FORENSIC ANALYSIS OF UNALLOCATED SPACE IN WINDOWS REGISTRY HIVE FILES

By

Jolanta Thomassen

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

04/11/ 2008

ABSTRACT

FORENSIC ANALYSIS OF UNALLOCATED SPACE IN WINDOWS REGISTRY HIVE FILES

By

Jolanta Thomassen

Windows registry is an excellent source of information for computer forensic purposes. The registry stores data physically on a disk in several hive files. Just like a file system, registry hive files contain used and free clusters of data. So far, the focus in Windows registry forensics has been on active keys and values that can be viewed with Windows registry editors. It has been a mystery, whether deleted or updated keys can be recovered from registry hive files, in a similar way that deleted files can be recovered from a file system.

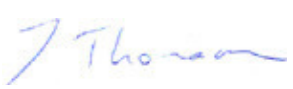
This project studies the physical structure of the binary registry hive files and shows that previously deleted or updated keys and their values indeed remain in the unallocated space until they become overwritten. The project proposes an algorithm for computing of unallocated space in registry hives as well as methods for recovery of deleted keys remaining in the unallocated space.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,



Jolanta Thomassen

ACKNOWLEDGEMENTS

I would like to express my gratitude to Laureate Online Education and the University of Liverpool for allowing me to study online for a Masters degree. I would like to thank Basem Shihada for agreeing to be my dissertation advisor and Harlan Carvey for agreeing to be my sponsor.

Last but not least I am truly grateful to my husband and daughter for supporting me during the past two years. Besides being supportive, they have both showed how much they have been inspired by me: my daughter excels at her school while my husband has just enrolled in an MBA programme.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

Chapter 1. INTRODUCTION

1.1 Scope

Windows registry stores keys and values in physical binary hive files. Used (or allocated) space in registry hive files contains active registry keys and values. The remaining space in registry hives constitutes an unallocated space.

The project's goal is to find and examine the unallocated space in registry hive files and to recover any relevant data remaining there. The project's scope is limited to forensic analysis of hive files performed in a postmortem investigation, where an investigator works with a disk image taken from a shutdown system. Hive files on a running system may contain added information, such as keys describing current hardware settings on a running machine, and therefore have a different structure than hive files copied from a shutdown system. Examination of unallocated space of registry hives requires considerable knowledge of the structure of hive files and how keys and values are stored and relate to one another. The project studies the allocated space in binary registry hive files to gain enough information to both identify the unallocated space and to interpret data remaining there. The final goal of the project is to recover any data that may be of interest to forensic investigators from the unallocated space.

1.2 Problem Statement

Harlan Carvey, the sponsor of the project, has been involved in incidence response and computer forensic analysis since 2000. He has been widely published in "Security Focus", "Information Security Bulletin", and "Digital Investigation Journal". He is the author of "Windows Forensics and Incident Recovery" published in July 2004 by AWL, and the au-

thor of "Windows Forensic Analysis" published in April 2007 by Syngress/Elsevier. His current interests include forensic analysis and incident response, and registry and physical memory analysis. (Carvey, 2008c)

"From the perspective of forensic analysis, there are several areas of interest surrounding the Windows Registry. Most of the attention focuses on what's in the actual hive files themselves, and some attention has recently been focused on extracting Registry data (hive files, keys, values, etc) from within memory dumps and the pagefile. However, little if any attention has been given to remnants left behind in hive files that are not part of the active hive file itself; Registry "slack space". These areas may possibly reveal indications of previously installed applications or user activity, and be extremely valuable to investigations, particularly those pursued by law enforcement." (Carvey, 2008a)

1.3 Approach

The first phase of the project studies the structure of registry hive files by gathering and corroborating the existing knowledge. In the second phase, a method for calculation of unallocated space is developed. Subsequently, the final phase focuses on recovery of relevant data from the unallocated space.

Since there is no previous research of the unallocated space in registry hive files, the project takes an experimental approach, where stages of analysis, design and implementation are performed simultaneously and refined continually. All tools are implemented in Perl to accommodate the sponsor of the project, who implements all his forensic tools in this language.

1.4 Outcome

The project provides a documentation of the structure of relevant parts of registry hive files, based on examination of the available set of test hive files. Since the test data set is limited, the project delivers added tools that can examine if a hive file conforms to the assumptions made about the hive file structure. If any of the assumptions fail in further tests, the final algorithm can be refined.

The project proposes and implements an algorithm for calculation of unallocated space and recovery of data. Finally, the project shows that keys can in fact be recovered from the unallocated space; however, there is a possibility that, despite careful validation, the recovered data might have been partially overwritten.

