available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Recovering deleted data from the Windows registry☆

Timothy D. Morgan

VSR Investigations, LLC, Boston, Massachusetts, United States

Keywords:
Windows registry
Registry data structures
Registry data recovery
Registry forensics
Registry undelete

A B S T R A C T

The Windows registry serves as a primary storage location for system configurations and as such provides a wealth of information to investigators. Numerous researchers have worked to interpret the information stored in the registry from a digital forensic standpoint, but no definitive resource is yet available which describes how Windows deletes registry data structures under NT-based systems. This paper explores this topic and provides an algorithm for recovering deleted keys, values, and other structures in the context of the registry as a whole.
© 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.

1. Introduction

The Windows registry stores a wide variety of information, including core system configurations, user-specific configuration, information on installed applications, and user credentials. In addition, each registry key records a time stamp when modified which can aid in event reconstruction. This makes the Windows registry a critical resource for digital forensic investigations conducted against the Windows platform, as numerous researchers have shown.

Little information has been published by Microsoft related to the specifics of how registry information is organized into data structures on disk. Fortunately, various open source projects have worked to understand and publish these technical details in order to write software compatible with Microsoft's registry format. However, no public resource was yet available describing what happens to registry data when it is deleted under Windows NT-based systems, [1] let alone how a forensic examiner might reliably recover this information in the context of a registry hive. Here, we attempt to shed light on questions related to the deletion of registry data structures and suggest an algorithm for recovering this information.

2. Previous work

Harlan Carvey and Derrick Farmer have provided extensive information on how to interpret various registry settings from a forensic standpoint (Carvey; Carvey, 2005; Farmer, 2007). Dolan-Gavitt (2007–2008) has demonstrated how to recover registry hives and other data structures from system memory images. Numerous proprietary and open source tools provide access to registry internals (LastBit Corp, 2006; MiTeC; Morgan, 2007; Sharpe, 2002; Williams, 2000; Registry Tool). Additionally, Lee (2001) wrote Registry UnDelete which recovers deleted registry data from the older Windows 98/ME registry format.

Registry internal structures have been outlined by Russinovich (1999) and Probert. Further detailed work has been published by an unknown author in (B.D. WinReg.txt). These resources, together with some testing and validation, allow one to gain a clear understanding of the data structures of Windows NT-based registries.

3. Registry structure overview

Here, we briefly provide an overview of the internal data structures of the registry. Information on the specific layouts of each structure may be found in Morgan (2008).

☆ Throughout this paper, note that Windows, Microsoft, Windows 95, Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP, Windows Vista, and Windows Server are registered trademarks of Microsoft Corporation.

E-mail address: tmorgan@vsecurity.com

[1] Windows 95-based systems use a significantly different registry format and are out of scope for this work.

1742-2876/$ – see front matter © 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2008.05.002

The Windows registry is organized in a tree structure and is analogous to a filesystem. For instance, registry values are similar to files in a filesystem as they store name and type information for discrete portions of raw data. Registry keys are closely analogous to filesystem directories, acting as parent nodes for both subkeys and values. Finally, individual registry files (or "hives") are presented to users in Windows under a set of virtual top-level keys in much the same way that multiple filesystems in UNIX [2] are mounted under the same root directory.

The internal structure of Windows registry hives does, however, differ a great deal from typical filesystems. Registry hive files are allocated in 4096-byte blocks starting with a header, or base block, and continuing with a series of hive bin blocks. Each hive bin (HBIN) is typically 4096 bytes, but may be any larger multiple of that size. HBINs are linked together through length and offset parameters as shown in Fig. 1. Each HBIN references the beginning of the next HBIN in addition to indicating its distance from the first HBIN.

Within each HBIN can be found a series of variable-length cells. These cells are stored in simple length-prefix notation where each cell's total length (including the 4 byte length header) is a multiple of 8 bytes. Fig. 2 illustrates the layout of a typical HBIN.

The data portion of each cell contains either value data or one of several different record types. Possible record types include: key (NK) records, subkey-lists, value-lists, value (VK) records, and security (SK) records.

