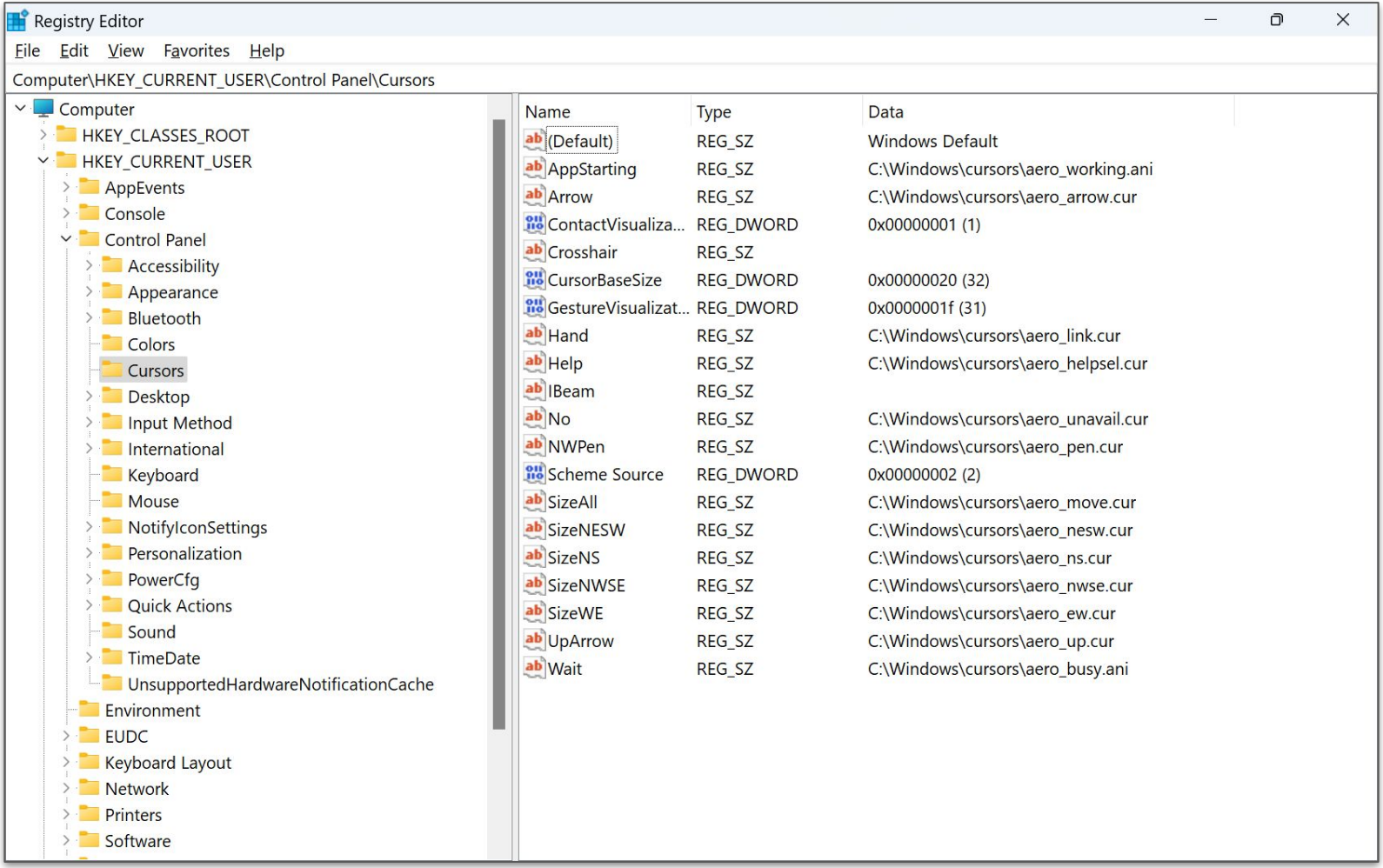
Practical Exploitation of Registry Vulnerabilities in the Windows Kernel

Mateusz Jurczyk

OffensiveCon, May 2024

The registry fundamentals

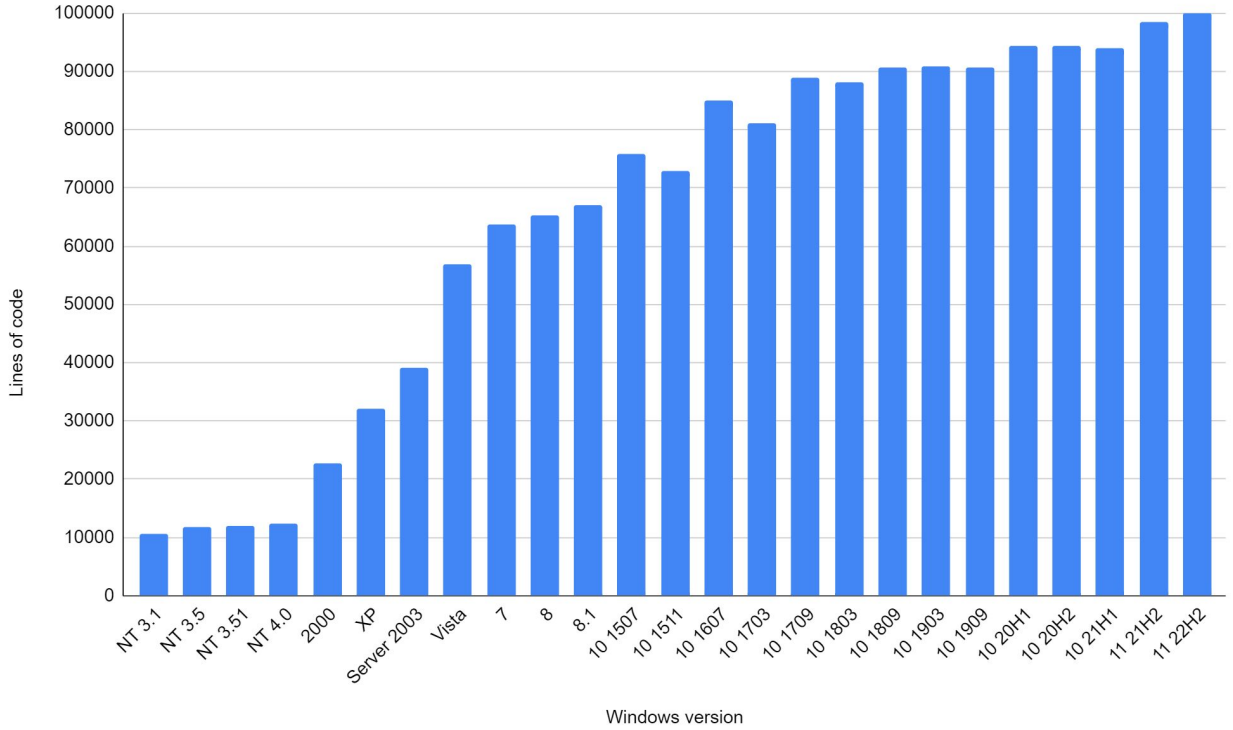
- A hierarchical database for storing system/application settings in Windows
- Essential concepts: **hives**, **keys** and **values**
- Built-in tools for management: **Regedit.exe** (GUI), **Reg.exe** (CLI)
- Documented Registry API for software developers
- Most of the implementation is in the kernel


Registry Editor
File Edit View Favorites Help
Computer\HKEY_CURRENT_USER\Control Panel\Cursors
Computer
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
AppEvents
Console
Control Panel
Accessibility
Appearance
Bluetooth
Colors
Cursors
Desktop
Input Method
International
Keyboard
Mouse
NotifyIconSettings
Personalization
PowerCfg
Quick Actions
Sound
TimeDate
UnsupportedHardwareNotificationCache
Environment
EUDC
Keyboard Layout
Network
Printers
Software
Name Type Data
(Default) REG_SZ Windows Default
AppStarting REG_SZ C:\Windows\cursors\aero_working.ani
Arrow REG_SZ C:\Windows\cursors\aero_arrow.cur
ContactVisualiza... REG_DWORD 0x00000001 (1)
Crosshair REG_SZ
CursorBaseSize REG_DWORD 0x00000020 (32)
GestureVisualizat... REG_DWORD 0x0000001f (31)
Hand REG_SZ C:\Windows\cursors\aero_link.cur
Help REG_SZ C:\Windows\cursors\aero_helpsel.cur
IBeam REG_SZ
No REG_SZ C:\Windows\cursors\aero_unavail.cur
NWPen REG_SZ C:\Windows\cursors\aero_pen.cur
Scheme Source REG_DWORD 0x00000002 (2)
SizeAll REG_SZ C:\Windows\cursors\aero_move.cur
SizeNESW REG_SZ C:\Windows\cursors\aero_nesw.cur
SizeNS REG_SZ C:\Windows\cursors\aero_ns.cur
SizeNWSE REG_SZ C:\Windows\cursors\aero_nwse.cur
SizeWE REG_SZ C:\Windows\cursors\aero_ew.cur
UpArrow REG_SZ C:\Windows\cursors\aero_up.cur
Wait REG_SZ C:\Windows\cursors\aero_busy.ani

A bit of history

- First introduced in **Windows 3.1** (1992) to replace INI files
- Current code and design directly rooted in **Windows NT 3.1** (1993) and **Windows NT 4.0** (1996)
- Started out small, then extended and improved over the next 30 years
  - Performance improvements: *faster subkey lookups, optimized key renaming*
  - Backwards compatibility: *registry virtualization*
  - New features: *big values, registry callbacks, transactions, application hives, differencing hives*
- Since **Windows Vista**, fully controlled hives can be loaded by normal users

Lines of decompiled kernel code



Registry as an attack surface: the good

👍 Kernel code accessible from unprivileged contexts opens up potential for LPE

👍 Access to sensitive data: system configuration, user credentials

👍 Error prone parts of the design: self-healing, size-bound, heavily optimized

👍 A mixture of complex C code from different eras: from 30 years ago to now

👍 A variety of potential bug classes and attack vectors

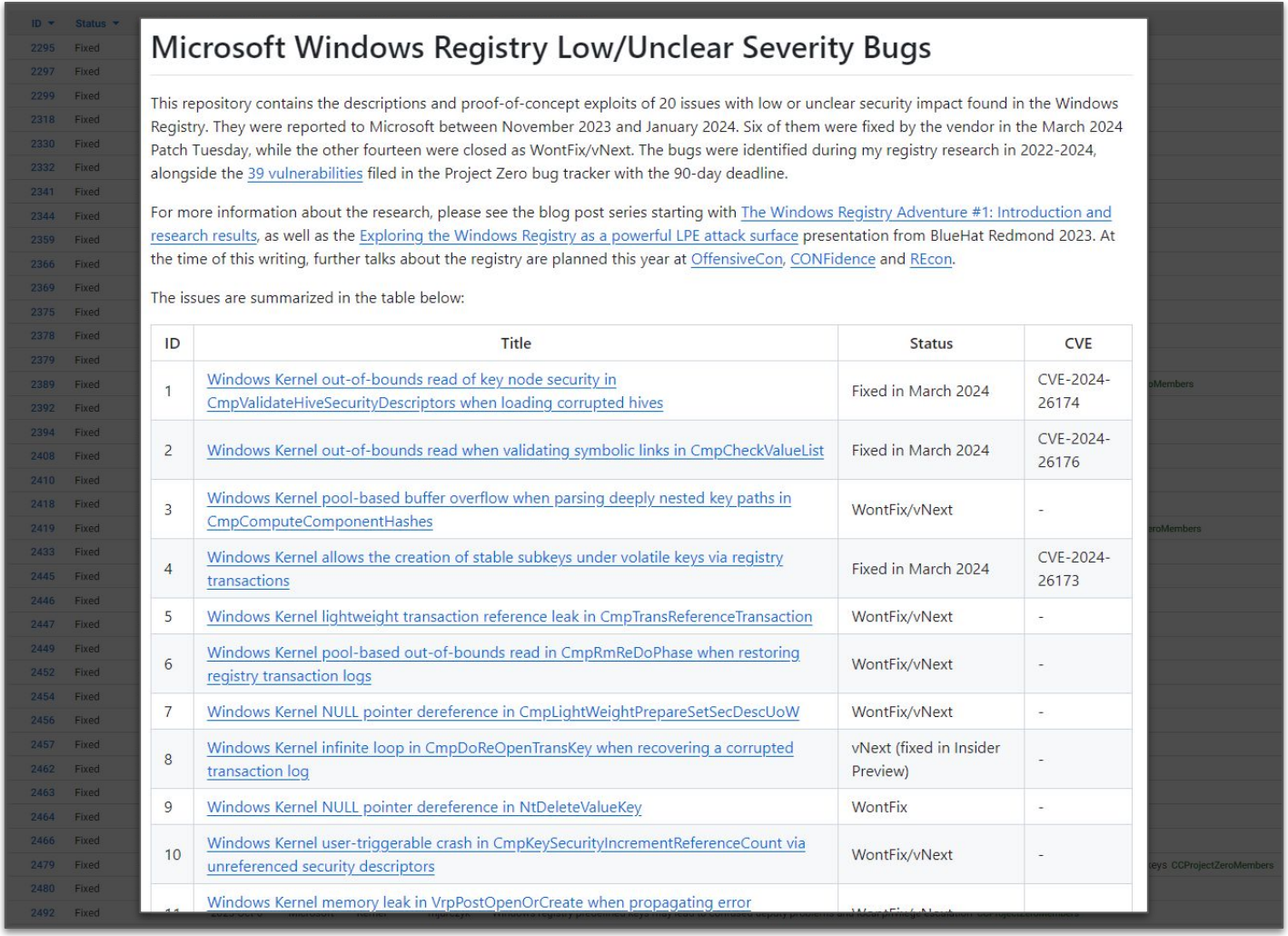
Registry as an attack surface: the bad*

- 👎 Very hard to fuzz effectively
- 👎 Source code not available, and documentation is poor for specific areas
- 👎 Public symbols incomplete, lack some type definitions
- 👎 Lots of reverse engineering required: significant time and energy investment
- 👎 Not all bugs are good, as usual

This effort

- Started in May 2022 as a hive fuzzing project
- Turned into a manual audit that lasted until December 2023 (20 months)
- Results:
  - **39 issues** reported in the Project Zero bug tracker (under a 90 day deadline)
  - **20 issues** reported outside the tracker (no deadline, low/unclear severity)
  - **= 50 CVEs** assigned by Microsoft across 15 monthly bulletins
- A variety of: attack targets, entry points, end results, root causes, types of corrupted memory, exploitation time, ...

ID	Status	Restrict	Reported	Vendor	Product	Finder	Summary + Labels
2295	Fixed	---	2022-May-11	Microsoft	Kernel	mjurczyk	Windows Kernel use-after-free due to refcount overflow in registry hive security descriptors CCProjectZeroMembers
2297	Fixed	---	2022-May-17	Microsoft	Kernel	mjurczyk	Windows Kernel invalid read/write due to unchecked Blink cell index in root security descriptor CCProjectZeroMembers
2299	Fixed	---	2022-May-20	Microsoft	Kernel	mjurczyk	Windows Kernel multiple memory problems when handling incorrectly formatted security descriptors in registry hives CCProjectZeroMembers
2318	Fixed	---	2022-Jun-22	Microsoft	Kernel	mjurczyk	Windows Kernel integer overflows in registry subkey lists leading to memory corruption CCProjectZeroMembers
2330	Fixed	---	2022-Jul-8	Microsoft	Kernel	mjurczyk	Windows Kernel registry use-after-free due to bad handling of failed reallocations under memory pressure CCProjectZeroMembers
2332	Fixed	---	2022-Jul-11	Microsoft	Kernel	mjurczyk	Windows Kernel memory corruption due to type confusion of subkey index leaves in registry hives CCProjectZeroMembers
2341	Fixed	---	2022-Aug-3	Microsoft	Kernel	mjurczyk	Windows Kernel multiple memory corruption issues when operating on very long registry paths CCProjectZeroMembers
2344	Fixed	---	2022-Aug-5	Microsoft	Kernel	mjurczyk	Windows Kernel out-of-bounds reads and other issues when operating on long registry key and value names CCProjectZeroMembers
2359	Fixed	---	2022-Sep-22	Microsoft	Kernel	mjurczyk	Windows Kernel use-after-free due to bad handling of predefined keys in NtNotifyChangeMultipleKeys CCProjectZeroMembers
2366	Fixed	---	2022-Oct-6	Microsoft	Kernel	mjurczyk	Windows Kernel memory corruption due to insufficient handling of predefined keys in registry virtualization CCProjectZeroMembers
2369	Fixed	---	2022-Oct-13	Microsoft	Kernel	mjurczyk	Windows Kernel use-after-free due to dangling registry link node under paged pool memory pressure CCProjectZeroMembers
2375	Fixed	---	2022-Oct-25	Microsoft	Kernel	mjurczyk	Windows Kernel multiple issues in the key replication feature of registry virtualization CCProjectZeroMembers
2378	Fixed	---	2022-Oct-31	Microsoft	Kernel	mjurczyk	Windows Kernel registry SID table poisoning leading to bad locking and other issues CCProjectZeroMembers
2379	Fixed	---	2022-Nov-2	Microsoft	Kernel	mjurczyk	Windows Kernel allows deletion of keys in virtualizable hives with KEY_READ and KEY_SET_VALUE access rights CCProjectZeroMembers
2389	Fixed	---	2022-Nov-30	Microsoft	Kernel	mjurczyk	Windows Kernel registry virtualization incompatible with transactions, leading to inconsistent hive state and memory corruption CCProjectZeroMembers
2392	Fixed	---	2022-Dec-7	Microsoft	Kernel	mjurczyk	Windows Kernel multiple issues with subkeys of transactionally renamed registry keys CCProjectZeroMembers
2394	Fixed	---	2022-Dec-14	Microsoft	Kernel	mjurczyk	Windows Kernel multiple issues in the prepare/commit phase of a transactional registry key rename CCProjectZeroMembers
2408	Fixed	---	2023-Jan-13	Microsoft	Kernel	mjurczyk	Windows Kernel insufficient validation of new registry key names in transacted NtRenameKey CCProjectZeroMembers
2410	Fixed	---	2023-Jan-19	Microsoft	Kernel	mjurczyk	Windows Kernel CmpCleanupLightWeightPrepare registry security descriptor refcount leak leading to UAF CCProjectZeroMembers
2418	Fixed	---	2023-Jan-31	Microsoft	Kernel	mjurczyk	Windows Kernel disclosure of kernel pointers and uninitialized memory through registry KTM transaction log files CCProjectZeroMembers
2419	Fixed	---	2023-Feb-2	Microsoft	Kernel	mjurczyk	Windows Kernel out-of-bounds reads when operating on invalid registry paths in CmpDoReDoCreateKey/CmpDoReOpenTransKey CCProjectZeroMembers
2433	Fixed	---	2023-Mar-7	Microsoft	Kernel	mjurczyk	Windows Kernel KTM registry transactions may have non-atomic outcomes CCProjectZeroMembers
2445	Fixed	---	2023-Apr-19	Microsoft	Kernel	mjurczyk	Windows Kernel arbitrary read by accessing predefined keys through differencing hives CCProjectZeroMembers
2446	Fixed	---	2023-Apr-20	Microsoft	Kernel	mjurczyk	Windows Kernel may reference unbacked layered keys through registry virtualization CCProjectZeroMembers
2447	Fixed	---	2023-Apr-27	Microsoft	Kernel	mjurczyk	Windows Kernel may reference rolled-back transacted keys through differencing hives CCProjectZeroMembers
2449	Fixed	---	2023-May-2	Microsoft	Kernel	mjurczyk	Windows Kernel renaming layered keys doesn't reference count security descriptors, leading to UAF CCProjectZeroMembers
2452	Fixed	---	2023-May-10	Microsoft	Kernel	mjurczyk	Windows Kernel CmDeleteLayeredKey may delete predefined tombstone keys, leading to security descriptor UAF CCProjectZeroMembers
2454	Fixed	---	2023-May-15	Microsoft	Kernel	mjurczyk	Windows Kernel out-of-bounds reads due to an integer overflow in registry .LOG file parsing CCProjectZeroMembers
2456	Fixed	---	2023-May-22	Microsoft	Kernel	mjurczyk	Windows Kernel partial success of registry hive log recovery may lead to inconsistent state and memory corruption CCProjectZeroMembers
2457	Fixed	---	2023-May-31	Microsoft	Kernel	mjurczyk	Windows Kernel doesn't reset security cache during self-healing, leading to refcount overflow and UAF CCProjectZeroMembers
2462	Fixed	---	2023-Jun-26	Microsoft	Kernel	mjurczyk	Windows Kernel passes user-mode pointers to registry callbacks, leading to race conditions and memory corruption CCProjectZeroMembers
2463	Fixed	---	2023-Jun-27	Microsoft	Kernel	mjurczyk	Windows Kernel paged pool memory disclosure in VrpPostEnumerateKey CCProjectZeroMembers
2464	Fixed	---	2023-Jun-27	Microsoft	Kernel	mjurczyk	Windows Kernel out-of-bounds reads and paged pool memory disclosure in VrpUpdateKeyInformation CCProjectZeroMembers
2466	Fixed	---	2023-Jul-7	Microsoft	Kernel	mjurczyk	Windows Kernel containerized registry escape through integer overflows in VrpBuildKeyPath and other weaknesses CCProjectZeroMembers
2479	Fixed	---	2023-Aug-10	Microsoft	Kernel	mjurczyk	Windows Kernel time-of-check/time-of-use issue in verifying layered key security may lead to information disclosure from privileged registry keys CCProjectZeroMembers
2480	Fixed	---	2023-Aug-22	Microsoft	Kernel	mjurczyk	Windows Kernel bad locking in registry virtualization leads to race conditions CCProjectZeroMembers
2492	Fixed	---	2023-Oct-6	Microsoft	Kernel	mjurczyk	Windows registry predefined keys may lead to confused deputy problems and local privilege escalation CCProjectZeroMembers



Microsoft Windows Registry Low/Unclear Severity Bugs

This repository contains the descriptions and proof-of-concept exploits of 20 issues with low or unclear security impact found in the Windows Registry. They were reported to Microsoft between November 2023 and January 2024. Six of them were fixed by the vendor in the March 2024 Patch Tuesday, while the other fourteen were closed as WontFix/vNext. The bugs were identified during my registry research in 2022-2024, alongside the 39 vulnerabilities filed in the Project Zero bug tracker with the 90-day deadline.