Chapter 2. BACKGROUND AND REVIEW OF LITERATURE

The Windows registry is *"A central hierarchical database used in Microsoft Windows 98, Windows CE, Windows NT, and Windows 2000 used to store information that is necessary to configure the system for one or more users, applications and hardware devices. The Registry contains information that Windows continually references during operation, such as profiles for each user, the applications installed on the computer and the types of documents that each can create, property sheet settings for folders and application icons, what hardware exists on the system, and the ports that are being used."* (Microsoft, 2002)

Available research in Windows registry forensic focuses on identification of keys that are relevant for forensic examiners and possibilities for data hiding. Since registry hive files can have hundreds of thousands of entries, forensic investigators need to familiarize themselves with the registry to know where to look for evidence. Carvey (2007b) shows why registry information is valuable to forensic examiners and explains where to look for information, such as system configuration and user activities. His work continues in his current development of the `Regripper` tool (Carvey, 2008d), which correlates types of information with registry keys, so forensic examiners can extract only data that is relevant to their investigation. Chang (2007) focuses on registry entries that provide the information about OS installation and last shutdown times, the system time zone information as well as information about mounted storage devices. Wong (2007) discusses keys of interest and opportunities for data hiding in the registry. An adversary can for example insert text data as a binary value, although the data would be disclosed if the value was displayed in a hex editor. A more sophisticated technique would involve converting a text string into a hexadecimal notation and storing it in a value of a string type. Malware can be hidden and run covertly from keys that automatically execute programs.

Microsoft registry editors: regedit.exe and regedt32.exe stop displaying key values if they encounter a key with a name longer than 256 characters (Wong, 2007). Regedit.exe has another limit in that it can only perform searches on data stored as strings (Microsoft, 2006). This implies a need for independent tools that can parse registry hive files.

Windows applications access the registry using a set of Microsoft registry access functions (MSDN, 2008c). Microsoft does not provide any documentation of registry hive files or registry access functions; neither does it provide any tools that can recover deleted data from the registry hives. However, a few tools extract active registry keys and values from registry hives without use of Microsoft's API functions. They shed some light on how binary registry hives can be parsed. Carvey (2007a) implements an offline registry viewer that parses binary registry hive files and retrieves "live" keys, their timestamps, and values. Another tool that, will be used as a reference, implements a library of C functions that access data in registry hives (Nordahl-Hagen, 2008).

A few takes on documentation of registry hives are also available. Russinovich (1999) provides a first explanation of the registry's building blocks: physical data blocks and logical bins and cells that contain registry keys and values. Further publications provide descriptions of how different cells are structured, the first one published by an unknown author B.D. (n.d.) and a more comprehensive one by Clark (2005). Both publications are incomplete and limited to Windows 2000 versions, thus excluding new artifacts introduced in Windows XP and Vista.

Finally, registry cleaner tools claim to manipulate the unallocated space is in registry hive files. For example, a tool `NTREGOPT` by Hederer (2005b) claims to remove registry "slack" space that may contain previously deleted keys and values, by rebuilding the registry tree.

Chapter 3. ANALYSIS AND DESIGN

The main purpose of this project is to decide whether any information about previously deleted or changed keys can be recovered from registry hive files. The registry contains not only information about keys and their values but also data used by the Windows Operating System to access and manipulate hive files, including in-memory operations. The focus of this project is to study hive files in a postmortem investigation, where an investigator works with a disk image of a shutdown system. The project will focus on data that seems most relevant for a computer forensic examination: keys, their values, and timestamps - critical information for investigations that need timeline information.

Registry hives have a tree structure where the root key points to its subkeys, which point to their child subkeys. Keys also contain offsets (pointers) to their values, security settings and class name. Information contained in the registry is stored in cells - logical units of data that also contain information about their size. All the cells referred to by the registry tree form an allocated space within the registry hive file. The questions to be answered are whether unallocated space that is not referred to by the registry tree contains any useful information about deleted or updated keys and values, and whether and how this information can be recovered.

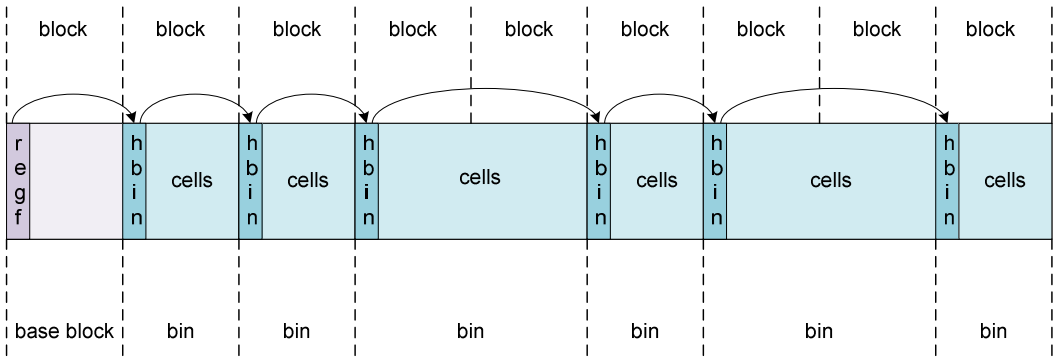
To solve the problem, one has to gain an understanding of how hive files are structured. Microsoft have never released any documentation of registry hives, however some takes on at documenting hive files have been made, unfortunately all incomplete and sometimes inaccurate. The analysis and design will consist of two main parts: parsing of the registry tree to learn how keys and values are stored in registry hive files, followed by calculation of unallocated space. Finally, if possible, deleted or updated keys will be recovered. The main purpose of the first phase is to confirm, verify, and extend the available information about the structure of registry hives, and the following stages will use the gained knowledge to calculate and examine the unallocated space.