The data structure which ties all of these elements together is the key record. NK records contain a number of offset [3] fields to other data structures. These referenced structures may exist in any HBIN. In order to keep track of a key's subkeys, NK records reference subkey-lists which in turn reference a set of other NK records. NK records also store the offset of their parent NK record. These key-related pointers are illustrated in Fig. 3. Subkey-lists themselves are simple lists of pointer/hash tuples, sorted in order by the hash value which is based on the referenced subkey names. Multiple types of subkey-lists have been used in different versions of Windows, but they appear to retain the same basic structure. In early versions, including Windows 2000, subkey-lists use records with a magic number of "`lf`", where the hash in each element is calculated simply by taking the first four characters of the associated subkey's name. Newer "`lh`" records appear to simply use a more efficient hash algorithm, though we are currently unaware of the specifics. A third, somewhat rare subkey-list type has a magic number of "`ri`" and seems to implement an indirect block system, similar to what is found in some filesystems.

NK records also contain pointers to value-lists which in turn reference value records. Value-lists are similar to subkey-lists, but do not have hash values associated with them and are not sorted in any particular order. VK records contain some minimal metadata about a value along with the offset to yet another cell which contains the value's data. See Fig. 4 for a sample illustration.

[2] UNIX is a registered trademark of the Open Group.

[3] Nearly all offset values stored in cell records (whether in the NK records or elsewhere) are measured in bytes from the beginning of the first HBIN, not from the beginning of the file.

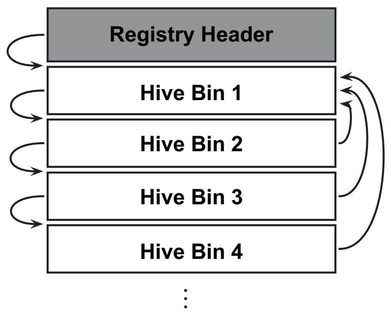

Fig. 1 – Top-level registry structure.

A final significant detail related to NK records is the inclusion of a modification time (MTIME) field. NK records are the only known record type, aside from the hive header, to contain any kind of time stamp. This field appears to be updated any time the NK record itself is updated (with some exceptions, detailed later), which includes changes to values and immediate subkeys.

Finally, a small number of security records are typically stored in a given registry hive and are referenced by NK records. SK records include a short header followed by a Windows security descriptor which defines permissions and ownership for local values and/or subkeys.[4] Multiple NK records may reference a single SK record which in turn stores a reference count to simplify deallocation.

To tie this all together, let us present a simple example. Suppose we had a simple registry hive rooted at a key named "`parent`", which has subkey named "`child`". Also suppose this subkey has a value stored under it named "`item`" which is a string, and this value's data is the string "`datum`". The user perspective of this structure is illustrated in Fig. 5. The registry records needed to support this simple path are illustrated in Fig. 6. In order to look up `\parent\child's item` value, one would first need to find parent, and locate its subkey-list. The hash value for "`child`" would be calculated and used to quickly narrow the list of NK records needing to be checked (i.e., the set of all colliding hashes). Searching this reduced list of NK offsets would yield an NK record which had the proper name. One would then traverse the `child` record's value-list sequentially, checking each referenced VK record to locate the proper value. If the `item` value's data was desired, the data pointer would be followed to the data record, unless a specific flag is set indicating that the data is stored in the offset field of the VK record, in which case it would be retrieved from there.

4. Testing methodology

In choosing a test platform for determining the changes induced by key and value deletions, one could go with one of (at least) two different routes. One could simply use the standard tools distributed with Windows (such as `regedit.exe` and `reg.exe`), or instead write a minimal utility to interface directly with the lowest-level Windows APIs. The former

[4] See Brown (2005) and Microsoft (2008) for more information on security descriptors.

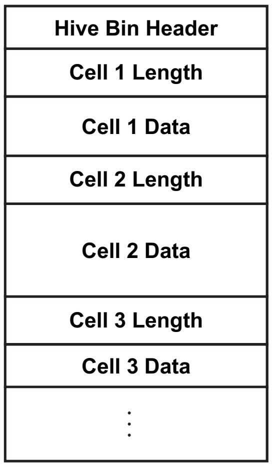

Fig. 2 – Hive bin structure.

approach carries the advantage that it is more likely to simulate real-world scenarios and would permit one to test the behaviors of common tools. However, the latter approach may allow a researcher to easily isolate the specific behaviors of the APIs and to perform tests more consistently across multiple versions of Windows. In addition, this second approach could yield more accurate results when considering the behaviors of third party software which also directly accesses the registry through the Windows APIs.