For more information about the research, please see the blog post series starting with The Windows Registry Adventure #1: Introduction and research results, as well as the Exploring the Windows Registry as a powerful LPE attack surface presentation from BlueHat Redmond 2023. At the time of this writing, further talks about the registry are planned this year at OffensiveCon, CONFidence and REcon.

The issues are summarized in the table below:

ID	Title	Status	CVE
1	Windows Kernel out-of-bounds read of key node security in CmpValidateHiveSecurityDescriptors when loading corrupted hives	Fixed in March 2024	CVE-2024-26174
2	Windows Kernel out-of-bounds read when validating symbolic links in CmpCheckValueList	Fixed in March 2024	CVE-2024-26176
3	Windows Kernel pool-based buffer overflow when parsing deeply nested key paths in CmpComputeComponentHashes	WontFix/vNext	-
4	Windows Kernel allows the creation of stable subkeys under volatile keys via registry transactions	Fixed in March 2024	CVE-2024-26173
5	Windows Kernel lightweight transaction reference leak in CmpTransReferenceTransaction	WontFix/vNext	-
6	Windows Kernel pool-based out-of-bounds read in CmpRmReDoPhase when restoring registry transaction logs	WontFix/vNext	-
7	Windows Kernel NULL pointer dereference in CmpLightWeightPrepareSetSecDescUoW	WontFix/vNext	-
8	Windows Kernel infinite loop in CmpDoReOpenTransKey when recovering a corrupted transaction log	vNext (fixed in Insider Preview)	-
9	Windows Kernel NULL pointer dereference in NtDeleteValueKey	WontFix	-
10	Windows Kernel user-triggerable crash in CmpKeySecurityIncrementReferenceCount via unreferenced security descriptors	WontFix/vNext	-
11	Windows Kernel memory leak in VrpPostOpenOrCreate when propagating error		

Microsoft Windows Registry Low/Unclear Severity Bugs

Thursday, April 18, 2024

The Windows Registry Adventure #1: Introduction and research results

Posted by Mateusz Jurczyk, Google Project Zero

In the 20-month period between May 2022 and December 2023, I thoroughly audited the Windows Registry in search of local privilege escalation bugs. It all started unexpectedly: I was in the process of developing a coverage-based Windows kernel fuzzer based on the Bochs x86 emulator (one of my favorite tools for security research: see Bochspwn, Bochspwn Reloaded, and my earlier font fuzzing infrastructure), and needed some binary formats to test it on. My first pick were PE files: they are very popular in the Windows environment, which makes it easy to create an initial corpus of input samples, and a basic fuzzing harness is equally easy to develop with just a single GetFileVersionInfoSizeW API call. The test was successful: even though I had previously fuzzed PE files in 2019, the new element of code coverage guidance allowed me to discover a completely new bug: issue #2281.

For my next target, I chose the Windows registry. That's because arbitrary registry hives can be loaded from disk without any special privileges via the RegLoadAppKey API (since Windows Vista). The hives use a binary format and are fully parsed in the kernel, making them a noteworthy local attack surface. Furthermore, I was also somewhat familiar with basic harnessing of the registry, having fuzzed it in 2016 together with James Forshaw. Once again, the code coverage support proved useful, leading to the discovery of issue #2299. But when I started to perform a root cause analysis of the bug, I realized that:

Exploitation

- Let's focus on the *offensive* angle
- Which bug(s) shall we exploit for LPE?

Exploitation

- Let's focus on the *offensive* angle
- Which bug(s) shall we exploit for LPE?
  - Logic issues: usually the easiest and most reliable to exploit, but not many good candidates

Exploitation

- Let's focus on the *offensive* angle
- Which bug(s) shall we exploit for LPE?
  - ~~Logic issues: usually the easiest and most reliable to exploit, but not many good candidates~~
  - Pool-based memory corruption: universal exploitation techniques apply, not that interesting

Exploitation

- Let's focus on the *offensive* angle
- Which bug(s) shall we exploit for LPE?
  - ~~Logic issues: usually the easiest and most reliable to exploit, but not many good candidates~~
  - ~~Pool-based memory corruption: universal exploitation techniques apply, not that interesting~~
  - **Hive-based memory corruption**: an unexplored class of issues specific to the registry

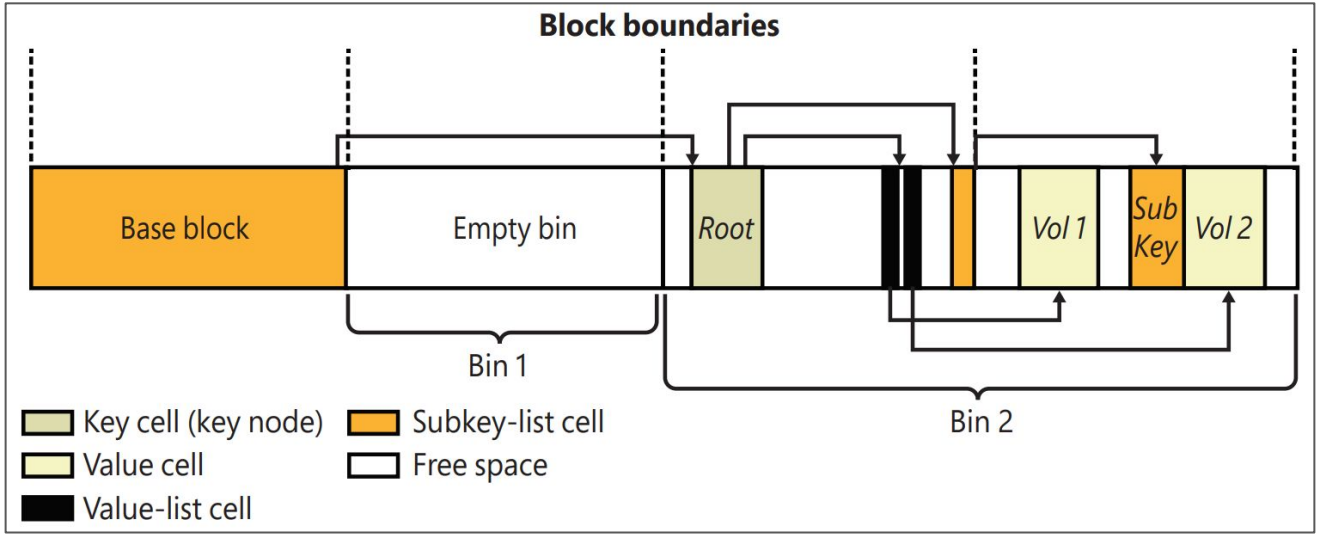
Exploitation

- Let's focus on the *offensive* angle
- Which bug(s) shall we exploit for LPE?
  - ~~Logic issues: usually the easiest and most reliable to exploit, but not many good candidates~~
  - ~~Pool-based memory corruption: universal exploitation techniques apply, not that interesting~~
  - **Hive-based memory corruption**: an unexplored class of issues specific to the registry
- But what gets corrupted, exactly?

Memory-mapped hives

- Hives are encoded in the "regf" format v1.3 - v1.6 (backwards compatible)
- When loaded, they are directly mapped in memory
  - Usually via file-backed sections in the user space of a thin "Registry" process
  - In-memory and on-disk copies are continuously synchronized
- High level structure:
  - Starts with 4 KiB header a.k.a. **base block**
  - Followed by one or more 4 KiB-aligned **bins**
    - Each bin starts with a 32-byte long header
    - The rest of the bin is filled completely with **cells** of arbitrary length, either allocated (i.e. in use) or free

Registry hive layout



source: Windows Internals, 7th Edition, Part 2 (A. Allievi, A. Ionescu, M. Russinovich, D. Solomon)

Cells – the heart of the hive

- The base block and bin headers are mildly interesting, the vital data is in the cells
- A total of **~8-10 cell types** (depending on how you count)
- Their structure layouts are publicly available in the kernel symbols, see:
  - `CM_KEY_NODE`
  - `CM_KEY_VALUE`
  - `CM_KEY_INDEX`
  - `CM_KEY_SECURITY`
  - `CM_BIG_DATA`
- They reference each other using *cell indexes*, i.e. 32-bit offsets in the hive

Key node example

Structure layout:

```
kd> dt _CM_KEY_NODE
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
   +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
```

Example key node data:

```
00000000   6E 6B                    nk
00000002   20 00                     .
00000004   25 24 0A ED 07 4E DA 01  %$.í.NÚ.
0000000C   00                       .
0000000D   00                       .

0000000E   00 00                    ..
00000010   FF FF FF FF              ....
00000014   02 00 00 00 00 00 00 00  ........
0000001C   B0 01 00 00 FF FF FF FF  °.......
00000024   00 00 00 00 FF FF FF FF  ........
0000002C   78 00 00 00              x...
00000030   FF FF FF FF              ....
00000034   0E 00                    ..
00000036   00                       .

00000037   00                       .
00000038   00 00 00 00              ....
0000003C   00 00 00 00              ....
00000040   00 00 00 00              ....
00000044   00 00 00 00              ....
00000048   04 00                    ..
0000004A   00 00                    ..
0000004C   52 4F 4F 54              ROOT
```

Key node: relevant parts

Structure layout:

```
kd> dt _CM_KEY_NODE
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
   +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
```

Example key node data:

```
00000000   6E 6B                    nk
00000002   20 00                     .
00000004   25 24 0A ED 07 4E DA 01  %$.í.NÚ.
0000000C   00                       .
0000000D   00                       .


0000000E   00 00                    ..
00000010   FF FF FF FF              ....
00000014   02 00 00 00 00 00 00 00  ........
0000001C   B0 01 00 00 FF FF FF FF  °...ÿÿÿÿ
00000024   00 00 00 00 FF FF FF FF  ....ÿÿÿÿ
0000002C   78 00 00 00              x...
00000030   FF FF FF FF              ....
00000034   0E 00                    ..
00000036   00                       .

00000037   00                       .
00000038   00 00 00 00              ....
0000003C   00 00 00 00              ....
00000040   00 00 00 00              ....
00000044   00 00 00 00              ....
00000048   04 00                    ..
0000004A   00 00                    ..
0000004C   52 4F 4F 54              ROOT
```

Key node: cell indexes

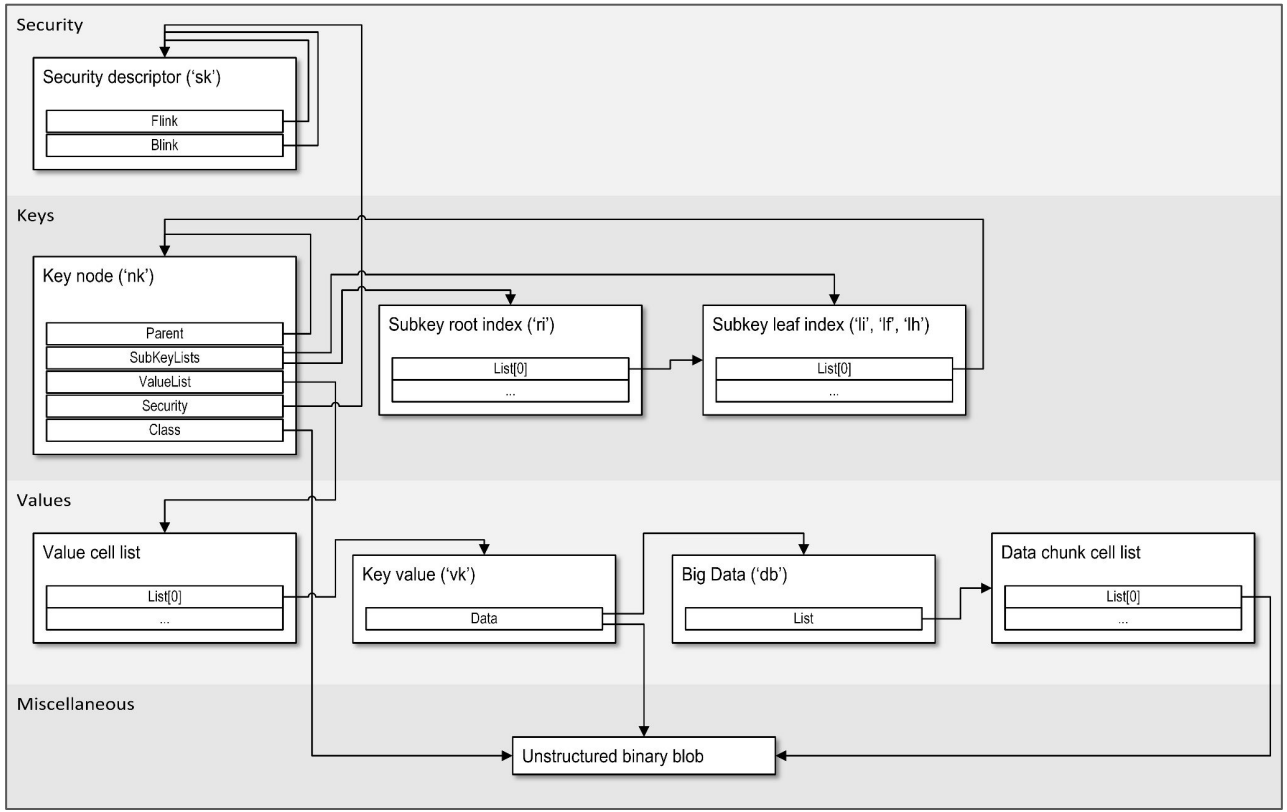
Structure layout:

```
kd> dt _CM_KEY_NODE
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
   +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
```

Example key node data:

```
00000000   6E 6B                    nk
00000002   20 00                     .
00000004   25 24 0A ED 07 4E DA 01  %$.í.NÚ.
0000000C   00                       .
0000000D   00                       .


0000000E   00 00                    ..
00000010   FF FF FF FF              ....
00000014   02 00 00 00 00 00 00 00  ........
0000001C   B0 01 00 00 FF FF FF FF  °.......
00000024   00 00 00 00 FF FF FF FF  ........
0000002C   78 00 00 00              x...
00000030   FF FF FF FF              ....
00000034   0E 00                    ..
00000036   00                       .

00000037   00                       .
00000038   00 00 00 00              ....
0000003C   00 00 00 00              ....
00000040   00 00 00 00              ....
00000044   00 00 00 00              ....
00000048   04 00                    ..
0000004A   00 00                    ..
0000004C   52 4F 4F 54              ROOT
```

Relationships between cells



State consistency guarantees

- On load, the kernel verifies the overall correctness of the hive
  - Tries to fix whatever it can in the process
  - Deletes / unlinks objects beyond repair
- At runtime, the kernel keeps the hive structure valid at all times
- This is a security-critical invariant, and breaking it may lead to interesting side effects...

How to corrupt a hive?

- Cells are chunks of binary data handled in C, so things can go wrong
  - Bad reference counting of security descriptors → UAF
  - Dangling cell index stored somewhere after freeing the cell → UAF
  - Unchecked cell index → arbitrary read/write of raw hive data
  - Bugs involving variable sized arrays (subkeys, values) → buffer overflows
  - Allowing other inconsistencies in the hive (e.g. invalid size specifiers) → out-of-bounds accesses

CVE-2022-34707

- My first manually discovered registry bug
- Can you spot the problem?

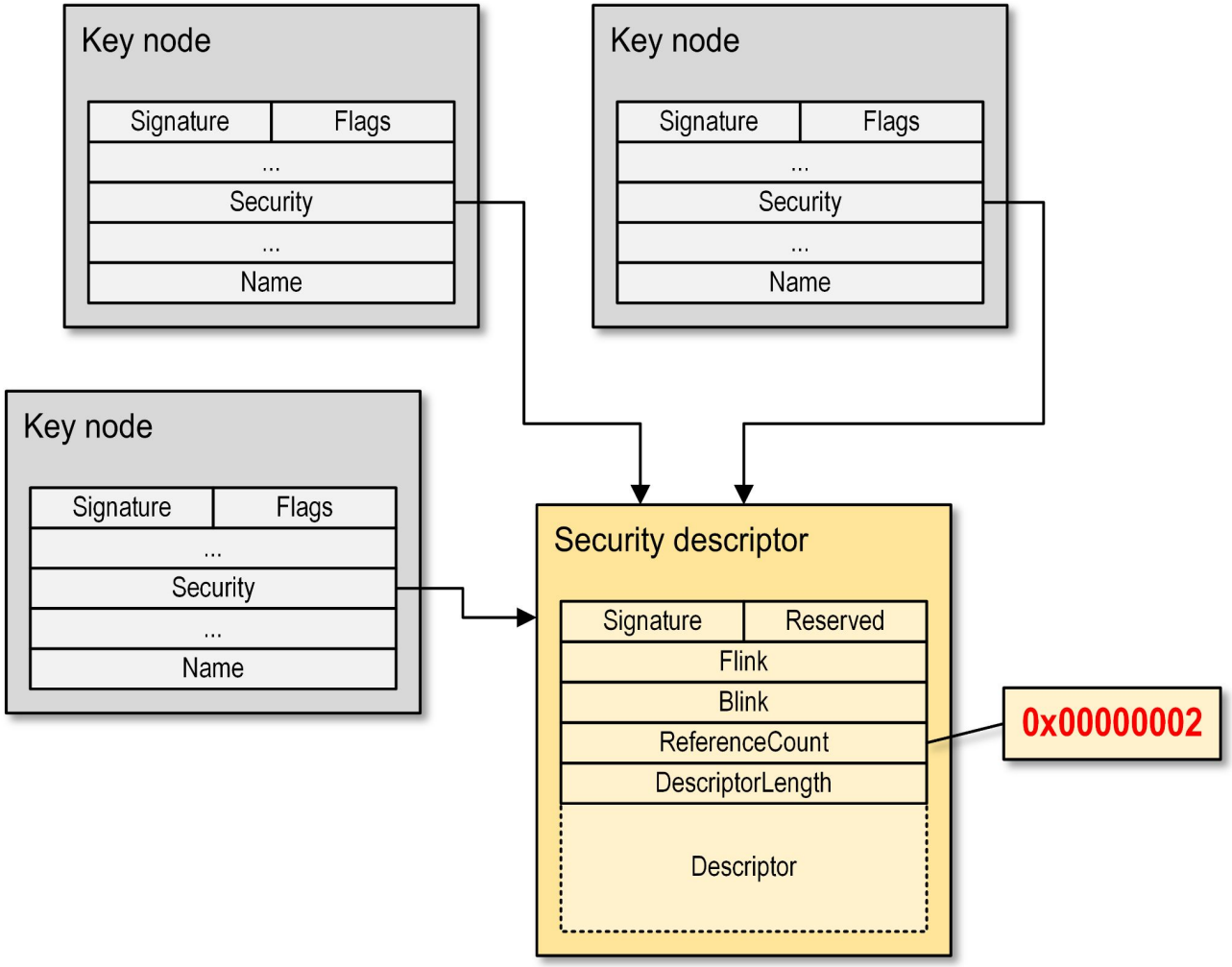
```
BOOLEAN CmpCheckAndFixSecurityCellsRefcount(CMHIVE *CmHive) {
  ...
  for (int i = 0; i < CmHive->SecurityCount; i++) {
    CM_KEY_SECURITY_CACHE_ENTRY *CacheEntry = &CmHive->SecurityCache[i];
    CM_KEY_SECURITY *SecurityNode           = CmHive->Hive.GetCellRoutine(CmHive, CacheEntry->Cell);

    if (SecurityNode->ReferenceCount < CacheEntry->CachedSecurity->RealRefCount) {
      SecurityNode->ReferenceCount = CacheEntry->CachedSecurity->RealRefCount;
    }
  }
  ...
}
```

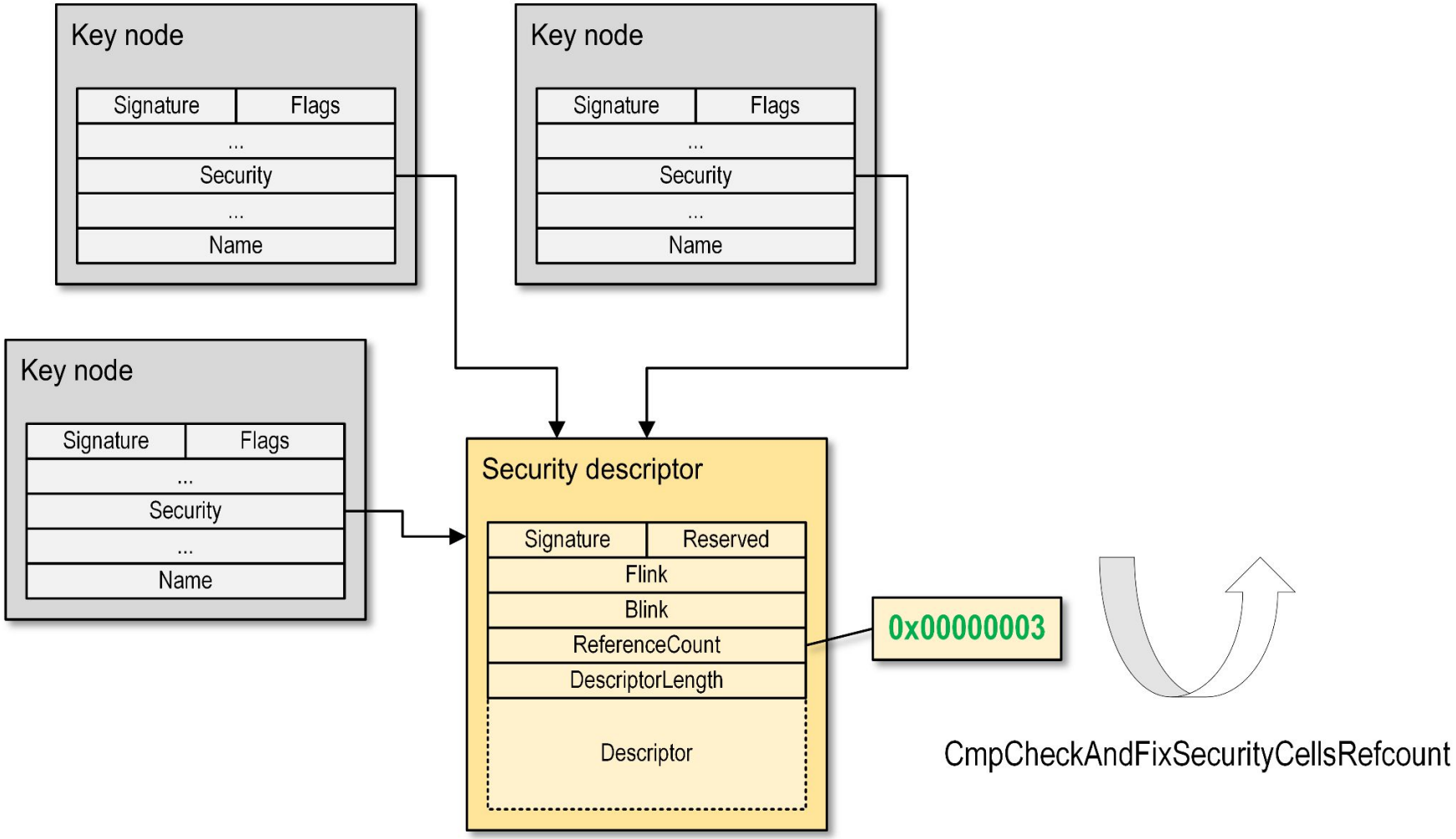
CVE-2022-34707

- My first manually discovered registry bug
- Can you spot the problem?