Because of the exploratory nature of the project, analysis, design and prototyping have been performed in parallel and revised continually; the following analysis and design represent therefore the combined knowledge gained in these iterations.

3.1 Registry hive structure

The first description of how the Windows Operating System physically manages the registry is provided by Russinovich (1999). The registry hive file contains blocks, bins and cells. Blocks are 0x1000 (4096) bytes in size, and registry hive files are therefore always a multiple of 0x1000 bytes in size. The first block of the registry is called a base block, and all the following blocks contain bins, that contain cells. The base block contains general information about the hive file. Bins are logical units of data and can occupy one or several blocks. All bins contain a bin header followed by multiple cells. Cells are the smallest units that contain registry data, and their size is always a multiple of 0x8 bytes. The last cell of a bin always fills out the remaining space of a bin, meaning there is never any space in a bin that does not belong to a cell.

Figure 1 Registry hive file

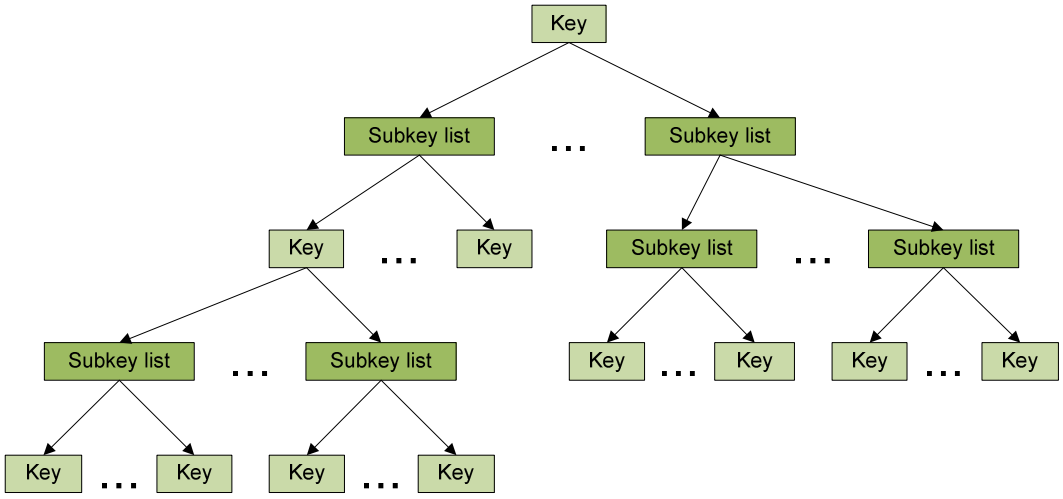


There registry contains the following types of cells (some have signatures while others do not):

- key cell "nk"
- class name cell
- security descriptor cell "sk"
- subkey list cell "lf", "lh", "ri", "li"
- value list cell
- value cell "vk"
- value data cell

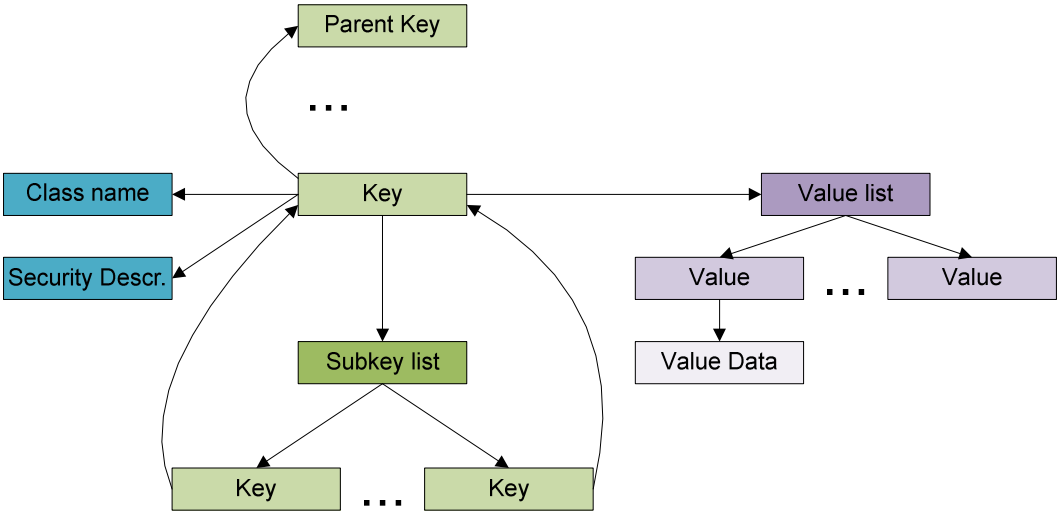
Registry key cells form an unbalanced tree, where the root key cell forms the root of the tree, and where keys contain pointers to their subkeys (child keys). Each key also contains a pointer (offset) back to its parent key. Subkeys are sorted alphabetically; following down the tree in a preorder manner, keys can be retrieved in alphabetical order - a Windows feature that allows efficient search for a key given its full name. (Russinovich, 1999)

Figure 2 Registry tree



Besides pointers to subkeys, each registry key contains pointers to its parent key, value list, class name (if any), and security descriptor. Value lists contain pointers to key values, and if value data is stored in a separate cell, value cells point to value data cells.

