSUFFICIENT_RESOURCES, which cause the loading process to be aborted.
CmpCheckKey	This is the central function in the hive sanitization process, with more than a thousand lines of code in decompiled output, and likely just as many in the original source code. It essentially checks the validity of all fields within a specific key node, and also orchestrates the validation of the value list and subkey index associated with the key. If there was one function I would recommend analyzing to better understand the regf format, it would be this one.
CmpCheckValueList	Checks the consistency of a value list, each of the value cells on the list, and their backing buffers / big data objects.
CmpCheckLeaf	Validates a specific leaf subkey index, i.e. one of 'li', 'lf', 'lh'. This includes checking the cell size, signature, validity of the subkey cell indexes and their hint values.
CmpCheckLexicographicalOrder	Compares the name of two consecutive subkeys to determine if the second one is lexicographically greater than the first, in order to ensure the right sorting of a subkey index.

CmpCheckAndFixSecurityCellsRefcount	Iterates over all security descriptors in the hive, compares their refcounts loaded from disk with the values independently re-calculated while sanitizing the key tree, and corrects them if they are unequal. Since November 2024, it also frees any unused security descriptors with the reference count set to zero (they had been previously allowed, as described in WinRegLowSeverityBugs issue #10).
CmpUpdateHiveRootCellFlags	The function makes sure that the root key of the hive has the KEY_NO_DELETE and KEY_HIVE_ENTRY flags set. Interestingly, these flags are the only aspect of the regf format that is not enforced directly while loading the hive (in CmpCheckKey), but only at a later stage when the hive is being mounted in the global registry view.

Self-healing properties

The Windows implementation of the registry has the unique property that it is self-healing: the system tries very hard to successfully load a hive even if it's partially corrupted. My guess is that the reason for this design was to make the mechanism resilient against random data corruption on disk, as failure to load a system hive early during start-up would make Windows unusable. Perhaps it was decided that it was a better tradeoff to forcefully remove the broken parts of the file, with the hope that they would be automatically re-created later at run time, or that they weren't very important to begin with and the system/applications could continue to function correctly without them. And even if not, giving the user a chance to troubleshoot the problem or recover their data would still be a better outcome than bricking the machine completely.

Consequently, whenever an error is detected by the hive loading logic, it is handled in one of several ways, depending on the nature of the problem:

- Bin recreation: if HvpValidateLoadedBin indicates that any part of a bin header is corrupted, then HvpRemapAndEnlistHiveBins re-initializes it from scratch, and declares it as 4096 bytes long (regardless of the previous length).
- Cell recreation: if HvpEnlistFreeCells detects a cell with an invalid length, it converts it to a single free cell spanning from the current offset until the end of the bin, potentially erasing any other data/cells previously residing in that region.

- Small, direct fix: if a single field within a key node is found to have an invalid state, and the good/expected state is known to the kernel, the problem gets fixed by directly overwriting the old value with the correct one. Examples include cell signatures and mandatory/illegal flags.
- Single value deletion: if any inconsistencies are found in a value cell or its associated data cell(s), the specific value is removed from the key's value list.
- Deletion of entire value list: if the descriptor of a value list (i.e. its cell index or length) are invalid, or if a symbolic link contains more than one value, the entire value list of the key is cleared.
- Single key deletion: if an irrecoverable problem is found within a key node (e.g. invalid cell index, invalid cell length, invalid name), then it is removed from its parent's subkey index, and the key tree validation process is restarted from scratch.
- Deletion of entire subkey index: if any irrecoverable problem is found in a subkey index, it is deleted, and the subkey list of its associated key is cleared.
- Security descriptor list reset: if any errors are detected in the list of security descriptors (bad Flink/Blink indexes or invalid binary format), the set of descriptors in the hive is reduced to the single root descriptor, which will then be inherited by all the keys in the hive.
- Rejection of entire hive: if any issues are found with the fundamental parts of the regf format or its properties (heavily corrupted header, missing bins, invalid root key, invalid root security descriptor), the loading of the hive is completely aborted.

As we can see, Windows implements a very defensive strategy and always attempts to either fix the corrupted data, or isolate the damage by deleting the affected object while preserving the overall hive integrity. Only when these repair attempts are exhausted does the kernel abort the loading process and return an error. This resilience can lead to situations where a freshly loaded hive is already in a "dirty" state, requiring the system to immediately flush its self-applied corrections to disk to maintain consistency.

One particularly interesting bug related to the self-healing process was CVE-2023-38139. To reproduce the issue, the self-healing logic would have to be triggered a large number of times (in the case of my PoC, 65535 times) in order to cause a 32-bit integer overflow of a security descriptor refcount, and later a UAF condition. I have also abused the behavior to demonstrate WinRegLowSeverityBugs #13, in which a key with an empty name would be removed during load, freeing up a reference to a security descriptor and resulting in the refcount being equal to zero upon loading. Overall, the self-healing property of the registry is not the most critical, but one that I find quite fascinating and certainly worth keeping in mind as part of one's toolbox when researching this subsystem.

Conclusion

Congratulations on reaching the end! This post aimed to systematically explore the inner workings of the regf format, focusing on the hard requirements enforced by Windows. Due to my role and interests, I looked at the format from a strictly security-oriented angle rather than digital forensics, which is the context in which registry hives are typically considered. Hopefully, this deep dive clarifies some of the intricacies of the hive format and complements existing unofficial documentation.

Keep in mind that hives store their data in the regf files on disk, but Windows also creates multiple auxiliary kernel objects for managing and caching this data once loaded. The next post in the series will discuss these various objects, their relationships, lifecycle, and, naturally, their impact on system security. Stay tuned!