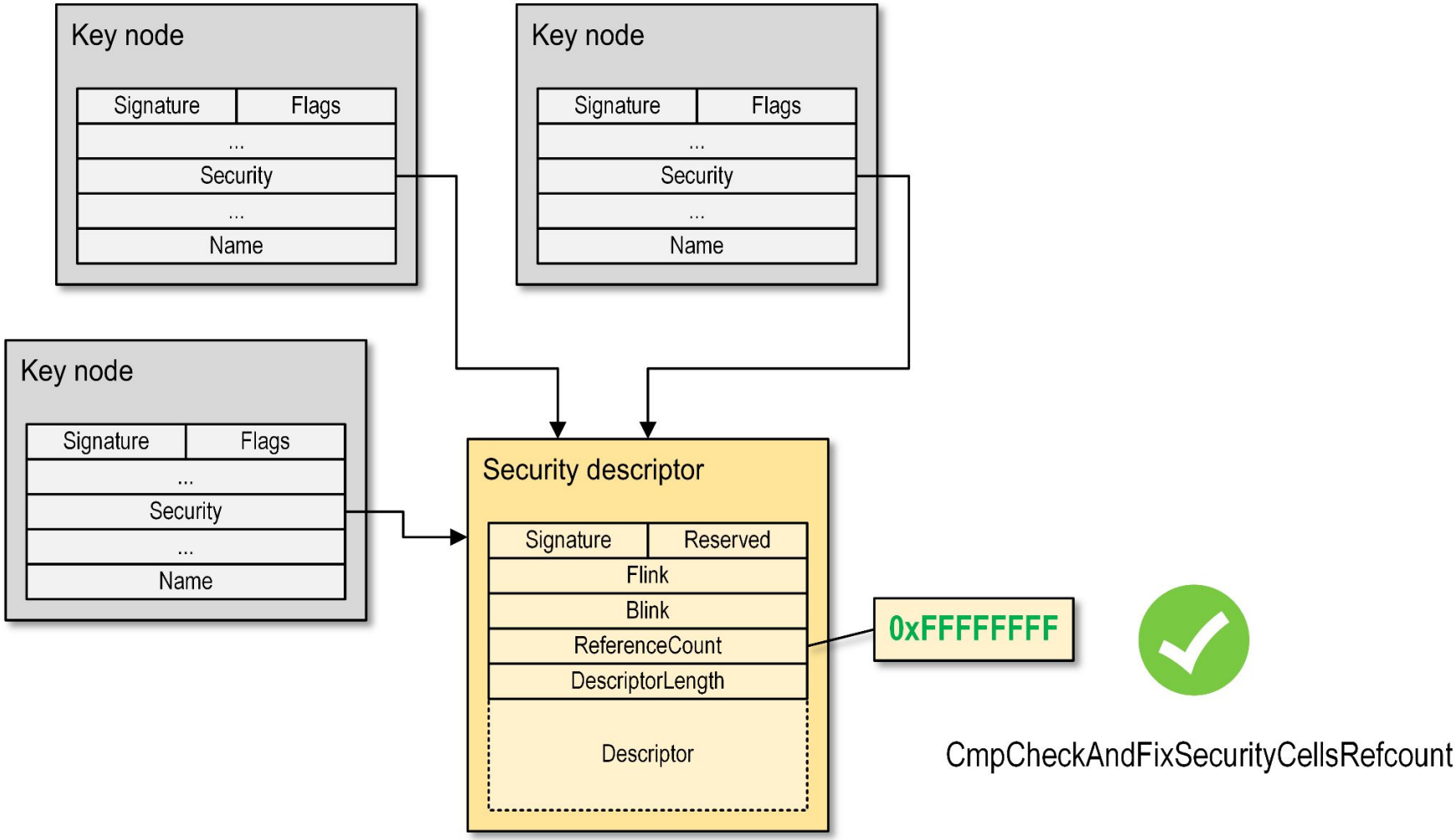
Only fixing inadequately small refcounts?

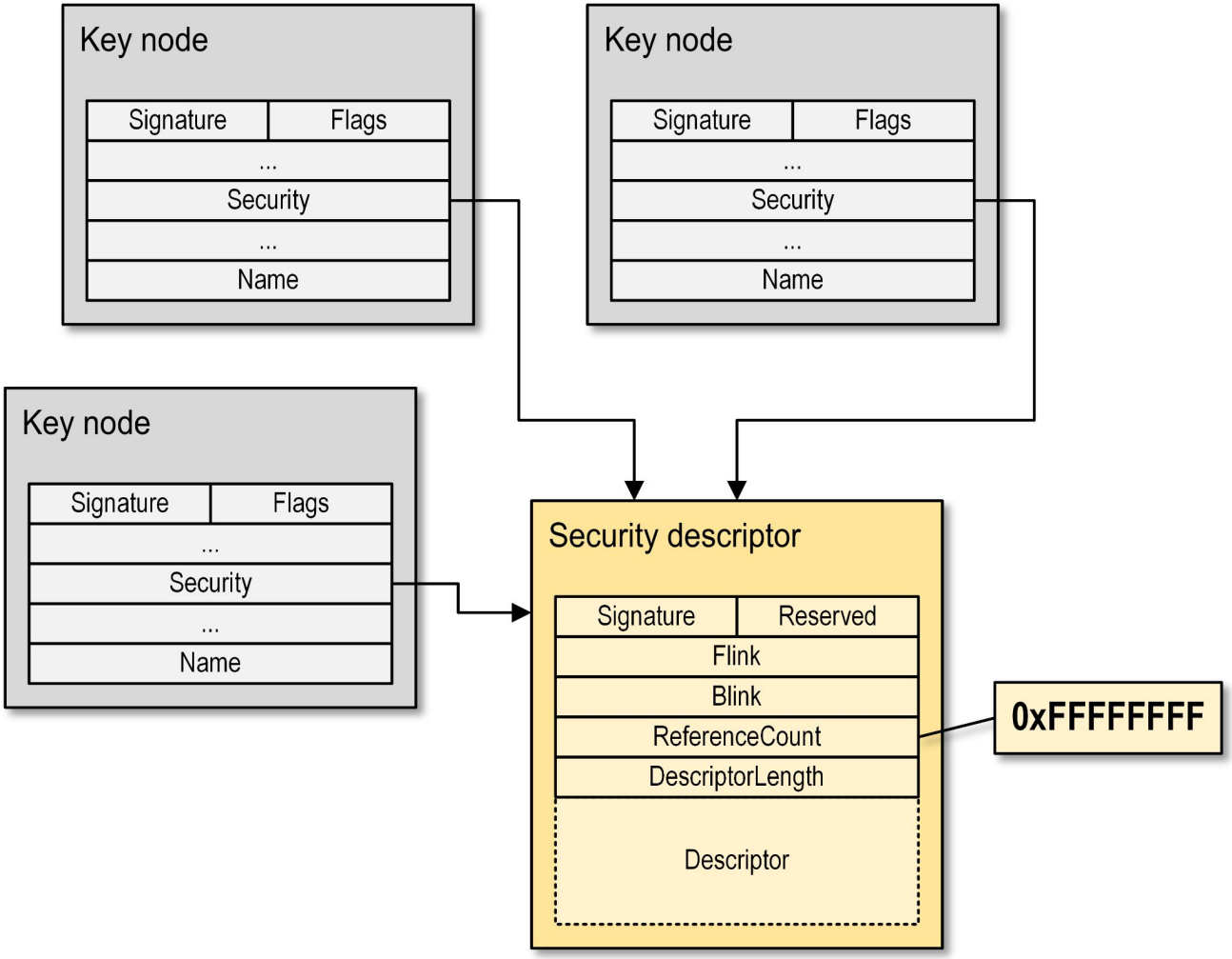
```
BOOLEAN CmpCheckAndFixSecurityCellsRefcount(CMHIVE *CmHive) {
  ...
  for (int i = 0; i < CmHive->SecurityCount; i++) {
    CM_KEY_SECURITY_CACHE_ENTRY *CacheEntry = &CmHive->SecurityCache[i];
    CM_KEY_SECURITY *SecurityNode           = CmHive->Hive.GetCellRoutine(CmHive, CacheEntry->Cell);

    if (SecurityNode->ReferenceCount < CacheEntry->CachedSecurity->RealRefCount) {
      SecurityNode->ReferenceCount = CacheEntry->CachedSecurity->RealRefCount;
    }
  }
  ...
}
```

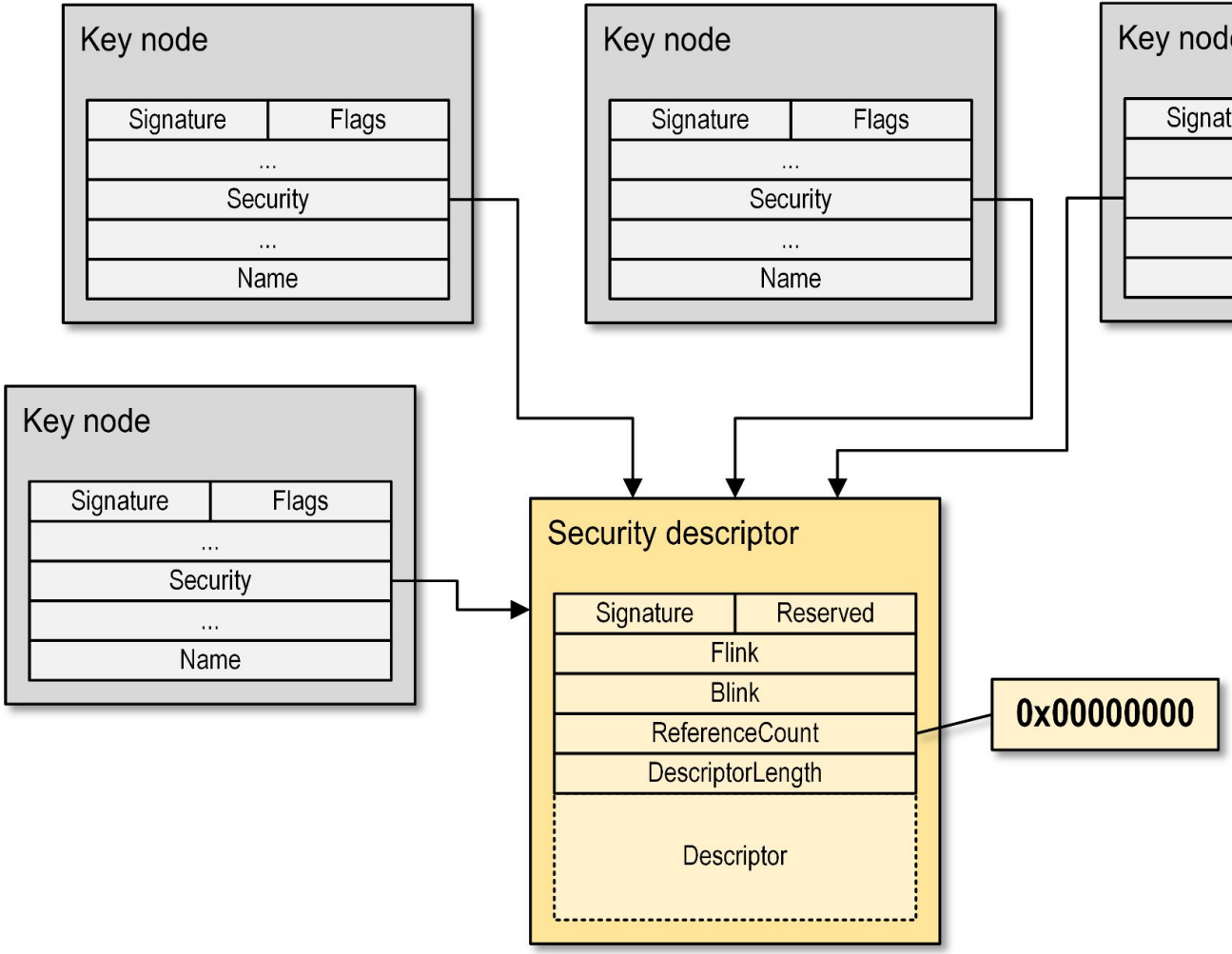

Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0x00000002

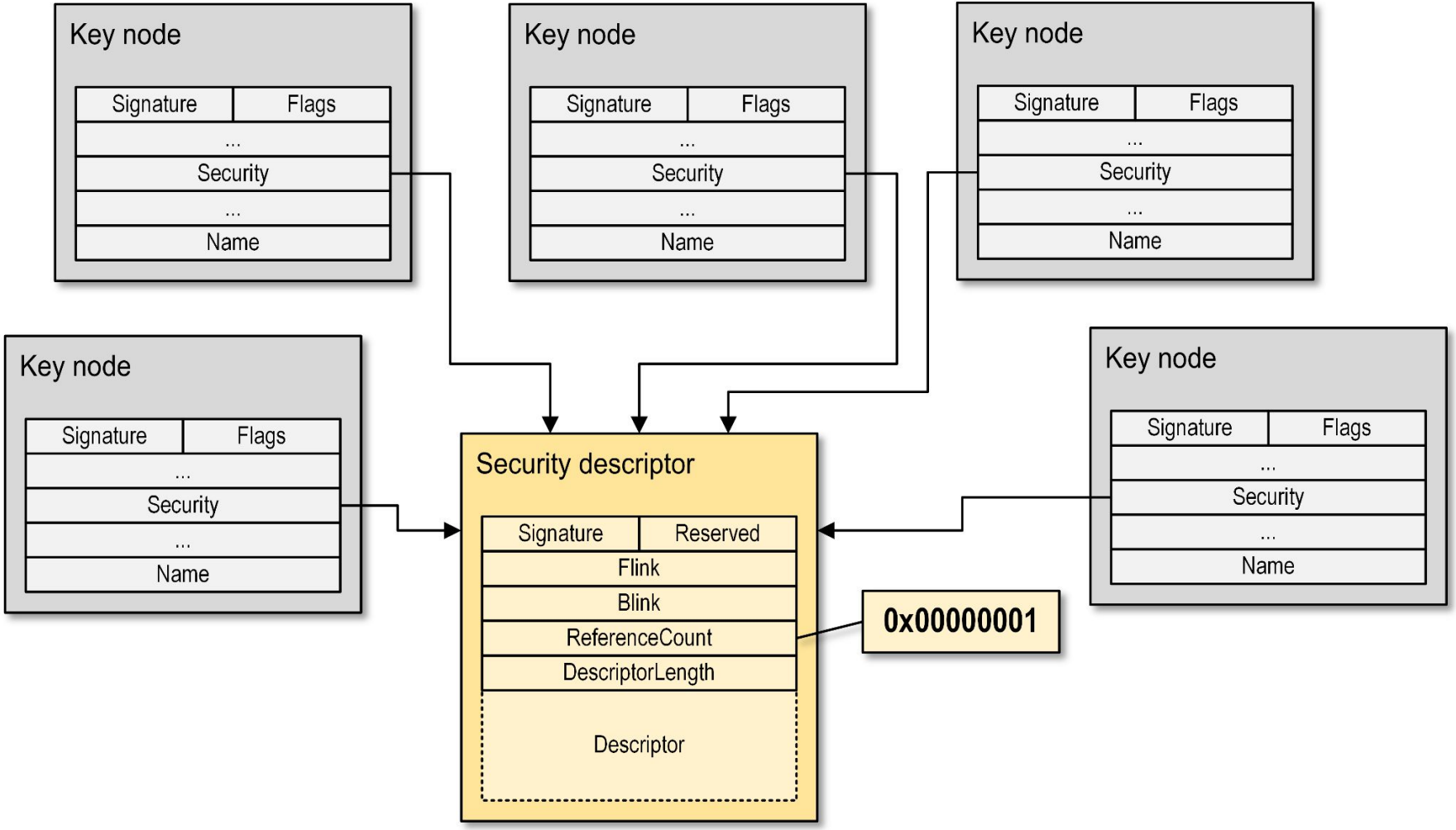

Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0x00000003
CmpCheckAndFixSecurityCellsRefcount

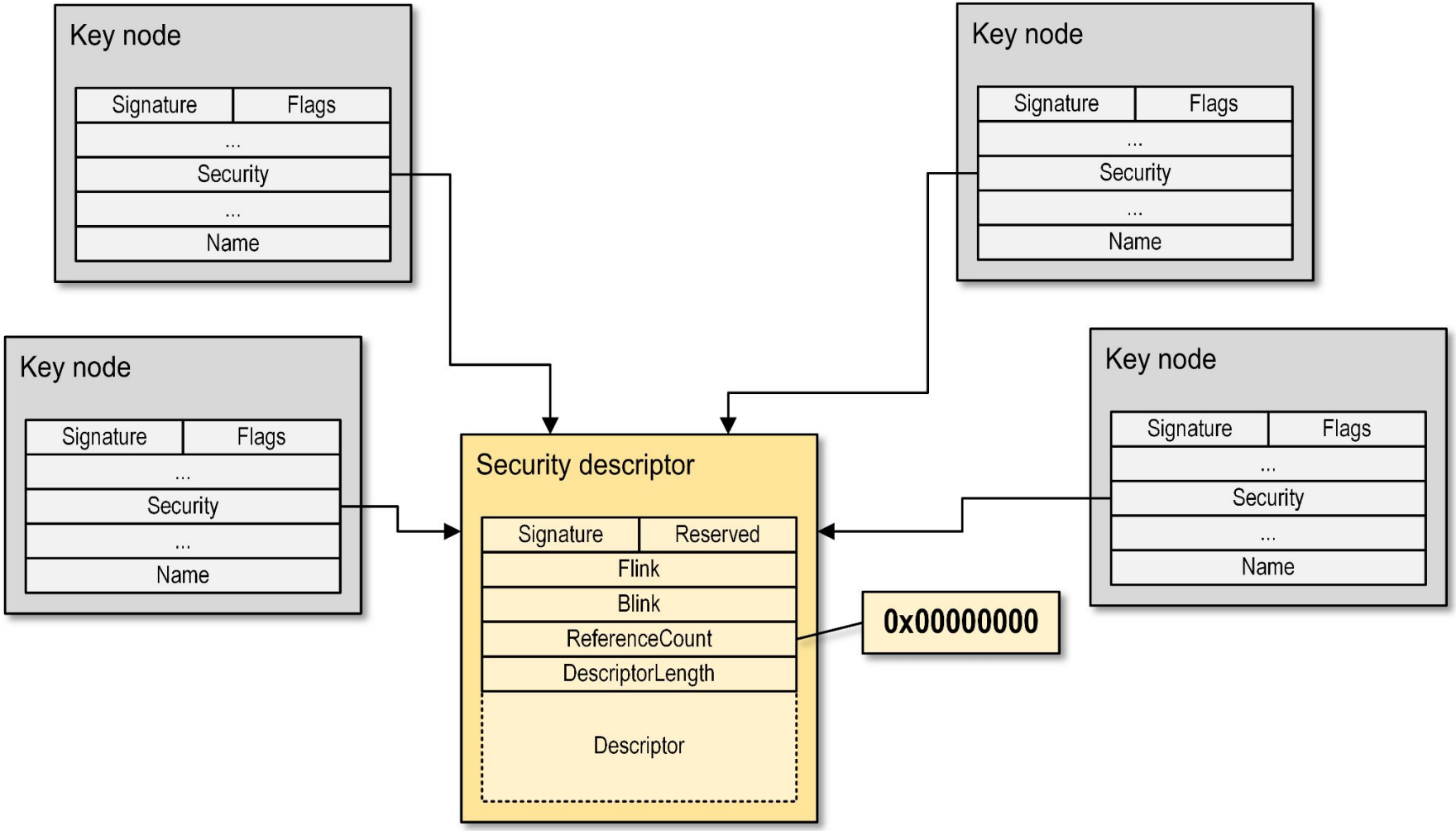
But...