Figure 3 Registry key



The sources used for documentation of base block, bin headers and cells include Clark (2005), B.D.(n.d), Carvey (2007a) and Nordahl-Hagen (2008). All of those sources are closely connected in that Nordahl-Hagen (2008) uses findings by B.D. (n.d.), while Carvey (2007a) uses findings published by Nordahl-Hagen (2008). The following documentation combines those sources with experimentation, coding and manual examination of sample binary hives. The following description of cell structures is strictly limited to records that are necessary for the purposes of the project (parsing of the registry tree, computing of unallocated space and recovery of keys, their timestamps and their values).

Two important points to note in the following cell descriptions:

- All offsets are relative to the first bin of a hive file, which follows immediately after the base block; to calculate file offsets 0x1000 bytes must be added to all extracted offsets

- Cell sizes, interpreted as signed long integers, are negative numbers. The size of a cell is therefore an absolute (or a negated) value of the extracted size.

The base block contains general information about a hive file and has a signature of "regf". From the base block, it is possible to extract the name of the file including its local path (except for `SYSTEM` files, where the local path is not included), the timestamp and the offset to the registry root key.

The file name is often shortened by the Windows operating system (first characters are cut off), because the maximum length of the stored name is limited to 32 characters.

Table 1 Base block

Offset:	Size (bytes):	Contents:
0x0000	0x4	"regf" signature
0x000c	0x8	timestamp (Windows Filetime format)
0x0024	0x4	root key offset
0x0030	0x40	file name (Unicode)

The size of a bin header is always 0x20 bytes, and the bin header includes information about the size of a bin. The size of a bin can also be interpreted as an offset to the following bin. In other words, bins are chained together, in that each bin points to the bin that follows.

Table 2 Bin header

Offset:	Size (bytes):	Contents:
0x0000	0x4	"hbin" signature
0x0008	0x4	bin size

Key cells carry the signature "nk" and contain information about the size of the cell, name and timestamp of the key, and offsets to parent key, subkey list, value list, security descriptor and class name. Offsets are set to 0xffffffff if a key does not have a subkey list, a value list, a security descriptor or a class name. The key type is not necessary for tree

parsing, but key type values 0x2c and 0xac are reserved for root keys. The information about the number of subkeys is also included in subkey list cells.

Table 3 Key

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x2	"nk" signature
0x0006	0x2	type
0x0008	0x8	timestamp
0x0014	0x4	parent key offset
0x0018	0x4	number of subkeys
0x0020	0x4	subkey list offset
0x0028	0x4	number of values
0x002c	0x4	value list offset
0x0030	0x4	security descriptor offset
0x0034	0x4	class name offset
0x004c	0x2	key name length
0x004e	0x2	class name length
0x0050	variable	key name

Subkey lists contain information about the number of subkeys that they refer to. There are two main types of subkey lists: "lf"/"lh" lists and "ri"/"li" lists. Lists with signatures "lf" or "lh" contain offsets to subkeys and either first four characters of subkey name or a checksum of subkey name characters, while "ri" and "li" lists contain offsets to subkeys only.

Table 4 Subkey list "lf"/"lh

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x2	"lf" or "lh" signature
0x0006	0x2	number of subkeys
0x0008	0x4	offset to subkey
0x000c	0x4	four characters of subkey name or checksum
0x0010	0x4	offset to subkey
0x0014	0x4	four characters of subkey name or checksum
…	…	…

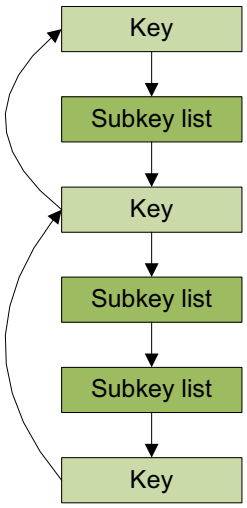
Table 5 Subkey list "ri"/"li"

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x2	"ri" or "li" signature
0x0006	0x2	number of subkeys
0x0008	0x4	offset to subkey or subkey list
0x000c	0x4	offset to subkey or subkey list
…	…	…

A subkey list does not necessarily point to subkeys but may also point to other subkey lists. There have been differences in interpretation of how subkey lists point to one another. Carvey (2007a), for example, has wrongly assumed that "ri" subkey list can only either point to "li" subkey lists or subkeys, whereas test have shown that "ri" subkey list can contain references to "lh" subkey lists as well. Norhdal-Hagen (2008) has corrected this in the latest versions of `ntreg.h` library. Since the purpose is to extract all active keys from the registry hive file, a parser needs to be flexible, and may as well allow any combination of pointers between subkey lists and subkeys.