Due to time constraints, we chose to use the former method for this research. Both `regedit.exe` and `reg.exe` were used for each major test case. While some differences in behavior were observed between these two tools, very little variation was found in the core deletion behaviors for a given platform. However, as with any conclusions drawn from a sampling of test environments, investigators should retest these findings with platforms similar to those being investigated if valuable evidence were found in deleted registry data structures.

The following versions of Windows were tested: Windows 2000 Professional (SP0), Windows XP Professional (SP2), Windows Server 2003 R2 (build 3790), and Windows Vista (SP0). Tests were performed by making snap shots of the system registry file before and after specific changes were made. Using these snap shots, each registry data structure was analyzed for the changes which took place during deletion.

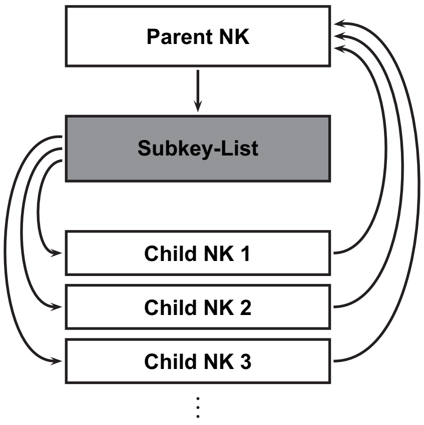

Fig. 3 – Key-related pointers.

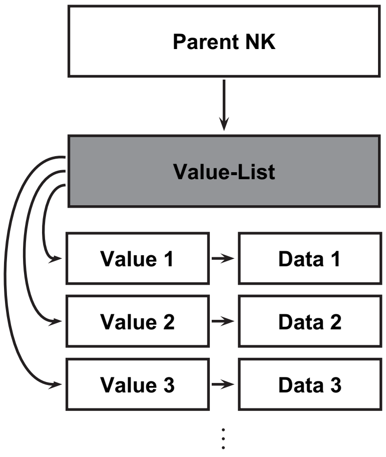

Fig. 4 – Value-related pointers.

Several sets of test cases were developed, each focusing on the behaviors related to a specific data structure and the outstanding questions related to deletion. Between three and six test runs were executed on each system, each with a batch of tests applied. Test cases that were suspected of interfering with one another were created under separate key trees, or tested in separate batches. As the initial set of test cases uncovered inconsistent or unexplained behaviors between platforms, additional test cases were run to refine the results. These additional test cases were then run again on all platforms to ensure consistency. An outline of the specific test cases is included below:

- Subkey-lists and value-lists
  - ◇ How are these lists are sorted?
    - Create a set of numbered elements under a key (e.g. "`subkey1`", "`subkey2`", ... or "`value1`", "`value2`", ...) in specific orders. Observe the ordering of elements to deduce the sorting rules.
  - ◇ How are these lists updated when elements are deleted?
    - Delete the previously created set of elements in specific orders. For instance, deleting value-lists elements in the reverse order of creation should result in no changes to the tail of the list.
  - ◇ Are these lists' cells ever shortened?
    - Create a large number of elements. Observe the size of the list's cell.
    - Next, delete all but one element. Compare the lists' current and previous sizes.
  - ◇ What happens to these lists when all elements are removed?
    - Create a list of elements. Observe the list and referencing NK record.

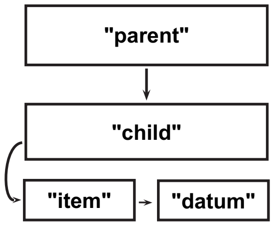

Fig. 5 – Simple example: logical view.

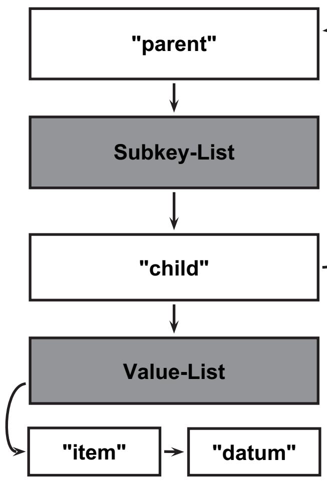

Fig. 6 – Simple example: physical view.

        • Remove all elements from the list and observe it again along with the referencing NK record.