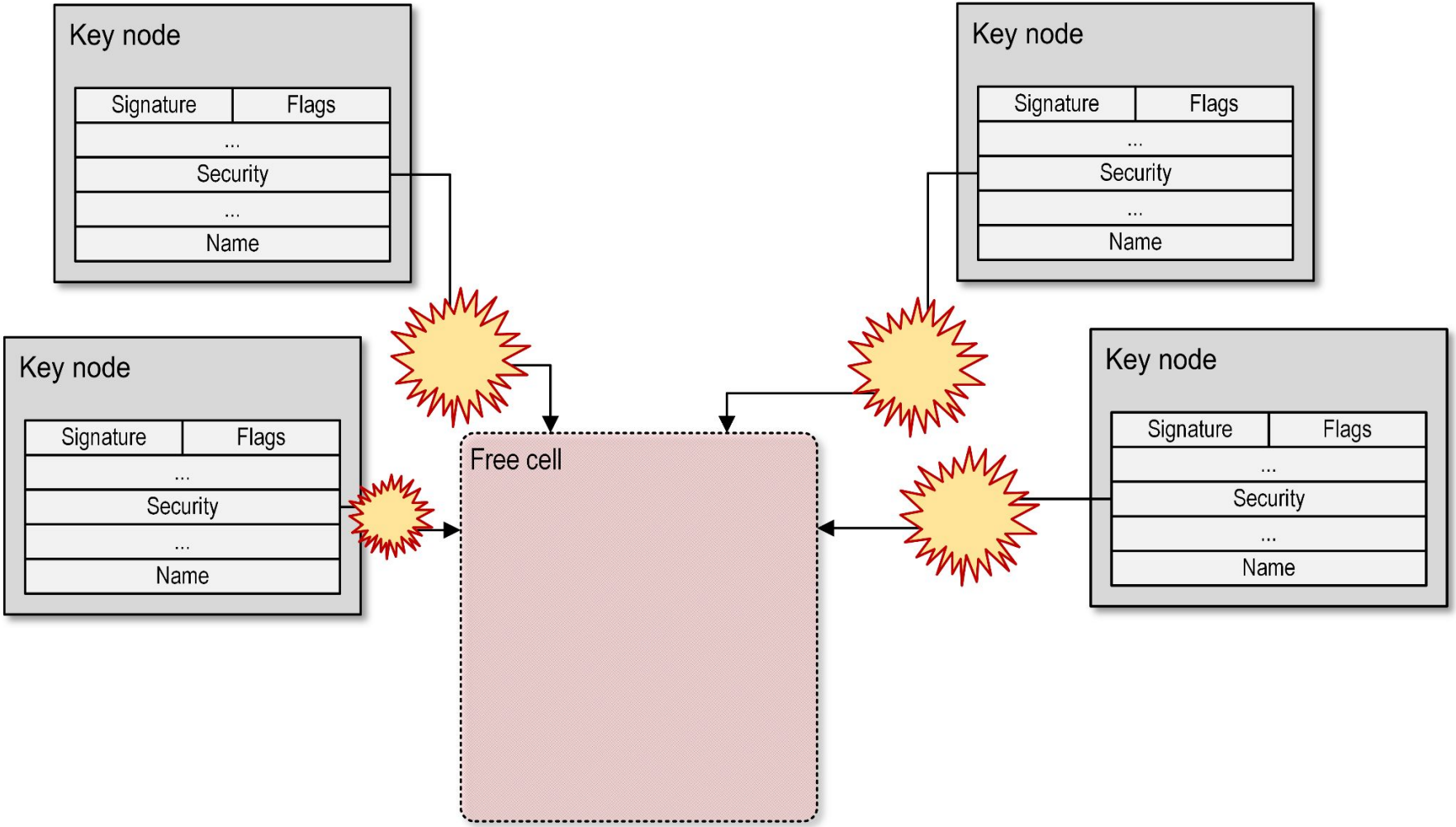

Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0xFFFFFFFF
CmpCheckAndFixSecurityCellsRefcount


Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0xFFFFFFFF


Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0x00000000


Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0x00000001


Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Security descriptor
Signature
Reserved
Flink
Blink
ReferenceCount
DescriptorLength
Descriptor
0x00000000


Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Key node
Signature
Flags
...
Security
...
Name
Free cell

CVE-2022-34707

- A clean, convenient UAF primitive available on demand
- Around ~15 equivalent bugs found in total
- What are the next exploitation steps? Follow along 🙂

Exploitation Opening:

The Hive Allocator

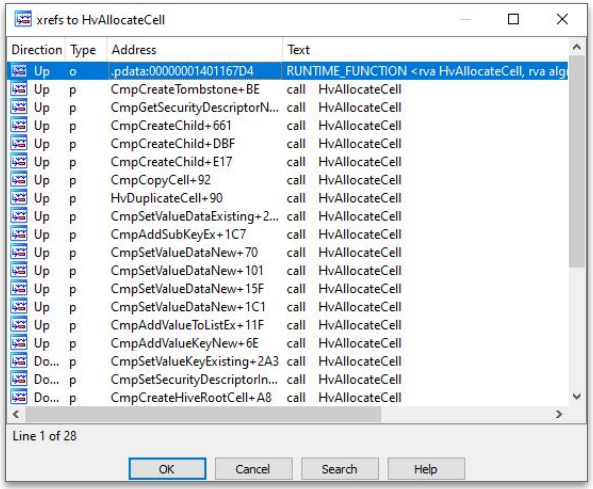
The internal allocator interface

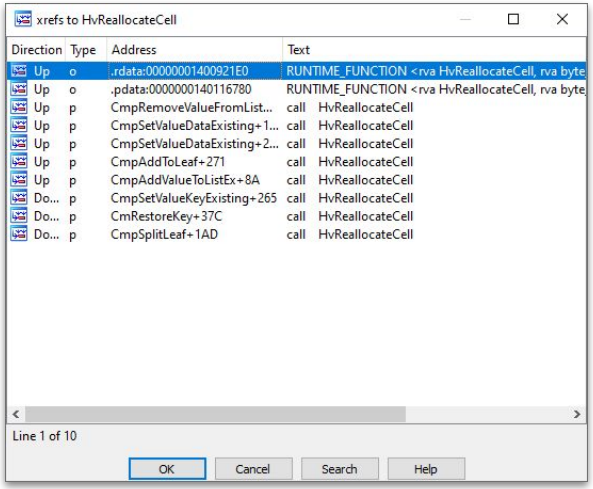
```
NTSTATUS HvAllocateCell(
    HHIVE *Hive,
    DWORD Size,
    HSTORAGE_TYPE StorageType,
    HCELL_INDEX *OutCellIndex,
    CELL_DATA **OutCellPointer,
    HV_GET_CELL_CONTEXT *CellContext
);
```

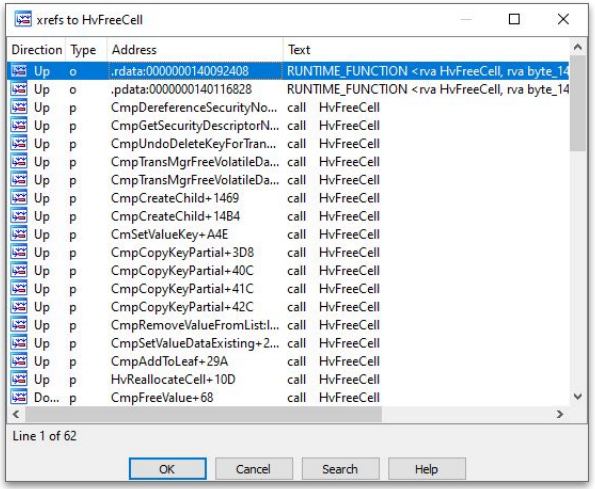
```
NTSTATUS HvReallocateCell(
    HHIVE *Hive,
    HCELL_INDEX CellIndex,
    DWORD Size,
    BOOLEAN bFreeOldCell,
    HCELL_INDEX *OutCellIndex,
    CELL_DATA **OutCellPointer,
    HV_GET_CELL_CONTEXT *CellContext
);
```

```
VOID HvFreeCell(
    HHIVE *Hive,
    HCELL_INDEX CellIndex
);
```

Allocator call sites







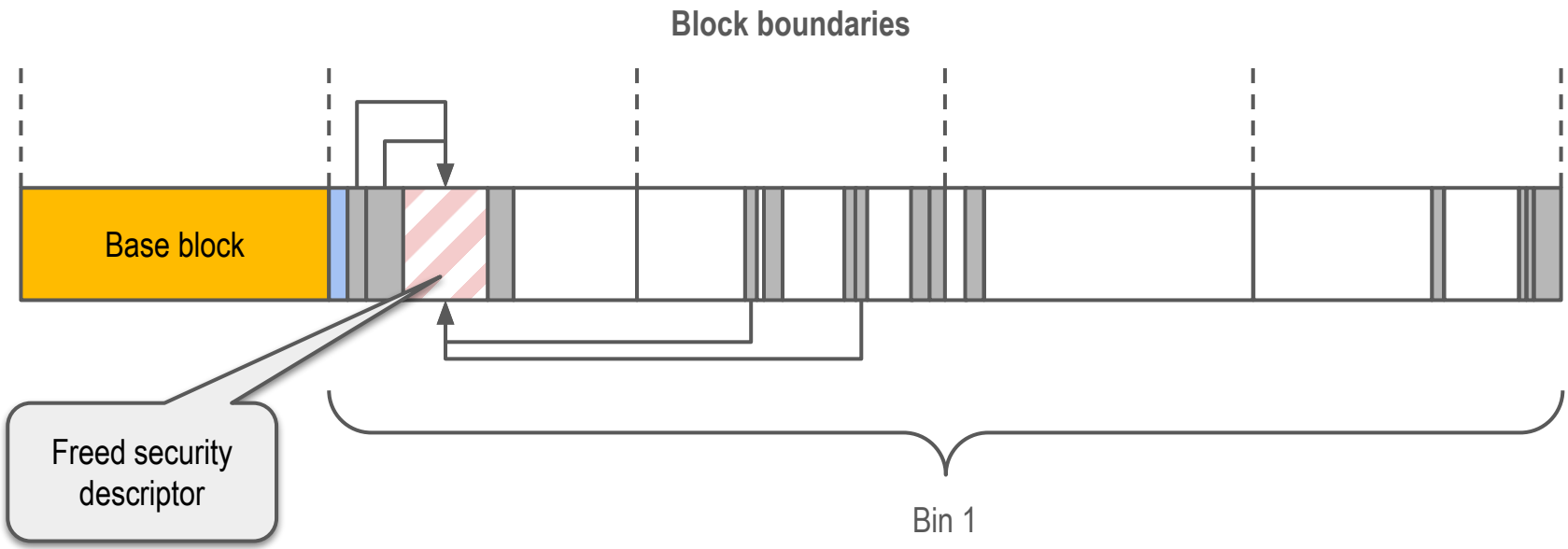
Comparing to heap/pool allocators

- Similarities:
  - Divided into contiguous "free" and "allocated" arbitrarily-sized chunks of data
  - Some cells have a predefined structure (e.g. key nodes), and others are entirely user-controlled (e.g. value data)
- Differences:
  - Exact same data layout maintained in memory and on disk
  - No randomization
  - No mitigations or any protection against temporal/spatial violations

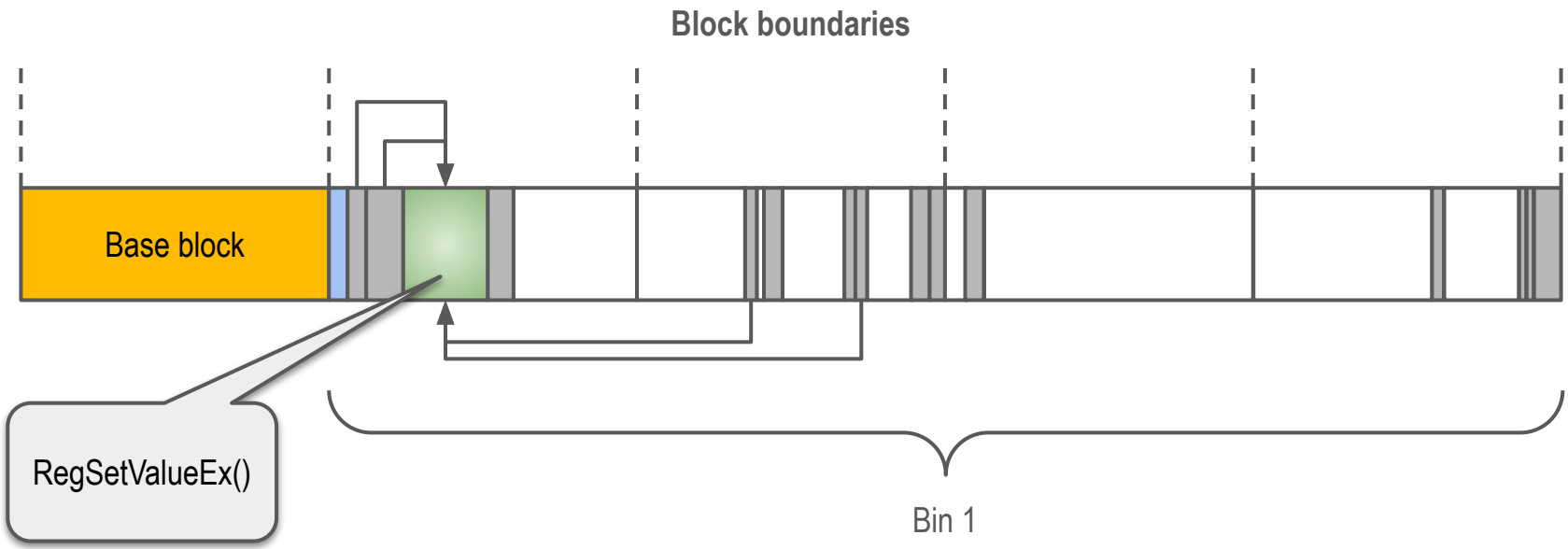
The 90's called... 📞

- No security cookies, double-free detection, etc.
- Fully deterministic, allocations always land in the same place
- Private app hives are not influenced by the environment noise
- Hive state grooming possible with a single function: **`RegSetValueEx`**

The space left by a freed cell...



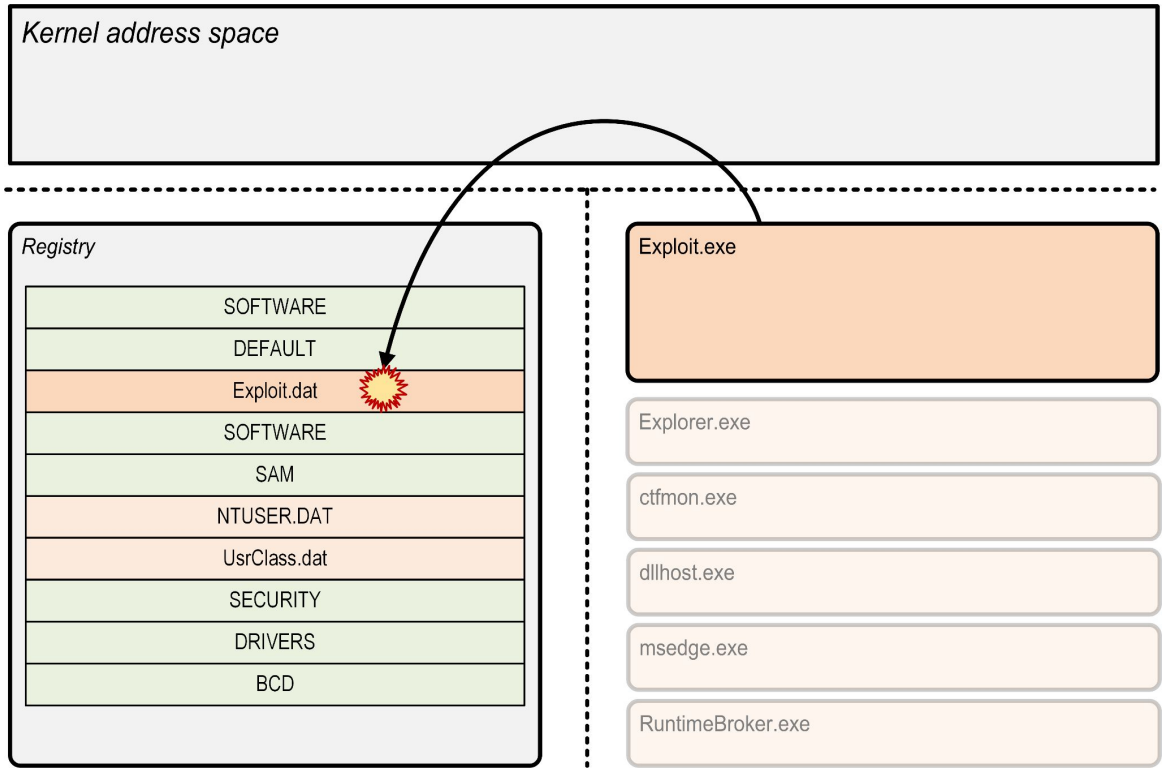
... can be easily reclaimed



Exploitation Middlegame:

Making Hive Corruption Useful

Current capability: the starting point



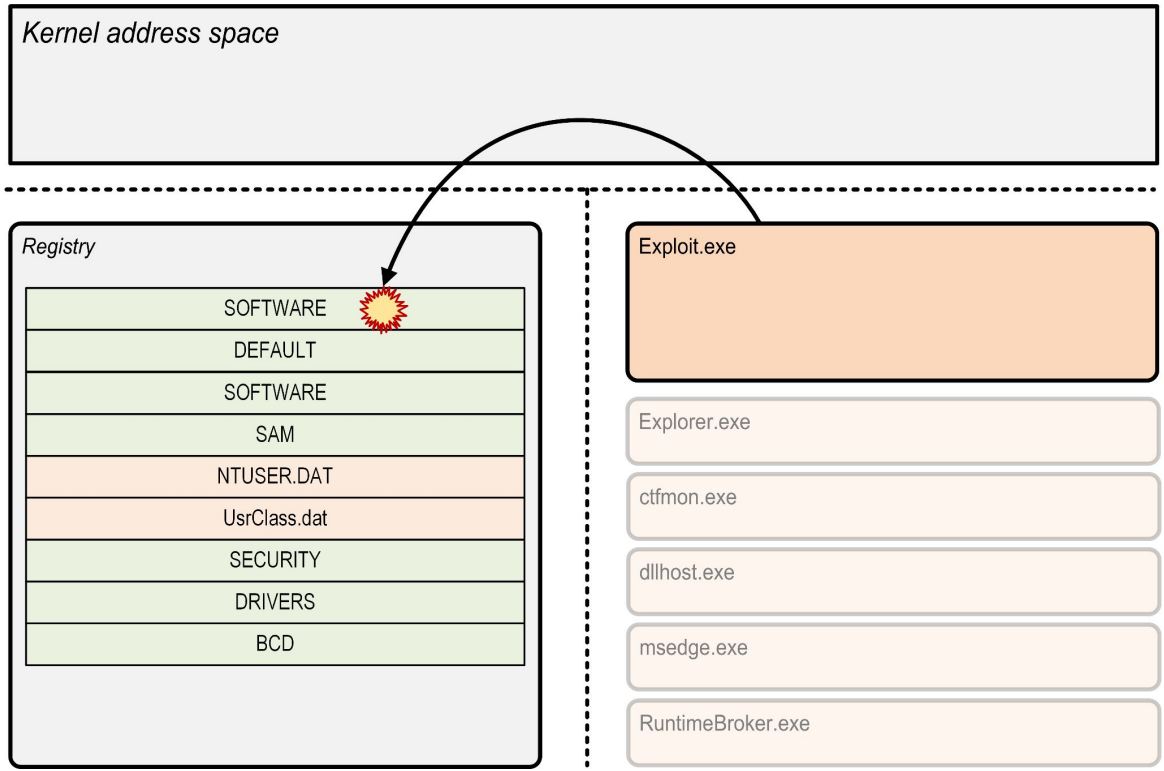
Corrupting security-relevant data

- Even full binary control doesn't offer much if it's our own hive
  - No secrets to steal via an infoleak
  - No "trusted" high-level data to alter
  - No virtual address space pointers to corrupt: the hive data is self-contained
- Finding a way to corrupt some other privileged/trusted memory is essential
- There are a few potential options...