A number of keys field in a parent key cell does not necessarily equal a number of keys field in a subkey list. If a subkey list points to other subkey lists, the number of subkeys field will contain a number of child subkey lists.

Figure 4 Keys and subkey lists



Each registry key can have multiple values and if this is the case, then the key cell contains an offset to a list of its values. A value list does not have any signature and contains only offsets to key values.

Table 6 Value list

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x4	offset to value
0x0008	0x4	offset to value
…	…	…

Values are stored in cells with signature "vk", and contain value name, type and either value data or an offset to a value data cell. Value name is simple to retrieve as the cell contains both the name length and the name. If a value cell does not contain a name, the named value field is set to 0x0 in which case the Windows registry editors translate the value name to "Default". In all other cases, the named value field is set to 0x1 and the value name is stored at offset 0x18.

Table 7 Value

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x2	"vk" signature
0x0006	0x2	value name length
0x0008	0x2	value data length
0x000b	0x1	data type
0x000c	0x4	value data or offset to value data
0x0010	0x4	value type
0x0014	0x2	named value
0x0018	value name length	value name

Value data is either stored internally in the value cell, if the value length does not exceed four bytes, or in a separate data cell, in which case the value cell contains an offset to a data cell. The data type field tells whether data is stored internally or externally: if the data type equals 0x80, the field at offset 0x0c contains value data, otherwise the field contains offset to a data cell where value data is stored.

Table 8 Value data

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	value data length	depends on value type

Value types play an important role, because data has to be interpreted depending on its type. Windows registry editors display string types as text, numbers as binary data, while all other types are displayed as both binary and hex dump. REG_SZ, REG_EXPAND_SZ and REG_MULTI_SZ are all null terminated strings. REG_EXPAND_SZ contains a reference to an environment variable, for example "DumpFile" = %SystemRoot%\MEMORY.DMP, where %SystemRoot% is replaced by an actual path at the run time. REG_MULTI_SZ value type contains several values separated by nulls. Value types of `SAM\Domain` keys can also have values in 500 and 1000 ranges, where account numbers are sometimes used as value types.

Table 9 Value type

Value	Type	Format
0	REG_NONE	undefined type
1	REG_SZ	fixed length string
2	REG_EXPAND_SZ	variable length string
3	REG_BINARY	binary data
4	REG_DWORD	32-bit number
5	REG_DWORD_BIG_ENDIAN	32-bit number (big-endian)
6	REG_LINK	Unicode string
7	REG_MULTI_SZ	multiple strings
8	REG_RESOURCE_LIST	binary data
9	REG_FULL_RESOURCE_DESCRIPTION	binary data
10	REG_RESOURCE_REQUIREMENTS_LIST	binary data
11	REG_QWORD	64-bit number

Class name is a hidden cell, in that it is not displayed by Windows registry editors. The parser will therefore extract this cell, so its contents can be examined.

Table 10 Class name

Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	class name length	class name (Unicode)

There are only a few security descriptors in the registry hives, because many keys, for efficiency purposes, are sharing the security descriptors whenever possible. Security descriptors are also the least documented cells in the registry. They contain security settings (permissions and audits) that control access to keys, and can be viewed using Windows registry editors. The parser will not include security data, as it seems less relevant for computer forensic examination. Just as any other cell, security descriptor cell contains information about its size, which will be used to calculate allocated space.

Table 11 Security descriptor

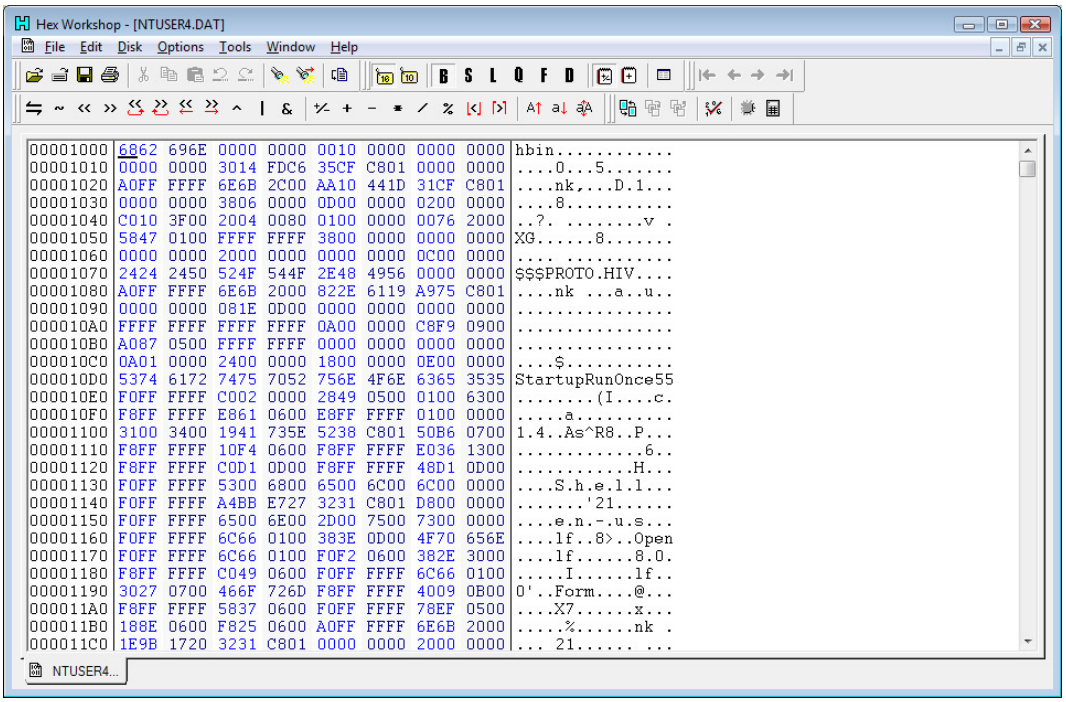
Offset:	Size (bytes):	Contents:
0x0000	0x4	size
0x0004	0x2	"sk" signature
…	…	…