- Values
  - ◇ What happens to VK and data records when values are deleted?
    - Create multiple VK records of different types and sizes. In particular, create some values with 4 bytes or less in data, and some with more than 4 bytes. Observe the VK and data records.
    - Delete the values. Observe any changes to the VK and data records.
- Security records
  - ◇ What happens to SK records when a referencing key is deleted?
    - Create a new key associated with an existing SK record. Observe the SK record.
    - Delete the referencing key. Observe any changes in the SK record.
    - Repeat test by creating a key with a dedicated SK record.
- Keys
  - ◇ What happens to NK records when keys are deleted?
    - Create multiple keys with different combinations of subkeys, values, and security records. Observe the NK records.
    - Delete keys and observe changes in the NK records.
  - ◇ What happens to a tree of data structures when the root key is deleted?
    - Create a tree of keys and values several levels deep. Record all data structures.
    - Delete the root key. Observe changes in all data structures. Compare these changes to the changes that occur when the structures are individually deleted.

While this set of tests may not encompass every possible behavior of data structure deletion, it does provide sufficient information to develop a general picture of registry deletion.

5. Registry deletion behaviors

The most basic behavior to understand in analyzing registry deletions is how the registry manages unallocated cells. As new records are added and free space is allocated, existing empty cells may be split if they are much larger than the required space. However, as cells are later deallocated, any adjacent unallocated cells would need to be merged in order to prevent serious fragmentation. Indeed, this is how the registry manages unallocated space. When a given cell is decommissioned, the cells directly before and after are checked. If either (or both) of these cells are already unallocated, the cells are merged together by updating the header length value of the earliest cell. The other cells' lengths are not updated. This process makes recovery somewhat complicated, since structures cannot be found at specific offsets within a cell.

We have found that the majority of structures stored in cells are preserved when they are deallocated; however, certain key pieces of information are explicitly destroyed or partially corrupted during deletion, and these behaviors vary from record type to record type. Here we outline the changes that take place for each record. More detailed information on each structure may be found in Morgan (2008).

Since registry keys act as the glue that ties registry elements together, they are of primary importance. When a key is deleted, its NK record is changed in a number of ways. For one, the pointer which references subkey-lists is destroyed (overwritten with `0xFFFFFFFF`) and the stored number of subkeys is set to 0. In addition, the pointer to a key's security record is similarly destroyed. If a key has subkeys, the record's modification time is updated, otherwise it is not. The only known exception to this rule is found on Windows 2000 where a key with subkeys does not have its modification time updated at all when it is deleted.

When analyzing changes to subkey-lists, we must consider two cases: first, where the parent key (and therefore all subkeys) is deleted; and second, where some number of subkeys are deleted. As it turns out, these two cases are actually very similar. When a single subkey is deleted, the element is removed from the list and the resulting list is rewritten to the cell. The remaining free space in the cell is not wiped and in

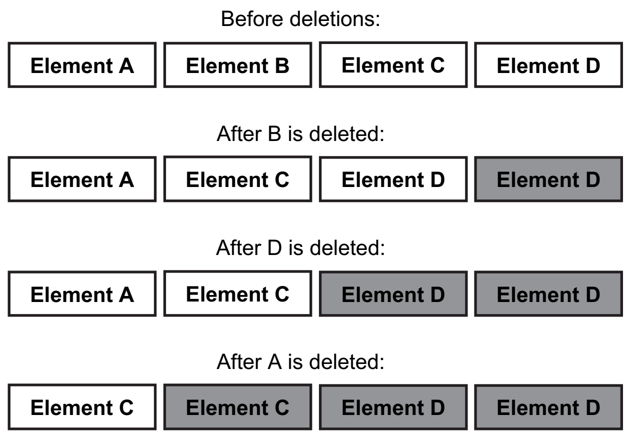

Fig. 7 – Hypothetical subkey deletion sequence.

```
Let LC be the list of all unallocated cells in registry file R, stored as a list of variable-length segments
Let LK be an empty list of NK records
For each segment, C, in LC:

    Search C for NK records in increments of 8 bytes
    For each NK record, K in C:

      Append K to LK
      Remove K's segment from LC (This may require segments in LC to be split)

For each NK record, K, in LK:

    Recursively follow K's parent links, storing each key name with K, until:

      The root NK record is found, resulting in complete path reconstruction
      Or, a non-NK record is found, resulting in partial path reconstruction
      Or, the registry path depth limit is reached, resulting in no path reconstruction

    If K's value-list, VL is in LC and is intact:

      Remove VL's segment from LC
      For each value record, V, in K's value-list:

        If V occurs in LC and is an intact VK record:

        Remove V's segment from LC
        Store V along with K
        If V's data cell, D, occurs in LC and is intact:

          Remove D's segment from LC
          Store D along with V

    Output K and all associated data
For each remaining segment, C, in LC:

    For each remaining intact VK record, V, in C:

      Remove V's segment from LC
      If V's data cell, D, occurs in LC and is intact:

        Remove D's segment from LC
        Store D along with V
      Output V

    For each remaining intact SK record, S, in C:

      Remove S's segment from LC
      Output S

For each remaining segment, C, in LC

    Output C
```