Idea #1: directly corrupt a system hive

- How about just executing the exploit in the context of a system hive?
- Both SOFTWARE and SYSTEM have some user-writable keys
- We could carry out a truly data-only attack to get LPE
  - Lots of options: modify service configuration, auto-run keys, security descriptors, ...
- Downside: requires a *really* good bug with ~no prerequisites
  - Only maybe 2-3 such bugs that I've seen

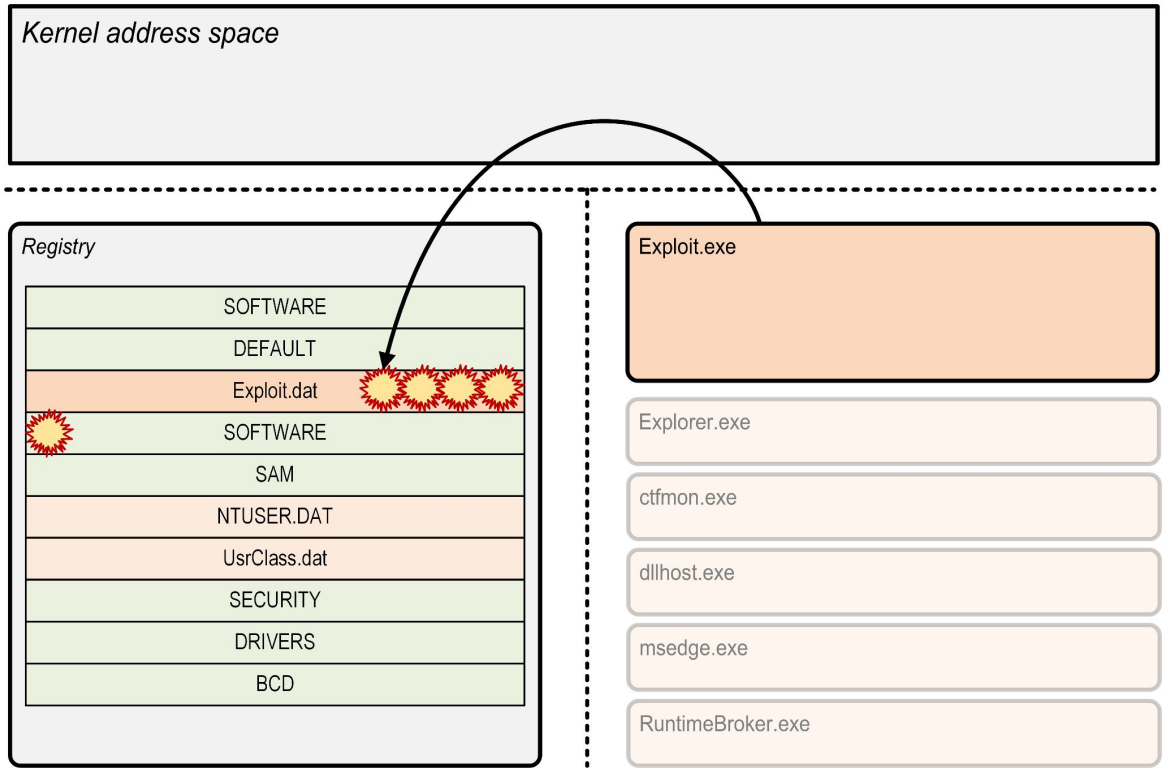
Idea #1: directly corrupt a system hive



Idea #2: corrupt adjacent hives

- Hives are mapped in the Registry process next to each other, with no gaps in between
  - With some memory feng shui, it is possible to have a system hive mapped after our own
  - We can potentially cross the boundary and corrupt the adjacent hive
- From there, exploitation techniques from idea #1 would apply
- Downside: reliability affected by the randomness of the memory layout

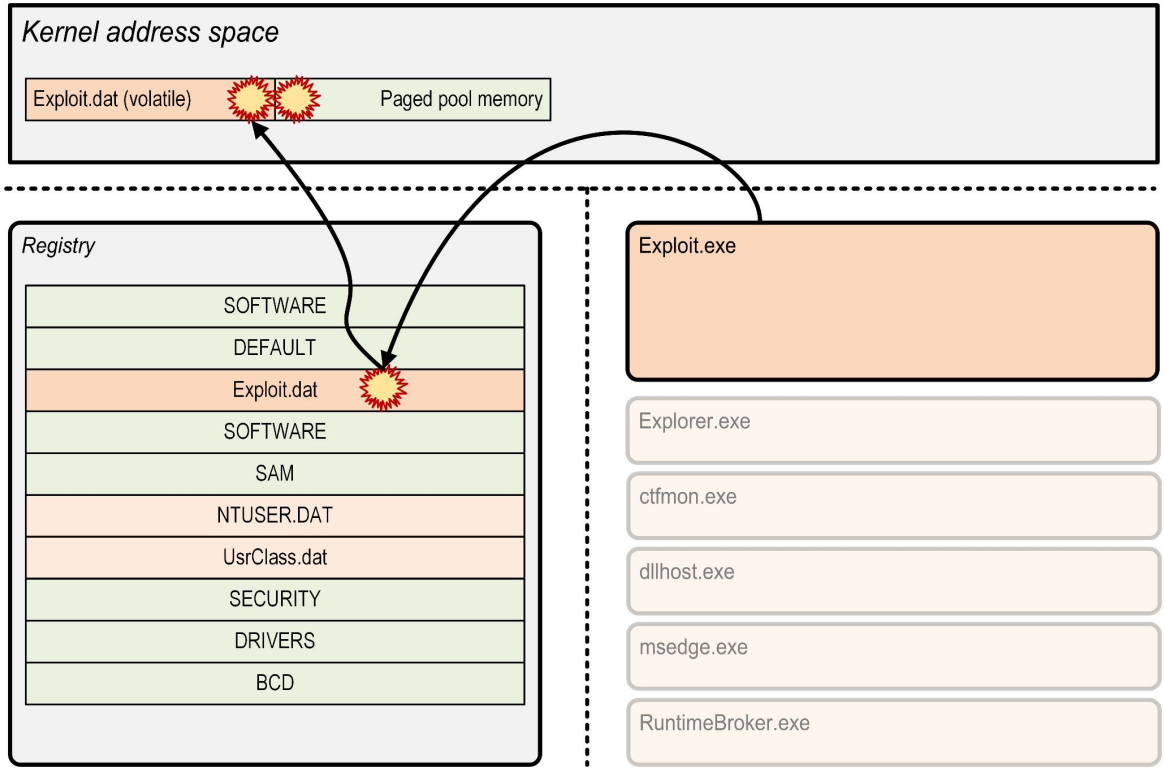
Idea #2: corrupt adjacent hives



Idea #3: corrupt adjacent kernel pool memory

- There are a few special cases where hive mappings reside in the kernel pool
  - The SYSTEM hive
  - Hives that have been freshly created by a load operation (the hive file didn't previously exist)
  - The *volatile* storage of every hive (i.e. the memory-only part)
- Stable space → volatile space → kernel pool corruption possible
- A very good and universal technique, the only downside: requires pool grooming and pool-specific exploitation primitives

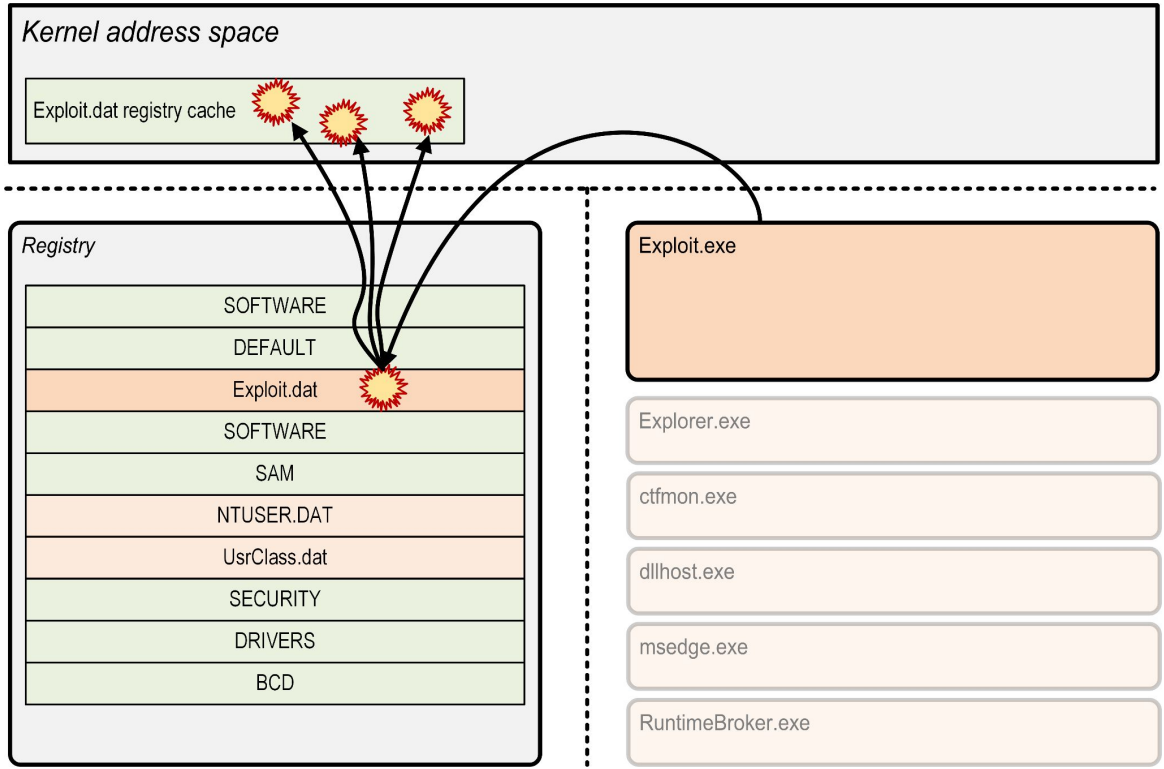
Idea #3: corrupt adjacent kernel pool memory



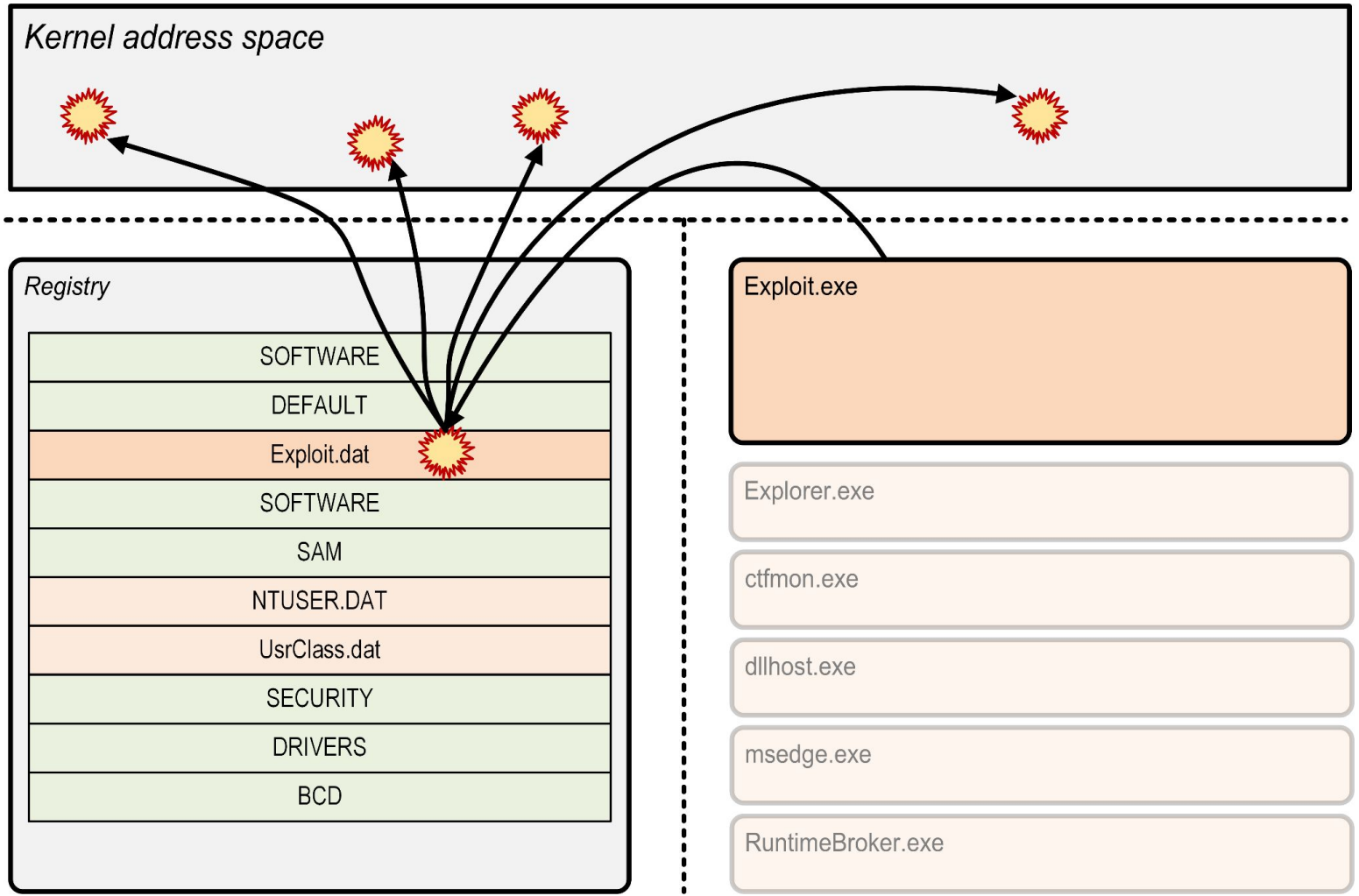
Idea #4: corrupt the registry cache structures

- The kernel caches portions of the hive in pool-based objects (KCBs etc.)
- By introducing inconsistencies, pool-based corruption can be induced
  - Modifying key names
  - Modifying key flags
  - Tampering with the reference counters of security descriptors
- Downside: no obvious path to LPE, would require digging more into the resulting primitives and devising a pool exploitation plan

Idea #4: corrupt the registry cache structures



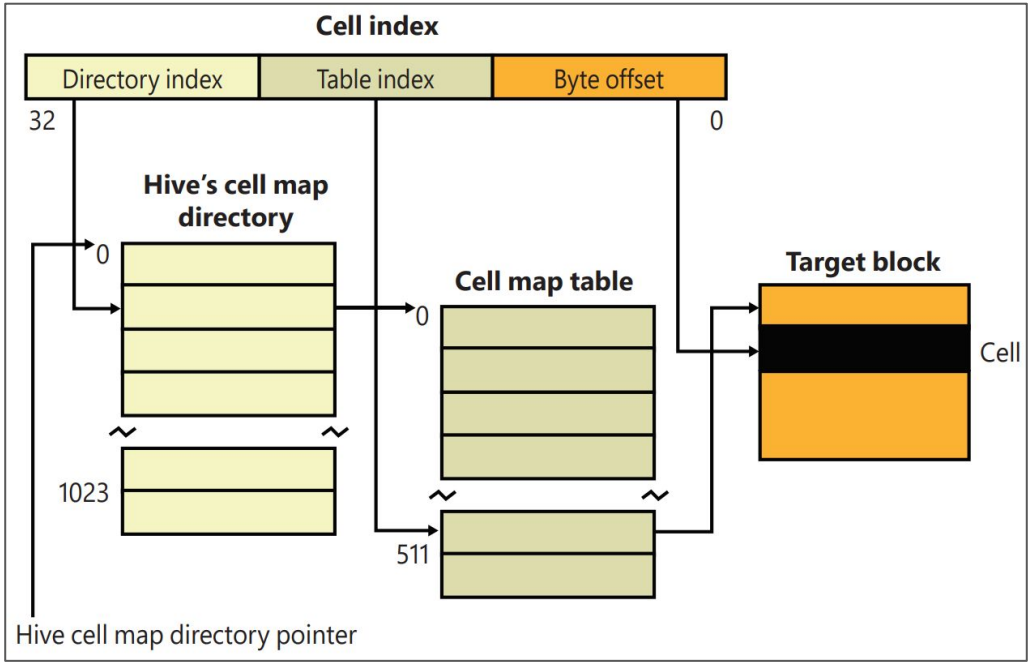
What we're aiming for


Kernel address space
Registry
SOFTWARE
DEFAULT
Exploit.dat
SOFTWARE
SAM
NTUSER.DAT
UsrClass.dat
SECURITY
DRIVERS
BCD
Exploit.exe
Explorer.exe
ctfmon.exe
dllhost.exe
msedge.exe
RuntimeBroker.exe

The trick: cell indexes

- **`Uint4B`** members used to cross-reference cells within the same hive
- In the file: simple offsets
- At runtime: more complicated, since hives are not mapped in memory as whole continuous blocks
  - They can grow and shrink dynamically
  - They are mapped with bin-granularity
- The index translation uses a page table-like structure called a *cell map*

Cell map layout



source: Windows Internals, 7th Edition, Part 2 (A. Allievi, A. Ionescu, M. Russinovich, D. Solomon)

Cell translation code (simplified)

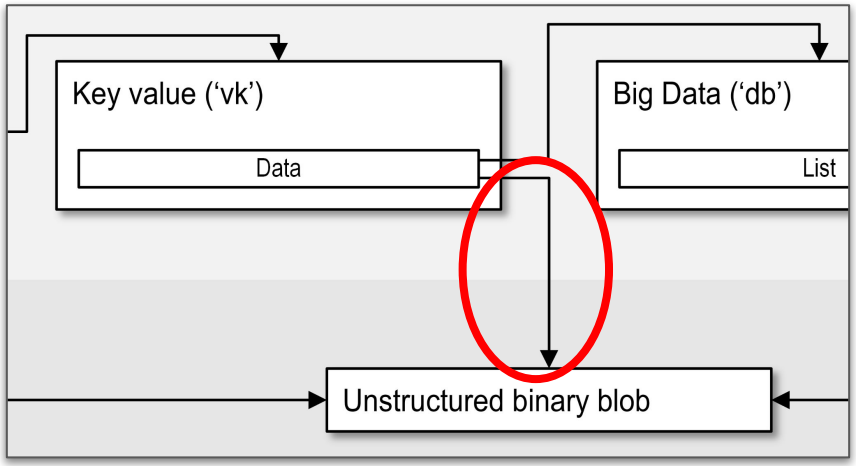
```
PVOID HvpGetCellPaged(HHIVE *Hive, HCELL_INDEX CellIndex) {
  HMAP_ENTRY MapEntry;

  MapEntry = Hive->Storage[CellIndex >> 31].Map
                 ->Directory[(CellIndex >> 21) & 0x3FF]
                 ->Table[(CellIndex >> 12) & 0x1FF];

  return (MapEntry->PermanentBinAddress & (~0xF)) +
         MapEntry->BlockOffset +
         (CellIndex & 0xFFF);
}
```

Typical usage

```
VOID CmpSetValueData(HHIVE *Hive, CM_KEY_VALUE *ValueNode, PVOID Data, ULONG DataLength) {
  //
  // Acquire the `Data` cell via HvpGetCellPaged
  //
  PVOID ValueDataPointer = Hive->GetCellRoutine(Hive, ValueNode->Data);

  //
  // Copy data to the cell through the raw pointer
  //
  RtlCopyMemory(ValueDataPointer, Data, DataLength);

  //
  // Release the cell via HvpReleaseCellPaged
  //
  Hive->ReleaseCellRoutine(Hive, ValueNode->Data);
}
```

Using values as proxy

- If we control `CM_KEY_VALUE.Data`, then we can use **`RegQueryValueEx`** and **`RegSetValueEx`** to directly read/write to that cell index



Abusing cell indexes

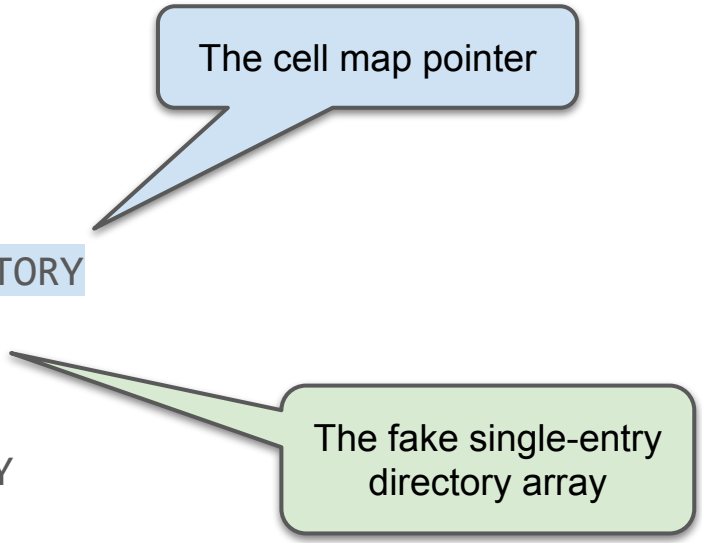
- A natural question: what if an index is greater than the hive size?
- The kernel doesn't care: cell indexes are trusted at runtime, so `HvpGetCellPaged` will just do the usual cell walk
- However, every level of the table is allocated in full size, and zeroed out
- Does that mean that any OOB index access will just lead to a NULL pointer dereference? Yes, but...

The "small dir" optimization

- With this logic, the kernel would have to allocate the full top-level directory (0x2000 bytes) even for tiny hives
- Optimization: if the hive is ≤ 2 MiB (fits into a single directory entry), a fake one-entry "array" placed **inline** in the hive descriptor is used
- Suddenly there *is* non-zero memory at OOB cell map indexes

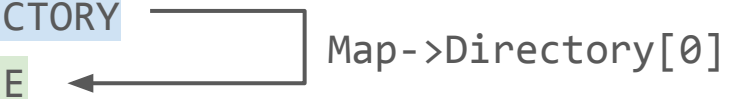
The cell map

```
kd> dt _DUAL
nt!_DUAL
   +0x000 Length           : Uint4B
   +0x008 Map              : Ptr64 _HMAP_DIRECTORY
   +0x010 SmallDir         : Ptr64 _HMAP_TABLE
   +0x018 Guard            : Uint4B
   +0x020 FreeDisplay      : [24] _FREE_DISPLAY
   +0x260 FreeBins         : _LIST_ENTRY
   +0x270 FreeSummary      : Uint4B
```



The cell map

```
kd> dt _DUAL
nt!_DUAL
   +0x000 Length           : Uint4B
   +0x008 Map              : Ptr64 _HMAP_DIRECTORY
   +0x010 SmallDir         : Ptr64 _HMAP_TABLE
   +0x018 Guard            : Uint4B
   +0x020 FreeDisplay      : [24] _FREE_DISPLAY
   +0x260 FreeBins         : _LIST_ENTRY
   +0x270 FreeSummary      : Uint4B
```

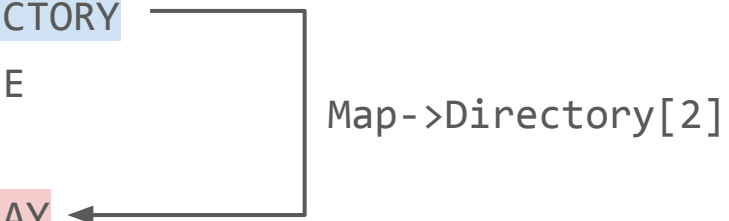


The cell map

```
kd> dt _DUAL
nt!_DUAL
   +0x000 Length           : Uint4B
   +0x008 Map              : Ptr64 _HMAP_DIRECTORY
   +0x010 SmallDir         : Ptr64 _HMAP_TABLE
   +0x018 Guard            : Uint4B
   +0x020 FreeDisplay      : [24] _FREE_DISPLAY
   +0x260 FreeBins         : _LIST_ENTRY
   +0x270 FreeSummary      : Uint4B
```

The cell map

```
kd> dt _DUAL
nt!_DUAL
   +0x000 Length           : Uint4B
   +0x008 Map              : Ptr64 _HMAP_DIRECTORY
   +0x010 SmallDir         : Ptr64 _HMAP_TABLE
   +0x018 Guard            : Uint4B
   +0x020 FreeDisplay      : [24] _FREE_DISPLAY
   +0x260 FreeBins         : _LIST_ENTRY
   +0x270 FreeSummary      : Uint4B
```