3.2 Registry parser

The main purpose of the registry parser is to parse down the registry tree to access all allocated registry cells. The secondary goal is to build an offline registry viewer that displays registry data in a readable form, including key names, their timestamps values and class names.

As Figure 5 shows below, registry hive files can be displayed in a hex editor, and data stored as text can be easily read. However, manual extraction and translation of time stamps (Windows Filetime format represent the number of 100 nanosecond intervals since January 1, 1601 (MSDN, 2008a)), or manual parsing to correlate keys and values would be highly impractical.

Figure 5 Binary data input

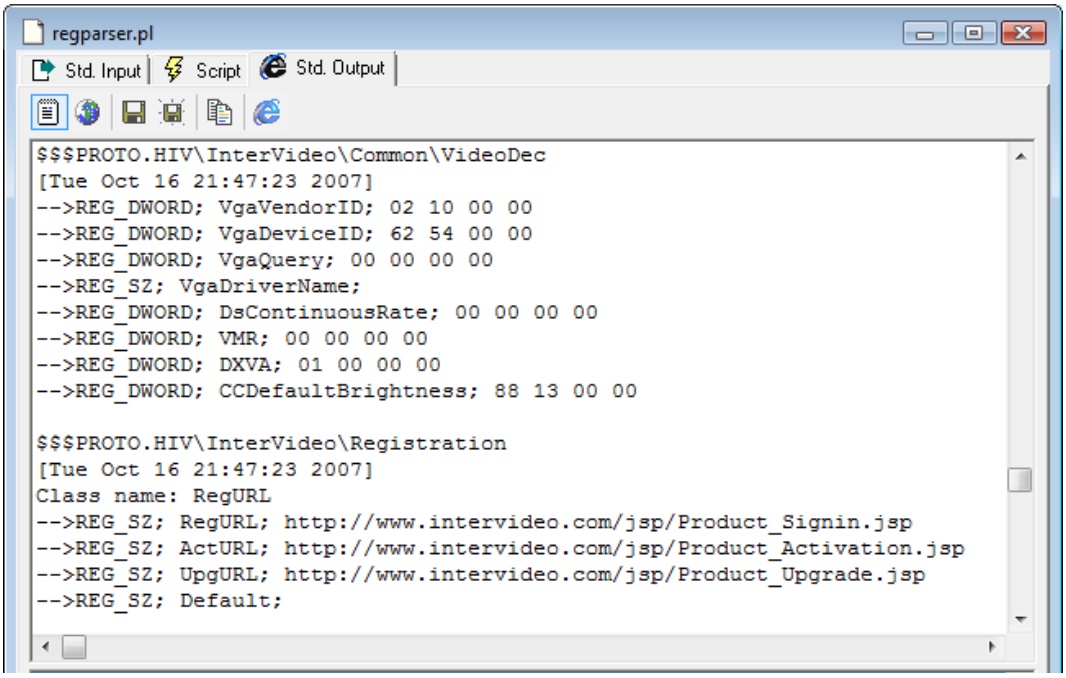


An off-line registry parser is a useful forensic tool for off-line examination of registry hive files. The tool will also be indispensable for evaluating whether the registry is parsed correctly, as output can be validated using Windows registry editors and other off-line registry viewers. Finally, the registry parser will be used for examination of valid data ranges of the different data fields.