no tests was the cell shortened to conserve wasted space. Consequently, a number of subkey-list elements (each 8 bytes in size) can be found at the end of a subkey-list that has been shortened. Unfortunately, this information is typically not very useful because in most cases the last element in the subkey-list will be repeated over and over, unless it was deleted midway through a set of deletions, at which point the second to last element would begin the repetitions as internal elements continue to be deleted. Fig. 7 illustrates how a subkey-list would look at each step if elements B, D, and A were removed, in that order, from an original list of: (A,B,C,D). When it comes to deletion of a parent key, our experiments indicated that all children are merely deleted in sequence with some unknown or arbitrary order. This causes the subkey-list to be repeatedly rewritten with each successive deletion, corrupting the majority of records in most cases. The number of elements in the subkey-list is also reduced to 0 upon deletion of a parent key.

The changes which occur to value-lists during deletion differ somewhat from those of subkey-lists, even though their structures are almost identical. As with subkey-lists, when values are deleted from a key the elements are removed and the list is simply rewritten. Here, slack space is also not wiped and value-list cells do not appear to be shortened as elements are removed, which matches the general behavior of subkey-lists. However, when a value-list's parent NK record is deleted, value-lists are not modified beyond having their holding cell deallocated; all links to the (now deleted) VK records are left intact.

In general, VK records and the data cells they reference are not altered when they are deleted. The only exception to this rule is on Windows 2000 where the first 4 bytes of these cells (for both VK records and data cells) are overwritten with `0xFFFFFFFF`. In the case of the VK record, this corrupts both the 2-byte magic number and the 2-byte length for the value's name. In the case of a data cell, there's no way of knowing what data would be lost. Fortunately, this behavior was only observed on Windows 2000 and may be indicative of a bug on that platform.

Finally, there are few changes associated with the deletion of security records. Of course since these records may be referenced by multiple keys, they are only deleted when all keys referencing them are also deleted or are set to reference other SK records. Our observations indicate that nothing changes in SK records when this occurs. In fact, not even the reference count (which would store a value of 1 before the final parent key deletion) was updated to 0 when the SK record was deallocated.

6. Data recovery and analysis

While it is unfortunate that certain critical portions of records are corrupted during deletion, it seems that the designers of the registry left behind just enough information to associate the deleted structures. While the links from a parent NK record to child NK records are apparently destroyed beyond repair, we are lucky that each NK record stores a pointer to its parent which is left intact during deletion. Based on this and the fact that the links from an NK record to its values are generally unmodified, we have developed a high-level algorithm for recovering registry data with context (see Algorithm 1).

Algorithm 1. Data recovery with context

Algorithm 1 relies on a method to validate that each located data structure is intact; something that is somewhat difficult to do. Performing checks to verify record length and signature is obviously straight-forward, but as far as we know, there are no checksums or other integrity mechanisms built into any cell records. Since most records contain a number of offset fields, a recovery tool could at least verify that each provided offset makes sense (i.e., is within the bounds of the file), which would help to prevent confusion of old data with old records.

However, against an adversary determined to confuse recovery, any isolated record validation would ultimately fail. For instance, an adversary could create a binary value and store a whole NK record in that space, and then remove it. Algorithm 1 would likely pick that record up as being a true deleted NK record. This could be achieved without direct access to registry files, which means even lower privileged users could perform such an attack. Unfortunately, little can be done to avoid this problem, since the links from the more authoritative NK records to deleted NK records are broken. Attempting to find uncorrupted subkey-lists could yield structures referencing a given NK record, but these may be few and far between. Another approach would be to adapt the algorithm to start by isolating values and their data cells and only search what is left over for NK records and other data structures. However, this would degrade recovery performance in the more common case where there is no adversary.