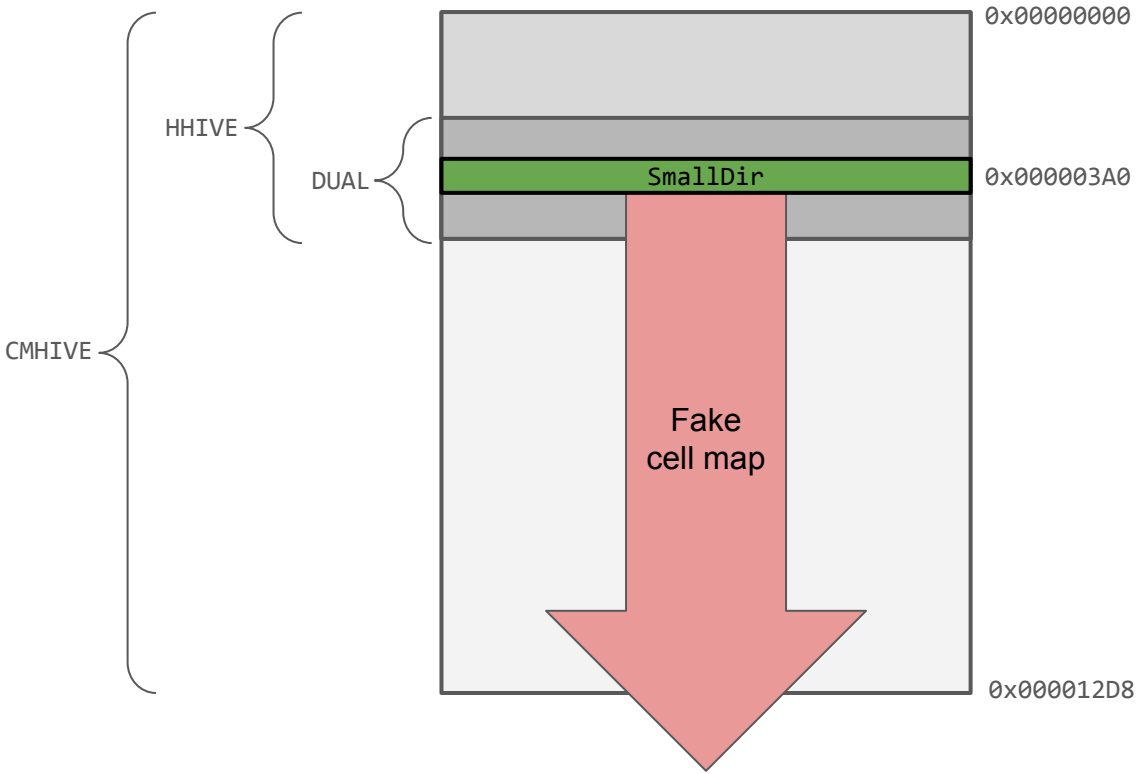


... and so on

The OOB cell walk primitive

- With this, we can perform a controlled two-level pointer dereference:
  a. **Access one of the 1024 pointers** directly following `SmallDir`
  b. From there, **access one of the 512 pointers** stored relative to that address (with caveats)
  c. **Add any 12-bit number** and have the sum returned by `HvpGetCellPaged`

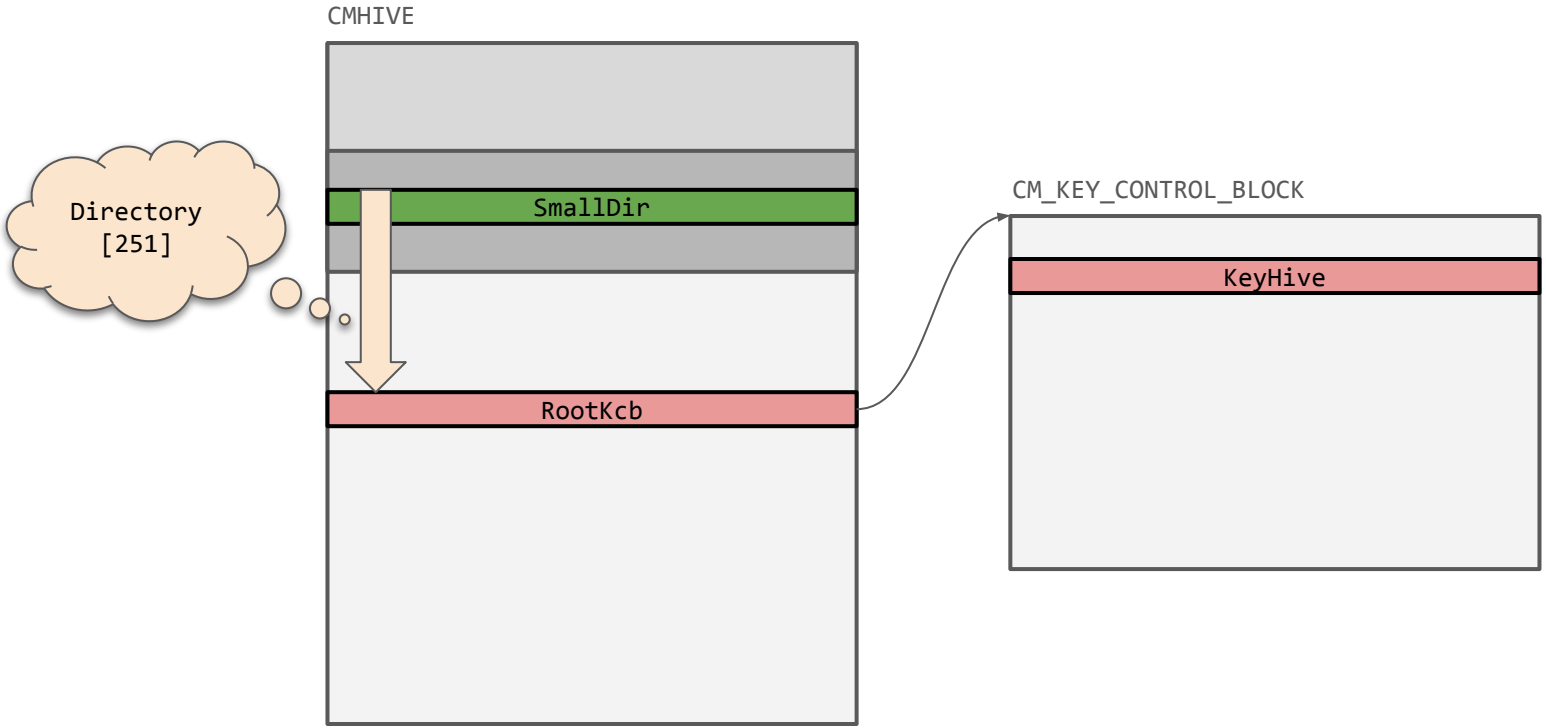
The CMHIVE structure layout



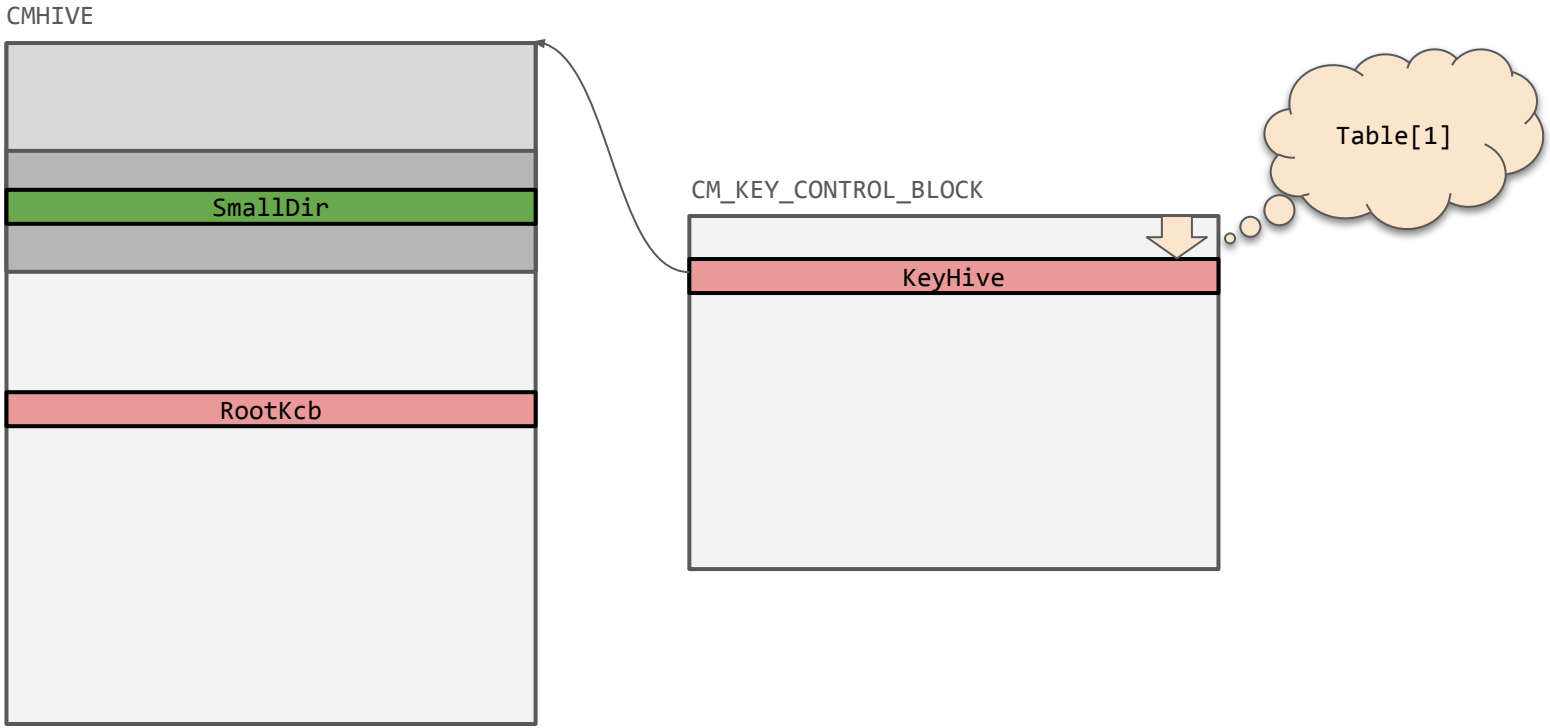
Where to point to?

- One idea is **`CMHIVE`** → **`KCB`** → **`CMHIVE`**, to have the cell point straight back at the hive descriptor
  - `CMHIVE.RootKcb` points at a KCB for app hives
  - `CM_KEY_CONTROL_BLOCK.KeyHive` points back at the hive object
- We can then add any offset up to 0xFFF to point at a specific field

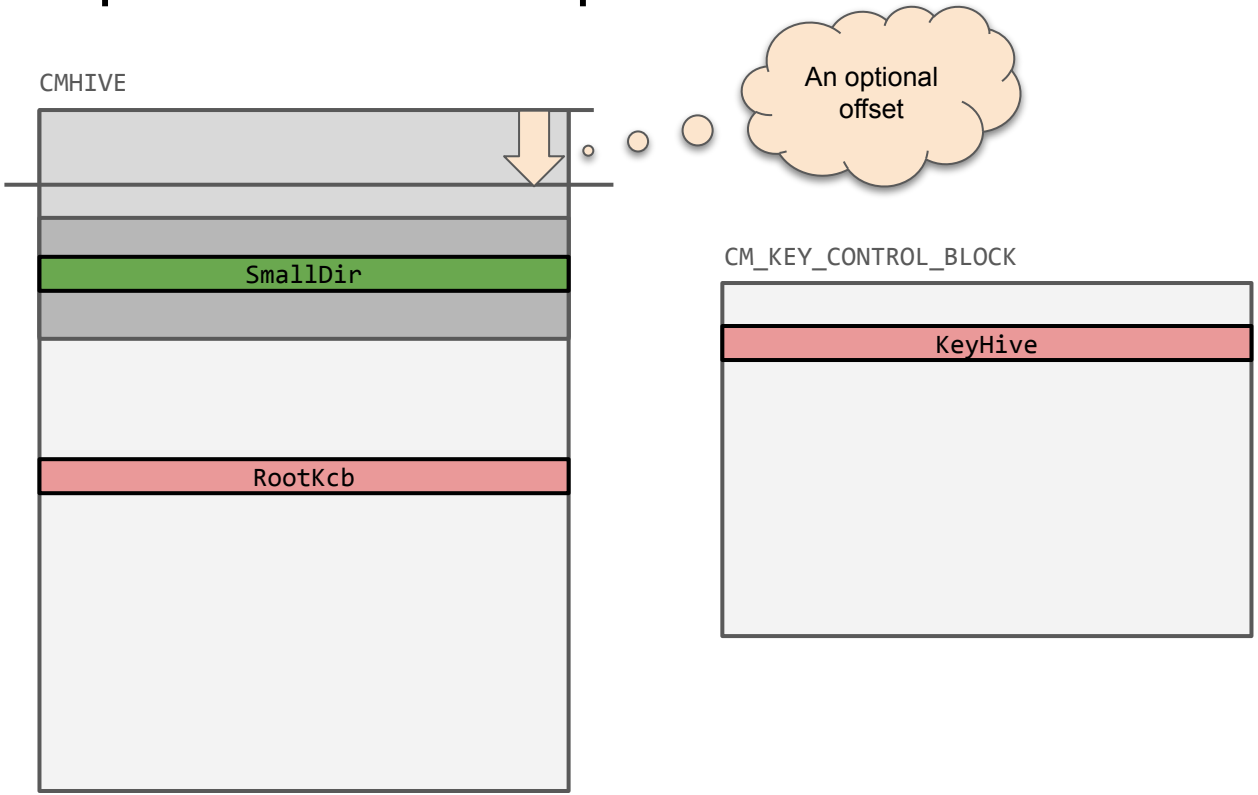
Hive descriptor index example



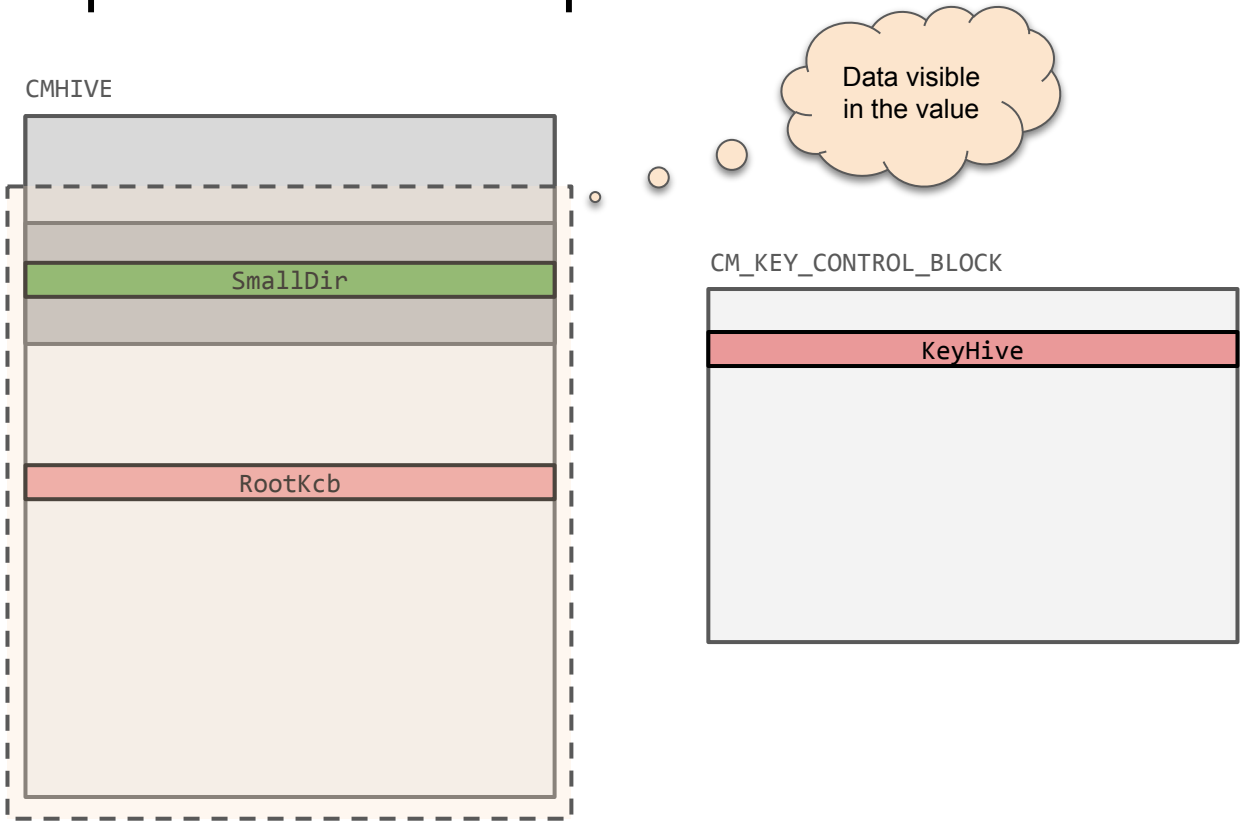
Hive descriptor index example



Hive descriptor index example



Hive descriptor index example



The final index

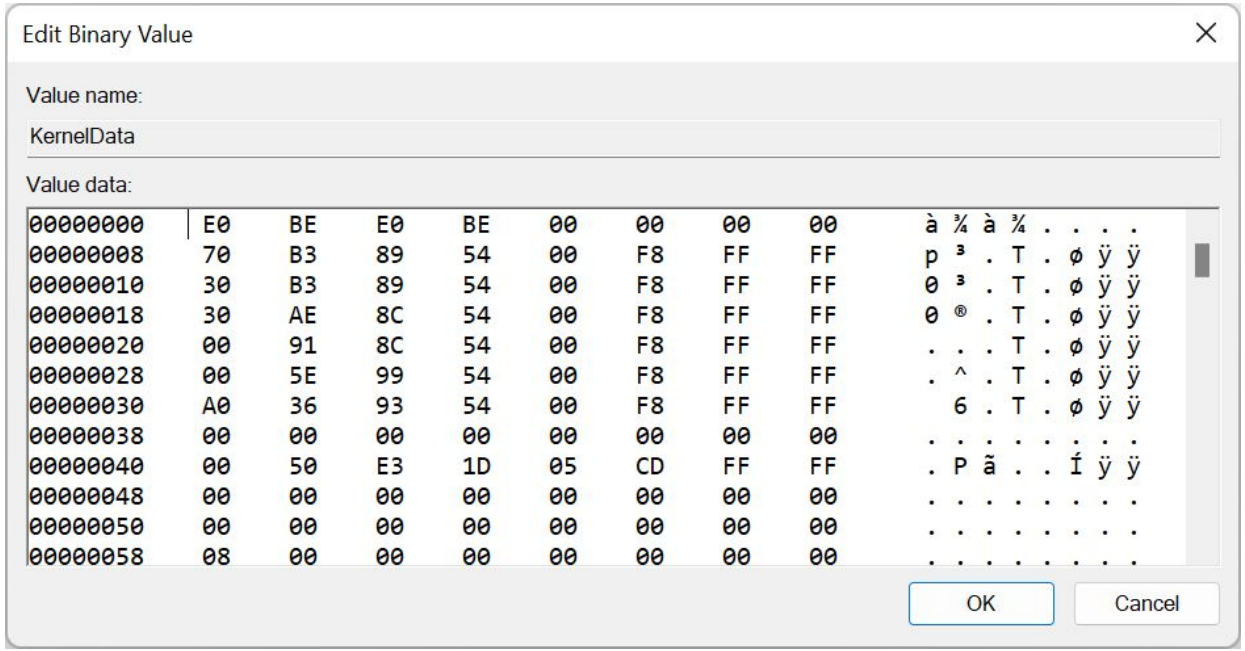
- Putting this together, we get the special index **0x9F601000**
- It can be verified in WinDbg:

```
0: kd> !reg cellindex ffffcd051ed81000 9f601000

Map = ffffcd051ed813a0 Type = 1 Table = fb Block = 1 Offset = 0
MapTable     = ffffcd051dd2a700
MapEntry     = ffffcd051dd2a718
BinAddress = ffffcd051ed81000, BlockOffset = 0000000000000000
BlockAddress = ffffcd051ed81000

pcell:  ffffcd051ed81004
```

Read/write access to the hive descriptor

- From here, there are already many ways to escalate privileges
- For example, the first few members are function pointers:

```
0: kd> dt _HHIVE ffffcd051ed81000
nt!_HHIVE
   +0x000 Signature        : 0xbee0bee0
   +0x008 GetCellRoutine   : 0xfffff800`5489b370     _CELL_DATA*  nt!HvpGetCellPaged+0
   +0x010 ReleaseCellRoutine : 0xfffff800`5489b330     void  nt!HvpReleaseCellPaged+0
   +0x018 Allocate         : 0xfffff800`548cae30     void*  nt!CmpAllocate+0
   +0x020 Free             : 0xfffff800`548c9100     void  nt!CmpFree+0
   +0x028 FileWrite        : 0xfffff800`54995e00     long  nt!CmpFileWrite+0
   +0x030 FileRead         : 0xfffff800`549336a0     long  nt!CmpFileRead+0
```


Edit Binary Value
Value name:
KernelData
Value data:
00000000 E0 BE E0 BE 00 00 00 00 à ¾ à ¾ . . . .
00000008 70 B3 89 54 00 F8 FF FF p ³ . T . ø ÿ ÿ
00000010 30 B3 89 54 00 F8 FF FF 0 ³ . T . ø ÿ ÿ
00000018 30 AE 8C 54 00 F8 FF FF 0 ® . T . ø ÿ ÿ
00000020 00 91 8C 54 00 F8 FF FF . . . T . ø ÿ ÿ
00000028 00 5E 99 54 00 F8 FF FF . ^ . T . ø ÿ ÿ
00000030 A0 36 93 54 00 F8 FF FF 6 . T . ø ÿ ÿ
00000038 00 00 00 00 00 00 00 00 . . . .
00000040 00 50 E3 1D 05 CD FF FF . P ã . . Í ÿ ÿ
00000048 00 00 00 00 00 00 00 00 . . . .
00000050 00 00 00 00 00 00 00 00 . . . .
00000058 08 00 00 00 00 00 00 00 . . . .
OK
Cancel