Figure 6 shows the format of the output, which contains:

- key name, including its full path name
- key timestamp (in square brackets)
- class name (if any)
- list of key values: →value type; value; name; value data

Figure 6 Off-line registry parser



Registry parser is designed to recursively traverse the tree and to extract all relevant information from registry cells. The traversal is performed in a preorder manner to preserve the alphabetical order of keys. In a simplified version, the parser can be modularized as follows:

```
Registry_Parser
{
      Parse_Base_Block
      Parse_Registry_Tree (root_key_offset)
}

Parse_Base_Block
{
      extract file name and timestamp
      return root_key_offset
}

Parse_Registry_Tree (offset)
{
      if key_cell Parse_Key_Cell (offset)
      else if subkey_list_cell Parse_Sub_Key_List_Cell (offset)
}

Parse_Key_Cell (offset)
{
      extract key name and timestamp
      Parse_Class_Name (class_name_offset)
```

```
      Parse_Value_List_Cell (value_list_offset)
      Parse_Registry_Tree (subkey_list_offset)
 }
Parse_Sub_Key_List_Cell (offset)
{
      for each subkey { Parse_Registry_Tree (subkey_offset) }
}

Parse_Value_List_Cell (offset)
{
        for each value { Parse_Key_Value_Cell (value_offset) }
}

Parse_Key_Value_Cell (offset)
{
      extract value name and type
      if value_data_type == 0x80 extract value data
      else Parse_Value_Data_Cell (value_offset)
}

Parse_Value_Data_Cell (offset)
{
            extract and translate value data
}
```

Extra functionality includes passing of the full key path during the traversal. Since the intention is to output full path names, the full path of the key is passed between modules that parse keys and subkey lists.

The purpose of the off-line registry parser is to display relevant data in a human readable form. The binary data extracted from the registry can have many formats and therefore need translation. Data stored in ASCII format can be output direct, while data stored in Unicode format needs translation. In a few special cases, value names are encrypted with the ROT-13 algorithm and need to be decrypted as well. Value data stored as null terminated strings can be displayed as text once the terminating nulls are removed. Values of type REG_MULTI_SZ contain multiple strings separated by null characters, which need to be separated, before being displayed separated by semicolons. All 32-bit and 64-bit numbers can be displayed as either numbers or binary data, or both. All other data types (binary and REG_NONE) will be displayed as both binary data and hex dump. The main reason for that is that binary type values can contain text - either ASCII or Unicode, and should therefore be displayed in a more readable hex format.

Many different applications write data to the registry, and data is sometimes stored in incorrect formats. Sometimes values of type REG_SZ contain arbitrary binary data, including control characters, such as new lines, tabs, etc. To assure proper formatting of the output, all control characters in the range of 0x00-0x19 will be removed from all text output.

3.3 Allocated space

Assuming that registry parser reaches all existing registry cells, unallocated space can be computed by subtracting the allocated space from the hive file. Allocated_space [] is a hash where offsets and sizes of all allocated cells are saved. The size of a base block is 0x1000 bytes and size of each bin header is 0x20 bytes. All cells have their size stored in the first four bytes and the sizes are negative, therefore the actual cell size is an absolute value (or negated value) of the signed long number.

```
Registry_Parser
{
      Parse_Base_Block
      Parse_Bin_Headers
      Parse_Registry_Tree (root_key_offset)
}

Parse_Base_Block
{
      Allocated_space [offset] = 0x1000
      return root_key_offset
}

Parse_Bin_Headers
{
      my blocks = file_size / 0x1000
      for each block
      {
            if bin header { Allocated_space [block_offset] = 0x20
}
      }
}

Parse_Registry_Tree (offset)
{
      if key_cell { Parse_Key_Cell (offset) }
      else if subkey_list { Parse_Sub_Key_List_Cell (offset) }
}
```

```
Parse_Key_Cell (offset)
{
      Allocated_space [offset] = key_cell_size
      Allocated_space [class_name_offset] = class_name_cell_size
      Allocated_space [security_descriptor_offset]
      = security_descriptor_cell_size
      Parse_Value_List_Cell (value_list_offset)
      Parse_Registry_Tree (subkey_list_offset)
 }

Parse_Sub_Key_List_Cell (offset)
{
      Allocated_space [offset] = subkey_list cell_size
      for each subkey { Parse_Registry_Tree (subkey offset) }
}

Parse_Value_List_Cell (offset)
{
      Allocated_space [offset] = value_list_cell_size
      for each value { Parse_Key_Value_Cell (value_offset) }
}

Parse_Key_Value_Cell (offset)
{
      Allocated_space [offset] = value_cell_size
      if value_data_type <> 0x80
            { Allocated_space [value_data_cell_offset]
            = value_data_cell_size) }
}
```

3.4 Unallocated space

The algorithm that calculates the unallocated space needs to first calculate the allocated space as shown above and then subtract the allocated space from the file.