7. Experimental results

We have implemented Algorithm 1 as an extension to RegLookup (Morgan, 2007). This new command line tool, `reglookup-recover`, attempts to recover as many intact

Table 1 – New user entries (long values truncated)

Path	Type	Value	MTIME
/SAM/Domains/Account/Users/000003EC	KEY		2008-05-04 23:43:19
/SAM/Domains/Account/Users/000003EC/F	BINARY	\x02\x00\x01\x00...	N/A
/SAM/Domains/Account/Users/000003EC/V	BINARY	\x00\x00\x00\x00...	N/A
/SAM/Domains/Account/Users/Names/Kobayashi	KEY	N/A	2008-05-04 23:43:19
/SAM/Domains/Account/Users/Names/Kobayashi/	0x03EC	N/A	N/A
/SAM/Domains/Builtin/Aliases/Members/ S-1-5-21-343818398-573735546-839522115/000003EC	KEY	N/A	2008-05-04 23:43:19
/SAM/Domains/Builtin/Aliases/Members/ S-1-5-21-343818398-573735546-839522115/000003EC/	EXPAND_SZ	! \x02\x00\x00 \x02\x00\x00	N/A

Table 2 – Deleted user entries

Offset	Length	Type	Path	MTIME	Num. Of Values	Data Type
0x1C88	0x58	KEY	/SAM/SAM/Domains/Account/Users/Names/Kobayashi	2008-05-04 23:43:19	1	N/A
0x4F58	0x58	KEY	/SAM/SAM/Domains/Builtin/Aliases/Members/ S-1-5-21-343818398-573735546-839522115/000003EC	2008-05-04 23:43:19	1	N/A
0x1CE8	0x18	VALUE		N/A	N/A	0x03EC

structures as possible and writes them to the standard output in a CSV-like format. Preliminary testing in a lab environment has shown that Windows does a reasonably good job at avoiding registry fragmentation over short periods of time, which means deleted data structures have a relatively short lifetime in an active registry. However, certain situations exist which allow for greater longevity. For example, the Security Accounts Manager (SAM) registry file is relatively inactive, since it is small and stores mostly user account and group information which changes infrequently. One might guess that deleted structures here likely have a long lifetime.

Using the reglookup and reglookup-recover command line tools, we tested a SAM-related scenario under Windows XP. First, a snapshot of the SAM registry was taken. Next, a new user was created with administrative privileges and another snapshot was taken. Finally, the new user was deleted, and a final snapshot was taken of the SAM. Between steps, the system was shut down in order to take the snapshot, and then booted again for the next step.

Upon creating the new user account, which was named "Kobayashi", three keys and four values were added to the SAM registry (see Table 1). A number of other registry keys and values were modified as well, but these weren't directly relevant to this test. Note the odd data type for the /SAM/Domains/Account/Users/Names/Kobayashi/ value. Windows does a peculiar thing by actually storing user ID numbers in the value type field of the SAM registry. Also note that this value has no name. In Windows, the value would be considered the "(default)" value for the key, but in order to avoid any possible naming ambiguity, reglookup simply returns a path with a trailing "/" and an explicit type field.

After deleting the user, the newly created keys were all deleted. The set of keys and values in the SAM registry was identical to what it was before the user creation, though once again, a number of values and key MTIMEs were updated. We then used the reglookup-recover tool to extract a subset of the keys and values that were deleted when the user was removed. These results are listed in Table 2 and allow us to see that the user once existed. Because neither the "Kobayashi" or "000003EC" keys had any subkeys, their MTIMEs are set to when they were last modified before deletion (in this case, when the user was created). The recovered value record has no path because the value-list linking the Kobayashi key to it had been overwritten. Luckily in this case, the value's type matches with the name of the 000003EC key under the local domain's group membership area, which would provide an investigator with at least some context.

In the general case of trying to find useful information in deleted registry keys, we can first try to determine what percentage of typical system registries are unallocated and intact. The definition of "typical" is of course open to interpretation, but as a starting point we gathered some statistics from a number of Windows systems, which is summarized in Table 3.

To generate these data, an aggregate of registry files was tested from each installation which included the system, software, SAM, and security registries. (User-specific registries and default or backup registries were not included.) The percentage of total unallocated space and percentage of successfully parsed unallocated space are listed. Finally, the average number of bytes per unallocated cell is included in Table 3 to help explain the results.