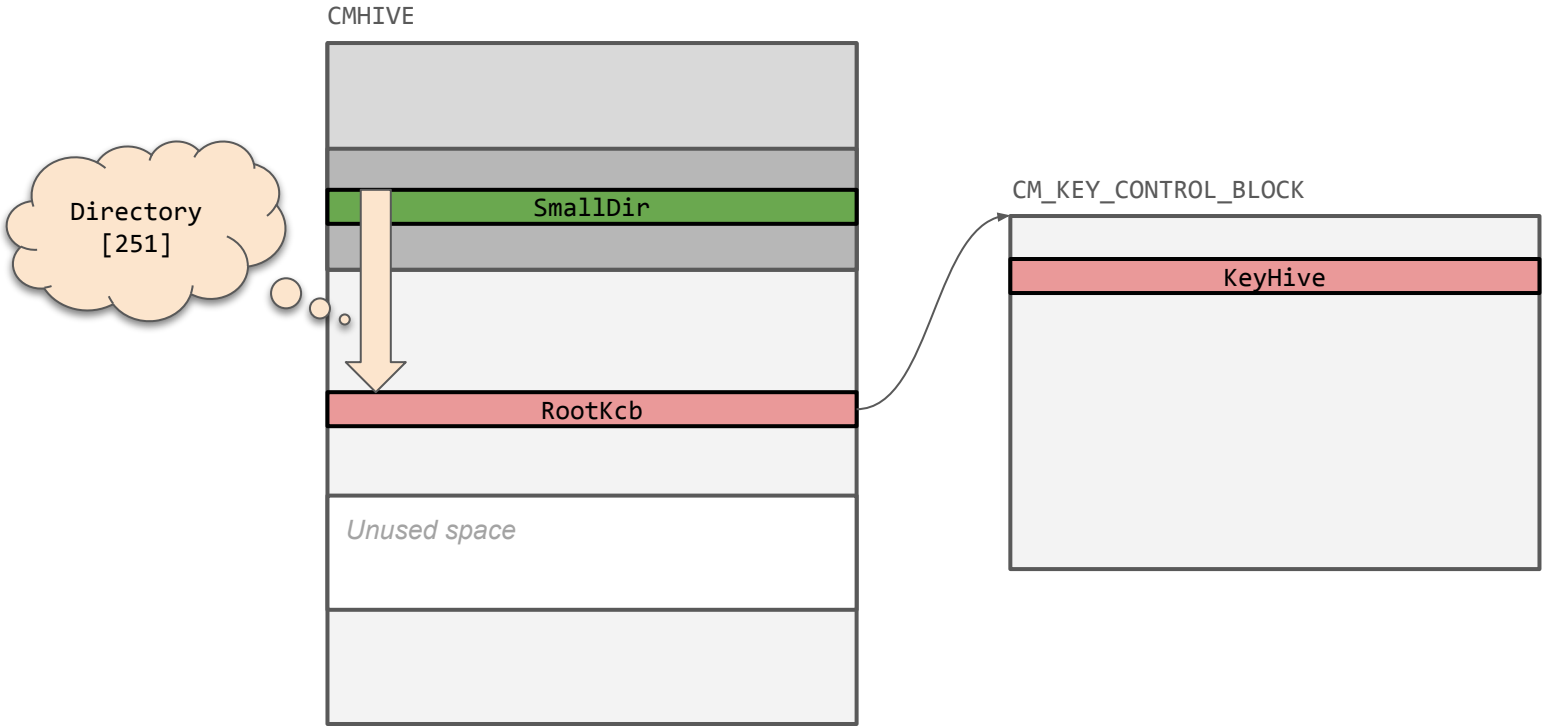
Kernel function pointers

- Corrupting them is tempting, but things are not as easy as they used to be
  - Kernel Control Flow Guard (kCFG)
  - On Windows 11, the calls are entirely devirtualized and the pointers are never used
- They are still very useful to bypass KASLR

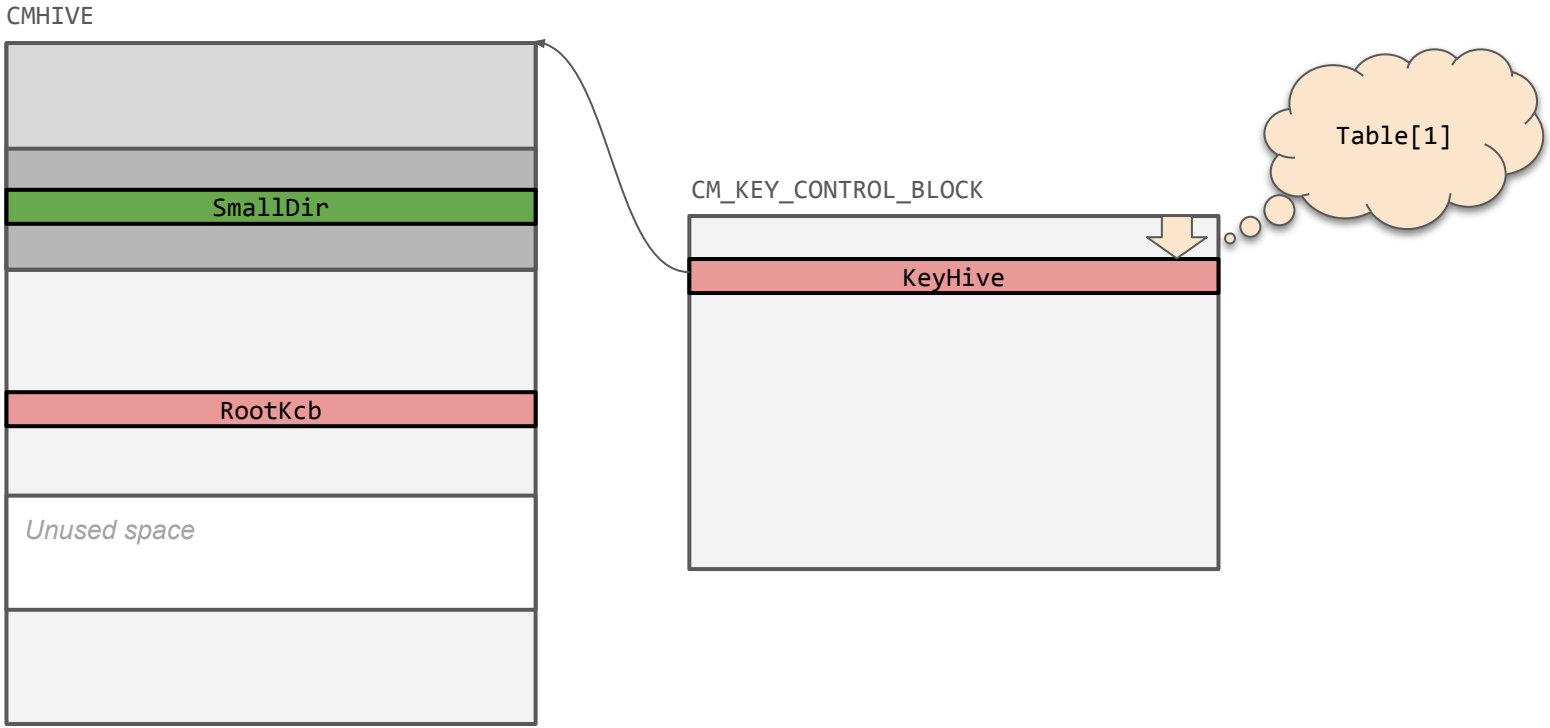
What else

- Let's go back to getting arbitrary read/write
- We have full access to the `CMHIVE` structure, but how to use it?
- Here's an idea:
  - Craft a "KernelAddr" value with its data index pointing to some empty/unused part of `CMHIVE`
  - Craft a "KernelData" value with its data index pointing **through** the same part of the structure, and treating it as the last level of the cell map (`HMAP_ENTRY`)
  - Write some address into KernelAddr
  - Memory from that virtual address should be automatically reflected in KernelData
- An example pair of indexes that achieve this: **0x9F601F14**, **0x8F252000**

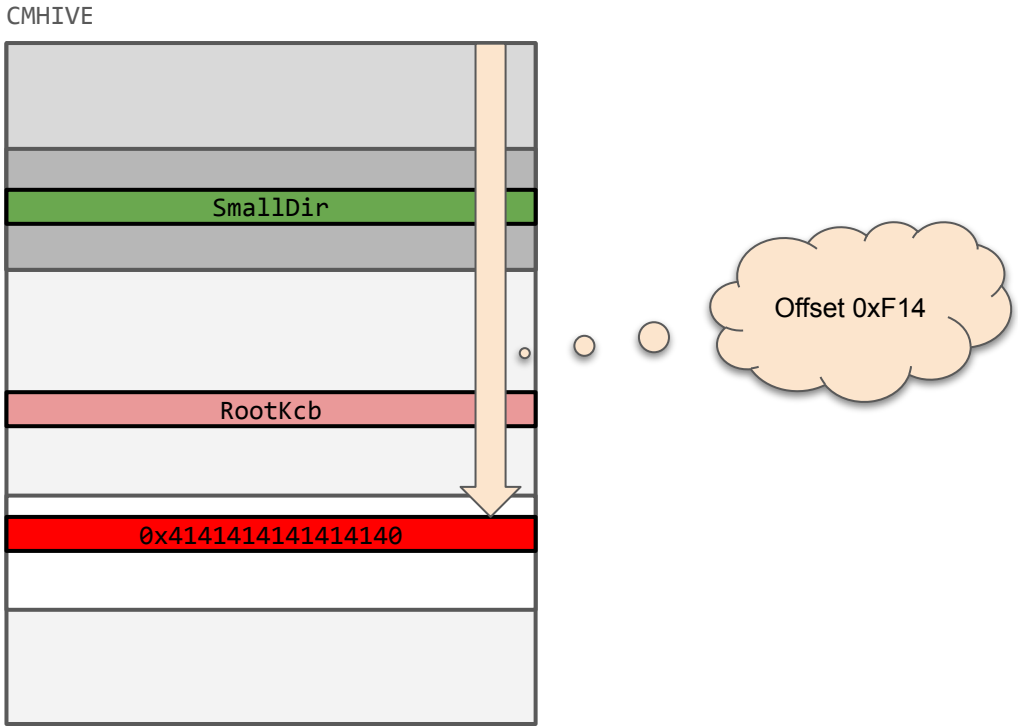
KernelAddr (0x9F601F14) cell walk



KernelAddr (0x9F601F14) cell walk

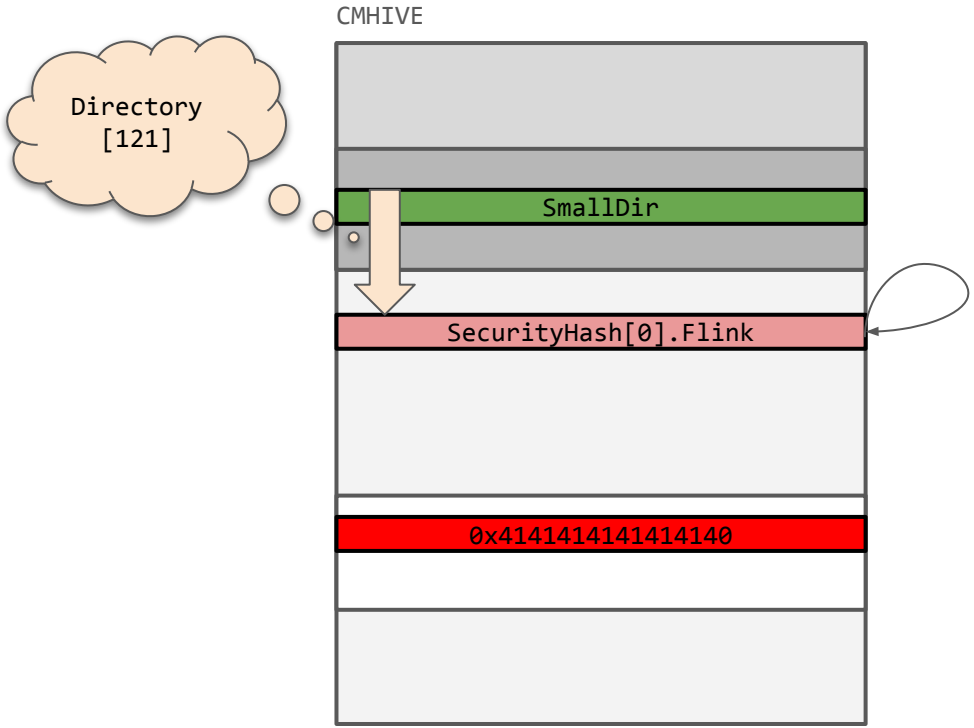


KernelAddr (0x9F601F14) cell walk

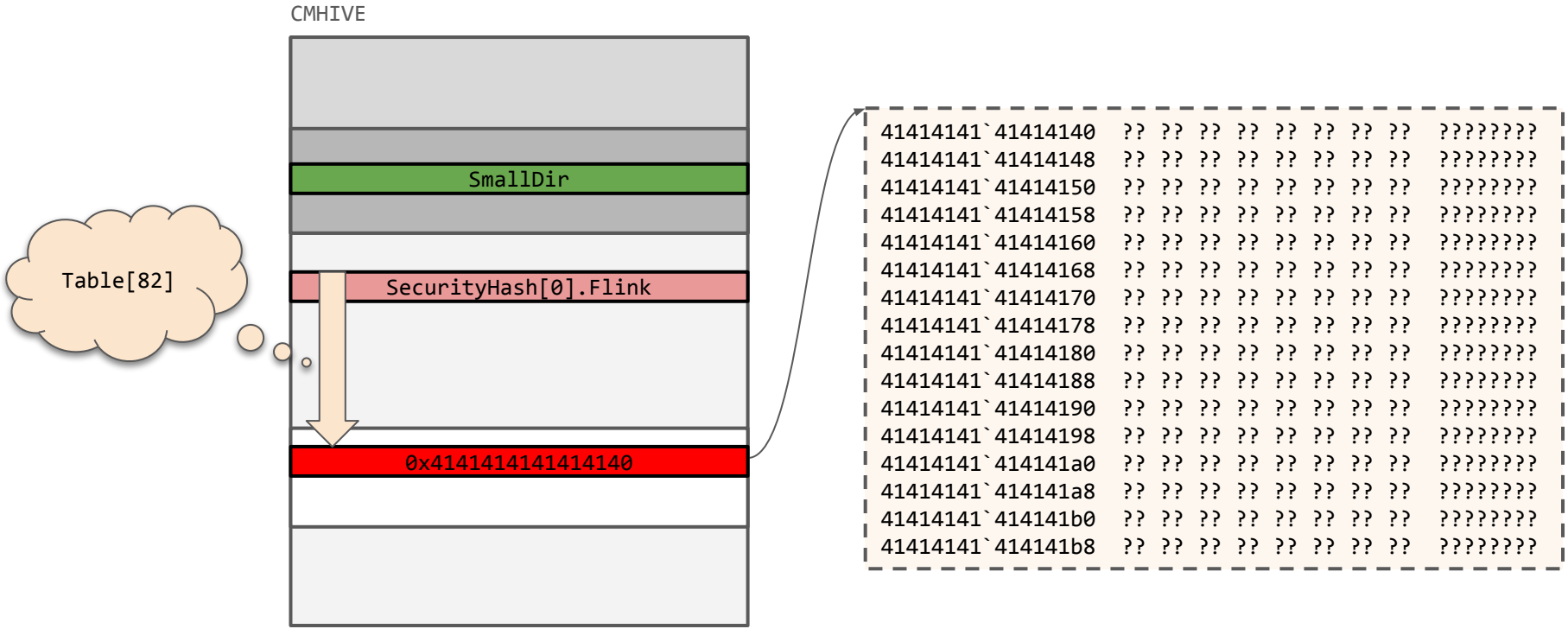


and then...

KernelData (0x8F252000) cell walk



KernelData (0x8F252000) cell walk



Mission accomplished

- We now have a direct window into the entire kernel address space
- We can finish the data-only attack by operating on the ✨magic✨ values
- All that's left to implement is the actual LPE logic

Exploitation Endgame:

Compromising the system

Exploit development

- The plan seems bulletproof, there is certainly nothing that can go wrong here
- Let's just write the exploit as a formality...

:(

Your device ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

100% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: REGISTRY_ERROR

???

```
kd> k
 # Child-SP          RetAddr               Call Site
00 ffffcf0c`3291ddd8 fffff800`08163642     nt!DbgBreakPointWithStatus
01 ffffcf0c`3291dde0 fffff800`08162e81     nt!KiBugCheckDebugBreak+0x12
02 ffffcf0c`3291de40 fffff800`08017957     nt!KeBugCheck2+0xa71
03 ffffcf0c`3291e5b0 fffff800`084874d5     nt!KeBugCheckEx+0x107
04 ffffcf0c`3291e5f0 fffff800`082a55e1     nt!HvpReleaseCellPaged+0x1ec1a5
05 ffffcf0c`3291e630 fffff800`082a45df     nt!CmpCompareNewValueDataAgainstKCBCache+0x191
06 ffffcf0c`3291e6c0 fffff800`082a5283     nt!CmSetValueKey+0x21f
07 ffffcf0c`3291e870 fffff800`08029e75     nt!NtSetValueKey+0x643
08 ffffcf0c`3291ea70 00007ffc`51b44aa4     nt!KiSystemServiceCopyEnd+0x25
09 000000bc`632ffae8 00000000`00000000     0x00007ffc`51b44aa4
```

??????????

```
VOID HvpReleaseCellPaged(HHIVE *Hive, HCELL_INDEX CellIndex) {
  //
  // Check that the index doesn't exceed the hive size
  //
  if ((CellIndex & 0x7FFFFFFF) >= Hive->Storage[CellIndex >> 31].Length) {
    KeBugCheckEx(REGISTRY_ERROR, 1, Hive, CellIndex, 0x267);
  }
}
```

Let's recap

Typical cell index usage

```
VOID CmpSetValueData(HHIVE *Hive, CM_KEY_VALUE *ValueNode, PVOID Data, ULONG DataLength) {
  //
  // Acquire the `Data` cell via HvpGetCellPaged
  //
  PVOID ValueDataPointer = Hive->GetCellRoutine(Hive, ValueNode->Data);

  //
  // Copy data to the cell through the raw pointer
  //
  RtlCopyMemory(ValueDataPointer, Data, DataLength);

  //
  // Release the cell via HvpReleaseCellPaged
  //
  Hive->ReleaseCellRoutine(Hive, ValueNode->Data);
}
```

This doesn't crash and returns an arbitrary pointer

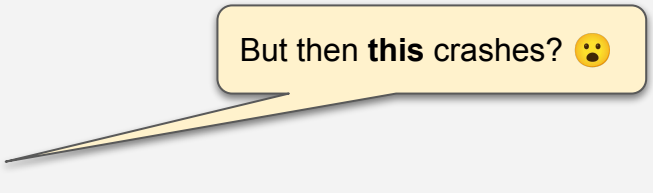
Typical cell index usage

```
VOID CmpSetValueData(HHIVE *Hive, CM_KEY_VALUE *ValueNode, PVOID Data, ULONG DataLength) {
  //
  // Acquire the `Data` cell via HvpGetCellPaged
  //
  PVOID ValueDataPointer = Hive->GetCellRoutine(Hive, ValueNode->Data);

  //
  // Copy data to the cell through the raw pointer
  //
  RtlCopyMemory(ValueDataPointer, Data, DataLength);

  //
  // Release the cell via HvpReleaseCellPaged
  //
  Hive->ReleaseCellRoutine(Hive, ValueNode->Data);
}
```

This doesn't crash and operates on the invalid pointer

Typical cell index usage

```
VOID CmpSetValueData(HHIVE *Hive, CM_KEY_VALUE *ValueNode, PVOID Data, ULONG DataLength) {
  //
  // Acquire the `Data` cell via HvpGetCellPaged
  //
  PVOID ValueDataPointer = Hive->GetCellRoutine(Hive, ValueNode->Data);

  //
  // Copy data to the cell through the raw pointer
  //
  RtlCopyMemory(ValueDataPointer, Data, DataLength);

  //
  // Release the cell via HvpReleaseCellPaged
  //
  Hive->ReleaseCellRoutine(Hive, ValueNode->Data);
}
```



The role of the "cell release"

- Historically, `HvpReleaseCellPaged` was used on and off in Windows history
  - **Windows XP – Windows 7**: used for bookkeeping and debug checks
  - **Windows 10 – Windows 11**: only performs the cell index bounds check
- It's not the best idea to sanitize data after use, but it still breaks the exploit
- If we can't access OOB indexes, the whole technique goes to waste

A last-ditch effort

```
VOID HvpReleaseCellPaged(HHIVE *Hive, HCELL_INDEX CellIndex) {
  //
  // Check that the index doesn't exceed the hive size
  //
  if ((CellIndex & 0x7FFFFFFF) >= Hive->Storage[CellIndex >> 31].Length) {
    KeBugCheckEx(REGISTRY_ERROR, 1, Hive, CellIndex, 0x267);
  }
}
```

- The length referenced in the check is stored in the `CMHIVE` object
- A moment earlier, we get **one** write to a controlled cell index
- Hmm... is this a CTF challenge? 🤔

A last-ditch effort

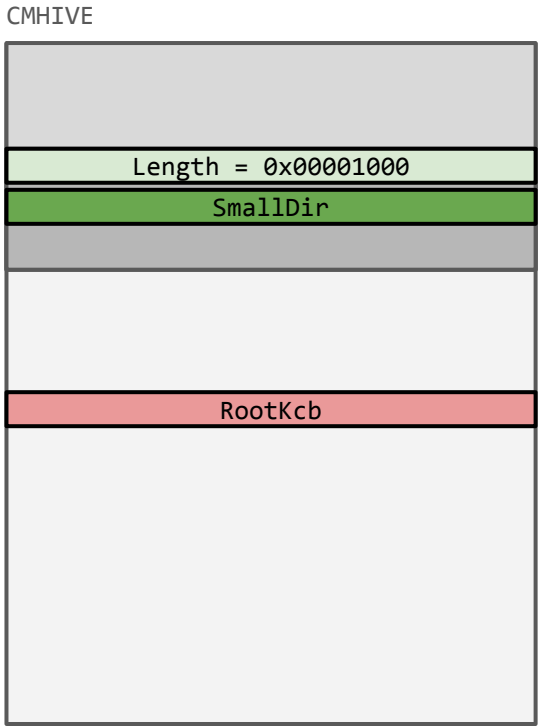
```
VOID CmpSetValueData(HHIVE *Hive, CM_KEY_VALUE *ValueNode, PVOID Data, ULONG DataLength) {
  //
  // Acquire the `Data` cell via HvpGetCellPaged
  //
  PVOID ValueDataPointer = Hive->GetCellRoutine(Hive, ValueNode->Data);

  //
  // Copy data to the cell through the raw pointer
  //
  RtlCopyMemory(ValueDataPointer, Data, DataLength);

  //
  // Release the cell via HvpReleaseCellPaged
  //
  Hive->ReleaseCellRoutine(Hive, ValueNode->Data);
}
```