The algorithm performs remarkably well on systems C and D, extracting the vast majority of deleted space. However, it does not fair well on systems A and B. The cause for this on system A is the very small mean unallocated cell sizes. Registries that have been in operation a long time

Table 3 – Unallocated space and intact structures

System	A	B	C	D
Windows version	Server 2003 SE (build 3790)	Server 2003 EE (build 3790)	Server 2003 EE (build 3790)	2000 Professional SP4
System purpose	IIS-based web server, heavy use	MSSQL-based DB server, moderate use	Testing platform, light use	Desktop, light use
System lifetime	3.3 years	2.7 years	2–3 months	3.5 years
Aggregate hive size (MB)	18.1	36.7	30.7	16.8
Unalloc. space (%)	0.31	7.4	5.8	2.9
Unalloc. and intact (%)	0.097	0.21	5.3	2.7
Mean unalloc. cell size (bytes)	18.64	361.8	324.3	3427

tend to accumulate a large number of 8 and 16 byte unallocated cells which contain only 4 and 12 bytes of meaningful data. The cause for the rather poor performance on system B is not immediately clear. By looking at system B's unallocated cells, we found that like system A, it had a large number of 8 and 16 byte cells, but this was offset by an additional set of 4064 byte cells at the end of the registry files. These large cells (which fill an entire HBIN's data area) were mostly filled with NUL bytes, indicating they had not yet been used. It is possible that on system B, Windows chose to pre-allocate a number of HBIN blocks, skewing the basic statistics presented here.

Overall, it seems that the amount of information available can vary widely from one system to another. The algorithm presented here seems to perform well in extracting this information, and in some instances may provide critical clues which would have otherwise been difficult to obtain in context.

8. Antiforensics

As with many forms of antiforensics, covering one's trail in the Windows registry may be achieved with some of the simplest of tools. In order to eliminate previously deleted data, a user would do a good job of it by simply creating many keys and small values to fill any fragmented free space. This may also have the benefit of overwriting key modification times if the values were created in a variety of locations. Another approach to overwriting deleted structures would be to use one of several tools available (Abexo; Auslogics; Hederer, 2005; iExpert Software) for defragmenting the registry, in which case it would be difficult to prove that the user was intentionally trying to destroy evidence.

A technically savvy adversary may also attempt to hide information in the registry where standard tools will not find it. For instance, Franchuk (2005) described a way to hide registry keys from certain versions of `regedt32.exe` by simply creating keys with long names, though these keys are easily found by more robust registry tools. Another way to hide data within the registry would be to place it within the registry file header. Here, there are nearly 4000 bytes of apparently reserved space that should be ignored by Windows and most registry tools. However, it would be easy for investigators to locate any hidden data in the header with a simple hexadecimal editor. Finally, an adversary could also traverse the list of cells in the registry, select an unallocated one, and simply change the length field to a negative value. This would mark the cell as allocated, even though no other registry structures reference it. Preliminary tests showed that Windows simply ignores these cells, as expected. Detection of this kind of data hiding is much more difficult, since it would require an investigator to build a list of all allocated cells and then ensure that at least one existing record references each individual cell. Of course there are likely other methods for hiding data within the registry. An obscure key or value placed deep within a little-known configuration tree would likely avoid notice in most cases due to the poorly documented nature of the registry.

9. Further research

Certain areas of Windows registry behavior present outstanding questions. Perhaps most unknowns lie in the area of specific tool behavior. For instance, simple tests reveal that `regedit.exe` and `reg.exe` do behave somewhat differently, particularly in creating new keys and values. For instance, when a new key or value is created with `regedit.exe`, a "`New Key #1`" or "`New Value #1`" structure is created immediately as a place holder with the expectation that a user will rename it. This typically leaves behind temporary key and/or value data structures. Additionally, `reg.exe` appears to always create at least one value, the "`(default)`" value, in the keys it creates. Other tools do not exhibit this behavior. These minor differences could be useful to investigators in determining how keys and values may have come to be created. Similar differences between tools may appear in relation to key or value renames, moves, or copies. For instance some tools may use low-level system calls to rename a value, while others could simply copy and delete.

It may also prove useful for investigators to understand the typical longevity of deleted data. In trying to determine this, one would need to ask questions such as: How fragmented does the registry become over time? What is Windows' reallocation strategy for deleted space? How does longevity of deleted cells differ based on their size? All of this information would help to provide a complete picture of Windows registry behavior.

10. Conclusions

This research draws from a fragmented and disparate set of sources to present digital forensic examiners a more complete view of the Windows registry's internal data structures. In using this information to analyze how Windows deletes registry keys, values and security properties, we have discovered that deleted information is recoverable if it has not been overwritten, though it may not be entirely trustworthy. We also found that the platforms tested behave quite similarly and should simplify the implementation of recovery tools.

Acknowledgements

We would like to thank the following individuals for their invaluable contributions to this paper: Jason DeMent, George Gal, Jason Morgan, and Joan Morgan.

REFERENCES

Abexo. Abexo Registry Cleaner. Available from: http://www.abexo.com/registry-cleaner.htm.

Auslogics. Auslogics Registry Defrag. Auslogics Software Pty Ltd. Available from: http://www.auslogics.com/registry-defrag.

Brown Keith. What Is A Security Descriptor.pluralsite.com. Available from: http://www.pluralsight.com/wiki/default.

aspx/Keith.GuideBook/WhatIsASecurityDescriptor.html; 2005 [ported 18.01.2005, accessed 09.03.08].

Harlan Carvey. Windows incident response: registry articles. Available from: http://windowsir.blogspot.com/search/label/Registry.

Harlan Carvey. The windows registry as a forensic resource. Digital Investigation. Available from: http://authors.elsevier.com/sd/article/S1742287605000587, September 2005;2(3): 201–5.

LastBit Corp. Alien Registry Viewer. Available from: http://lastbit.com/arv/; April 2006.

B.D. WinReg.txt. Available from: http://home.eunet.no/pnordahl/ntpasswd/WinReg.txt.

Dolan-Gavitt Brendan. Push the red button: registry articles. Available from: http://moyix.blogspot.com/search/label/registry; 2007–2008 [accessed 04.05.08].

Farmer Derrick. A forensic analysis of the Windows registry. Available from: http://eptuners.com/forensics/Index.htm; 2007.

Franchuk Igor. Miscrosoft Registry Editor 5.1/XP/2K long string key vulnerability. Available from: http://lists.grok.org.uk/pipermail/full-disclosure/2005-August/036448.html; http://seclists.org/fulldisclosure/2005/Aug/0801.html; August 2005.

Hederer Lars. NTREGOPT. Available from: http://www.larshederer.homepage.t-online.de/erunt/index.htm; September 2005.

iExpert Software. Free Registry Defrag/Compact. Available from: http://www.registry-clean.net/free-registry-defrag.htm.

Lee Paul. Registry UnDelete. Available from: http://paullee.ru/regundel; September 2001.

Microsoft. SECURITY_DESCRIPTOR structure. Available from: http://msdn2.microsoft.com/en-us/library/aa379561.aspx [last updated 19.02.08].

MiTeC. Windows registry recovery (analysis tool). Available from: http://mitec.cz/wrr.html.

Morgan Timothy D. RegLookup. Available from: http://projects.sentinelchicken.org/reglookup/; March 2007.

Morgan Timothy D. The Windows NT registry file format. Available from: http://www.sentinelchicken.com/research/registry_format/; August 2008.

David B. Probert. Windows kernel internals: NT registry implementation. Available from: http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/09-Registry/Registry.pdf.

Registry Tool. Available from: http://www.registrytool.com/.

Russinovich Mark. Inside the registry. Windows NT Magazine, Microsoft. May 1999 edition. Available from: http://www.microsoft.com/technet/archive/winntas/tips/winntmag/inreg.mspx%3Fmfr=true May 1999.

Sharpe Richard. editreg.c. Available from: http://websvn.samba.org/cgi-bin/viewcvs.cgi/trunk/source/utils/editreg.c%3Frev=2&view=markup; 2002.

Williams Nigel. dosreg.c. Available from: http://www.wednesday.demon.co.uk/dosreg.html; 2000.

Timothy D. Morgan taught himself BASIC programming when he was twelve years of age and has been studying computers ever since. After earning his Computer Science degrees (B.S., Harvey Mudd College and M.S., Northeastern University), he has worked to secure his clients' systems and applications as a security consultant. Tim has conducted dozens of digital investigations over the past several years and recently co-founded VSR Investigations, LLC where he leads the digital forensics practice. He is also the author and maintainer of several open source forensics tools, namely GrokEVT, RegLookup, and tableau-parm.