1 Choose an OOB cell index that points at the `Length` member

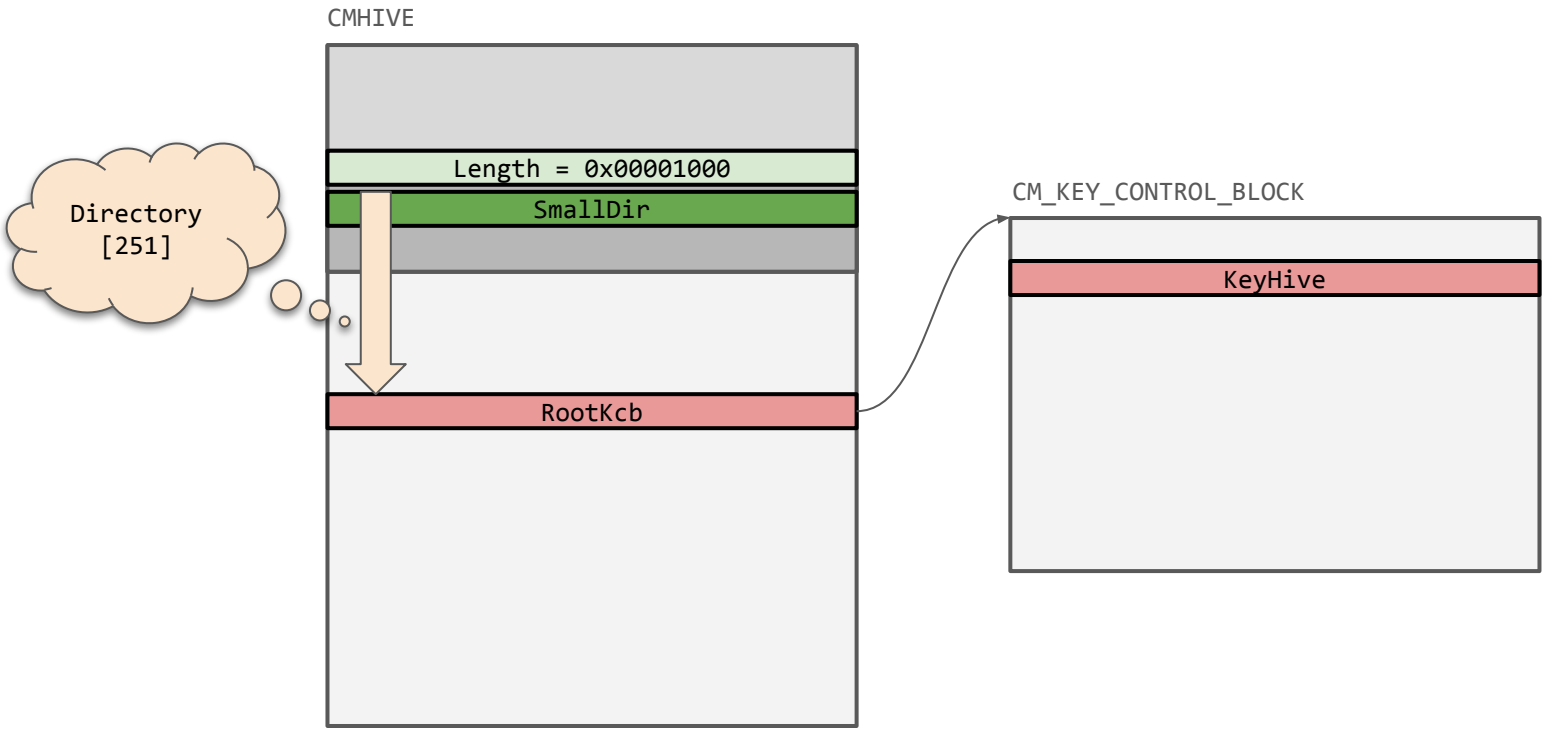
2 Overwrite it with a large integer, like 0xFFFFFFFF

3 This won't crash now!

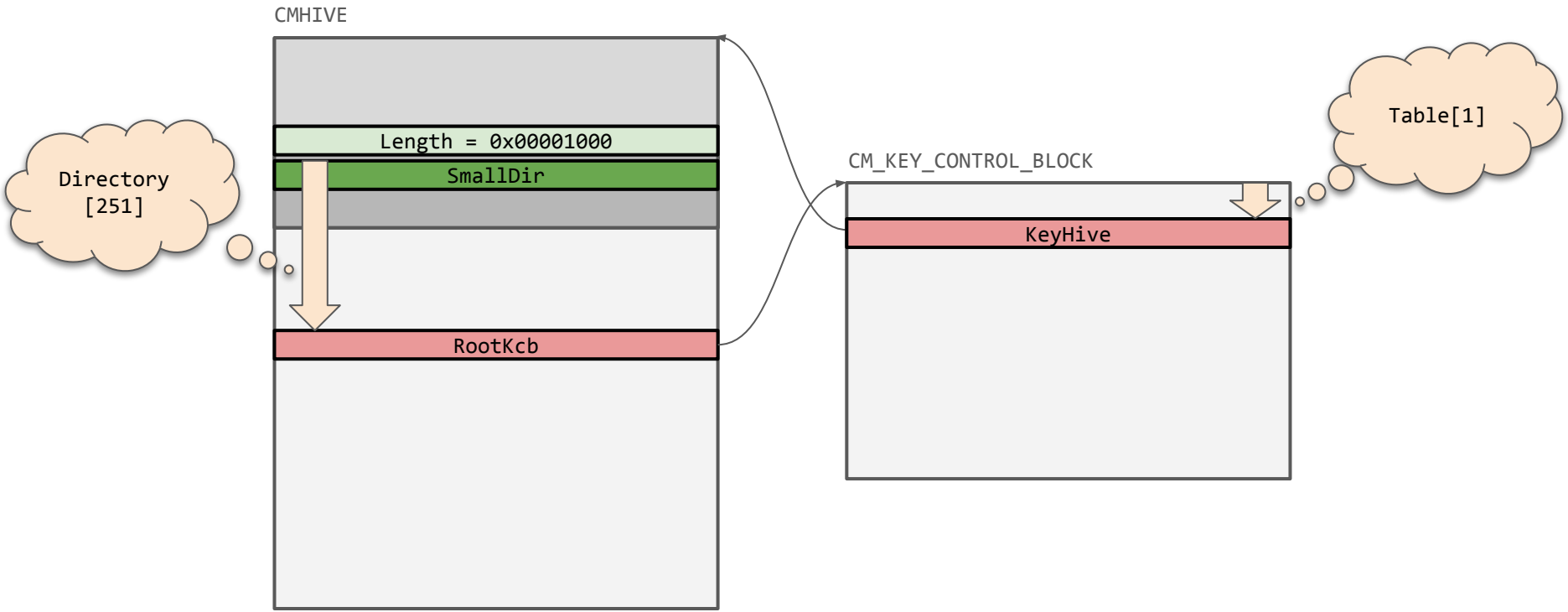
Length corruption cell walk (0x9F60138C)



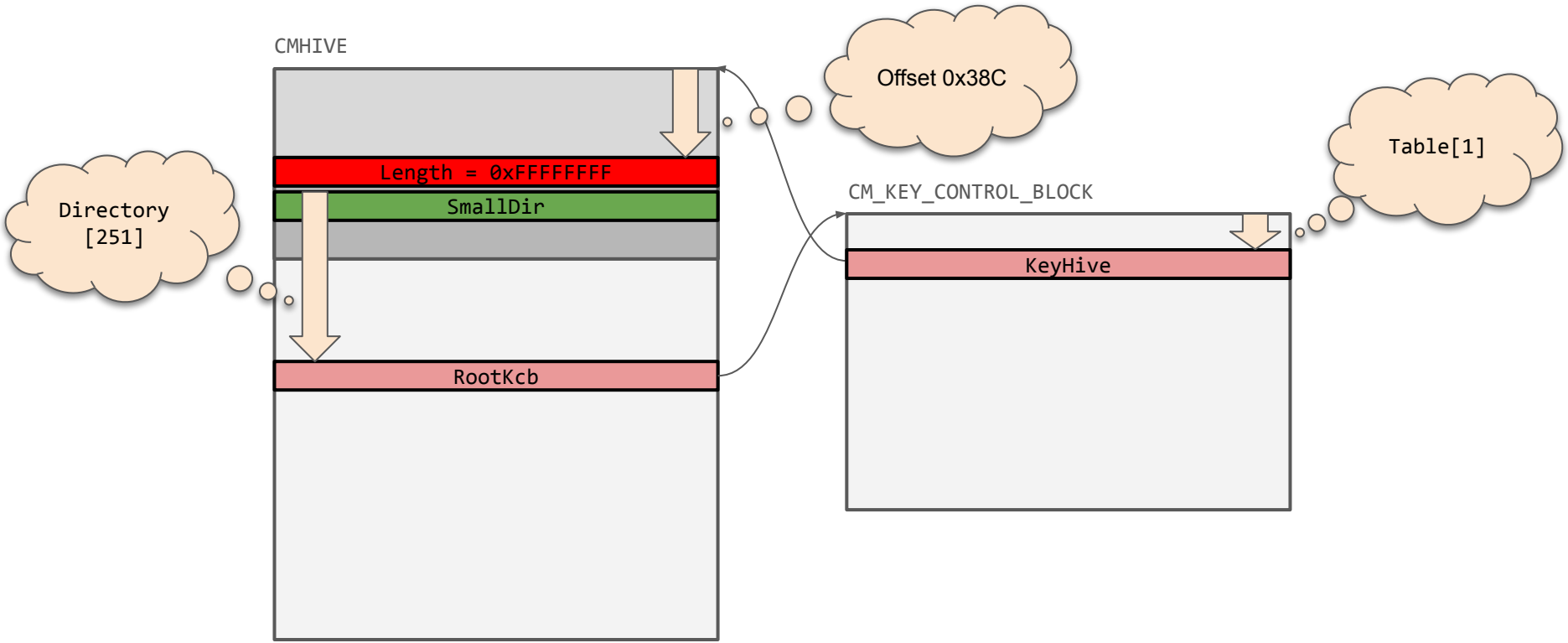
Length corruption cell walk (0x9F60138C)



Length corruption cell walk (0x9F60138C)

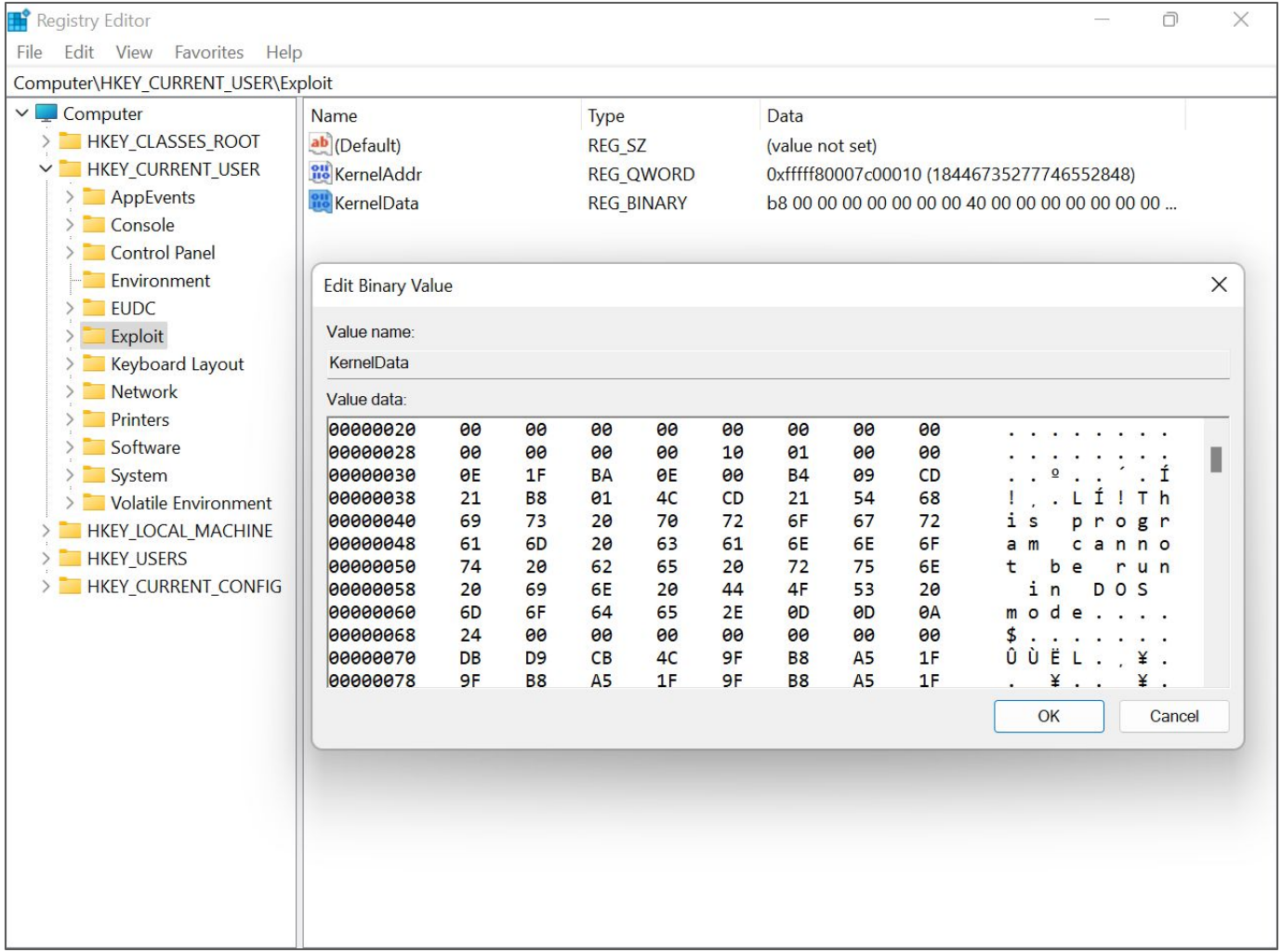


Length corruption cell walk (0x9F60138C)



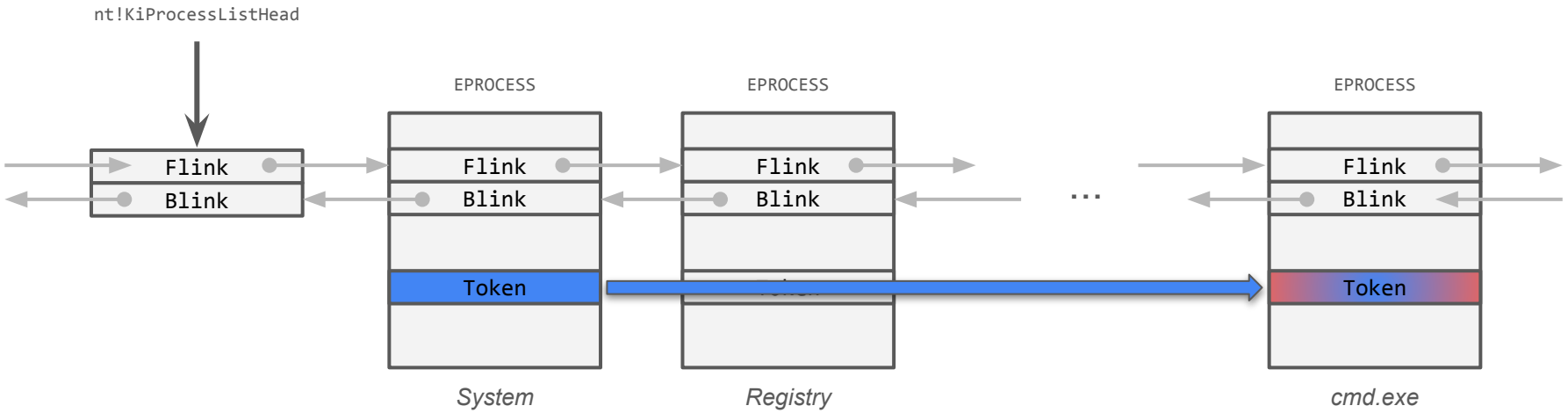
It works!

- By pure luck, we get to overwrite the reference length nanoseconds before it is used for sanitization and causes a crash
- The length is long lived, so a single corruption works for all future references to OOB indexes

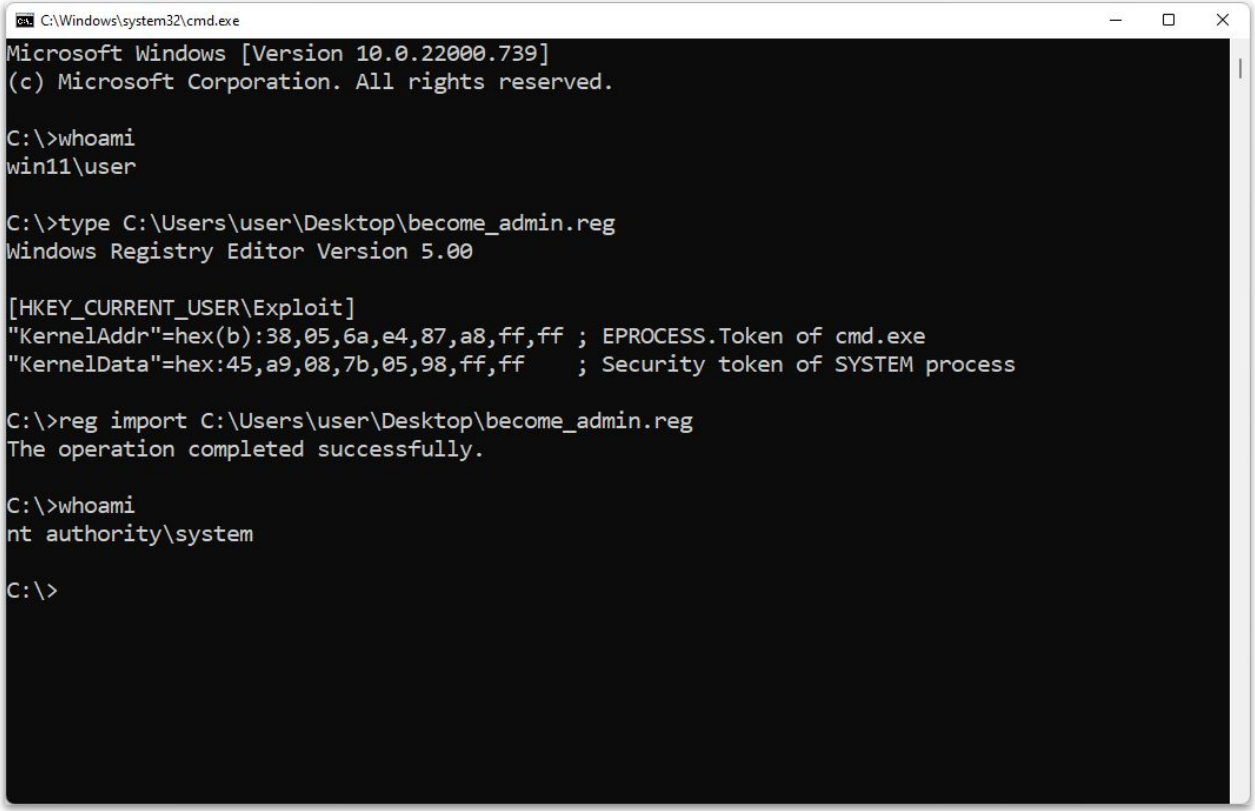

Registry Editor
File Edit View Favorites Help
Computer\HKEY_CURRENT_USER\Exploit
Computer
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
AppEvents
Console
Control Panel
Environment
EUDC
Exploit
Keyboard Layout
Network
Printers
Software
System
Volatile Environment
HKEY_LOCAL_MACHINE
HKEY_USERS
HKEY_CURRENT_CONFIG
Name Type Data
(Default) REG_SZ (value not set)
KernelAddr REG_QWORD 0xfffff80007c00010 (18446735277746552848)
KernelData REG_BINARY b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ...
Edit Binary Value
Value name:
KernelData
Value data:
00000020 00 00 00 00 00 00 00 00 . . . . . .
00000028 00 00 00 00 10 01 00 00 . . . . . .
00000030 0E 1F BA 0E 00 B4 09 CD . . º . . ´ . Í
00000038 21 B8 01 4C CD 21 54 68 ! ¸ . L Í ! T h
00000040 69 73 20 70 72 6F 67 72 i s p r o g r
00000048 61 6D 20 63 61 6E 6E 6F a m c a n n o
00000050 74 20 62 65 20 72 75 6E t b e r u n
00000058 20 69 6E 20 44 4F 53 20 i n D O S
00000060 6D 6F 64 65 2E 0D 0D 0A m o d e . . . .
00000068 24 00 00 00 00 00 00 00 $ . . . . . .
00000070 DB D9 CB 4C 9F B8 A5 1F Û Ù Ë L . ¸ ¥ .
00000078 9F B8 A5 1F 9F B8 A5 1F . ¸ ¥ . . ¸ ¥ .
OK
Cancel

Elevating privileges

- With the kernel base address and arbitrary read/write, all that's left to do is to spawn a command prompt and swap the security token:



Demo (CVE-2023-23420)



What's next

- There is room for mitigations and structural improvements
  - Hardening the cell index translation code
  - Separating hive mappings in *Registry* user-space and the kernel pools
  - Integer overflow checks in security descriptor reference counting
- Recommendations provided to Microsoft, some work already done
- Better fuzzing could yield interesting results
- Worth keeping an eye on any new registry features

Personal takeaways

- Long, persistent analysis pays off
- The registry is a fascinating research target with a lot of depth
- Weird machines, custom allocators and other constructs that live outside the realm of traditional exploit mitigations continue to persist in modern software
- There are insights to be gained from writing exploits even (or especially) when working on the defensive side