A hash `Unallocated_space []` stores offsets and sizes of all unallocated cells, where

`Unallocated_space [cell_offset] = cell_size.`

```
previous_offset = 0x0
for each offset in sorted Allocated_space []
{
      current_offset = offset + Allocated_space [offset]
      if previous_offset < offset
            { Unallocated_space [previous_offset]
            = offset - previous_offset }
      previous_offset = current_offset
}
```

To include the space between the last allocated cell and the end of a file, an extra value needs to be added to the Allocated_space hash, before unallocated space is computed:

```
Allocated_space [file_size] = 0x0
```

3.5 Simpler way to compute unallocated space

Examination of unallocated space, computed as described above, has shown that the first four bytes of each unallocated cell interpreted as a signed long number, equal to what has been computed as the cell's size. This number is positive as opposed to cell sizes of allocated cells, which are always negative. In fact, a PowerPoint presentation by Probert (n.d.) includes the following information about a cell size:

> *"positive = free cell, negative = allocated cell (actual size is –Size)".*

The statement seems to implicate that, upon deletion, cell sizes are negated and therefore previously deleted records could be found by searching for valid signatures preceded by positive cell sizes, but tests have shown that it was not the case. Russinovich (1999) explains that neighboring deleted cells are joined together and as a result, unallocated cells can contain multiple keys and values, as opposed to allocated cells that only contain a single key, value, list, etc. In conclusion, while it is not possible to search for deleted records direct, it is possible to calculate unallocated space solely based on cell sizes, and without parsing down the tree. The following simple algorithm computes unallocated space without parsing the registry tree:

```
do {
      if base_block { cell_size = -0x1000 }
      else if bin_header { cell_size = -0x20 }
      else { cell_size = signed long }
      if  cell_size > 0x0
            { Unallocated_space [offset] = cell_size }
      last_cell_offset = offset
      offset = offset + abs (cell_size)
} until cell_size = 0x0 or offset > file_size

if last_cell_offset < file size
      { Unallocated_space [last_cell_offset] =
      file_size - last_cell_offset }
```

All cells can be classified as allocated or unallocated solely based on their size (positive or negative). A base block and bin headers do not carry information about their sizes, but their sizes are constant. The algorithm sets base block and bin header sizes to be negative numbers in order to exclude them from the unallocated space.

Tests have shown that hive files often have some additional space after the last bin and that this space often contains zeros, but in some case the extra space contains random data, therefore the algorithm includes that space as unallocated.

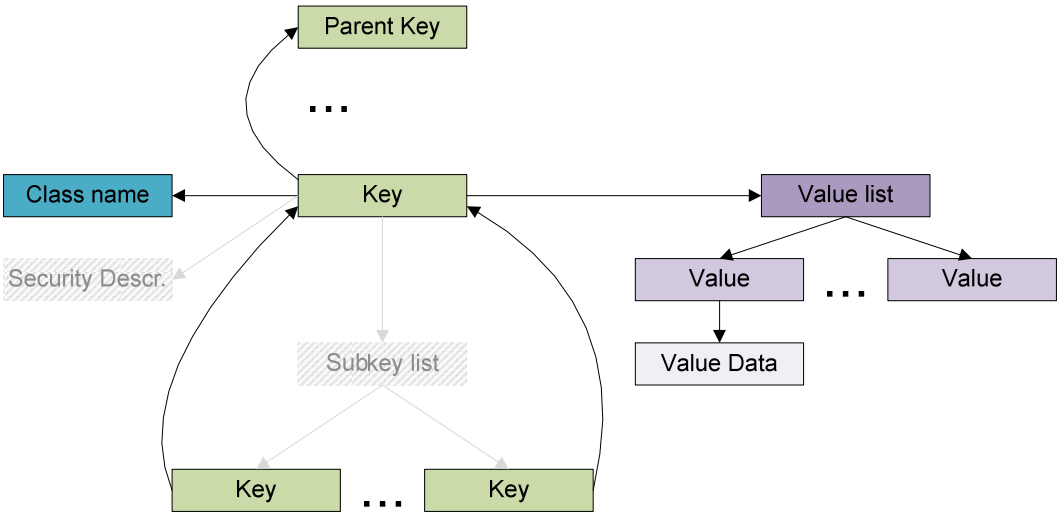
3.6 Recovery of deleted data

Once the unallocated registry space is computed, registry keys and values can be found by searching for their signatures. In recovery of deleted data, the project will focus on recovery of keys and their values. The project sponsor has expressed that restoring values without any connection to a key, could be compared to finding a text string in a slack space. That being said, it could still be relevant to look for text strings in the registry "slack", which can be achieved by displaying all unallocated space as a hex dump for visual examination. The focus of the project will be to recover keys and their values, while everything else will be included in a hex dump.

3.6.1 Recovery of keys and their values

There are limits to what can be extracted from unallocated cells. Registry keys link back to their parent keys. Whenever a key is restored, it is possible reconstruct the full key path, assuming that ancestor keys are either not deleted, or deleted but recoverable. Subkey links are removed on deletion, but if paths can be restored, they will show eventual parent child relations between restored keys. Furthermore, links between keys and their security descriptor are removed as well. Interestingly class name links are not overwritten, and class name can therefore be extracted if it exists, but only few keys have class names.

Figure 7 Deleted key



Deleted keys still carry valuable information about their timestamps and values. If a key can be recovered, there is a good possibility that at least some of its values can be recovered as well, if their value lists are intact.

Examination of the unallocated space has shown that in the beginning of a hive file unallocated cells tend to be small, while larger unallocated cells are most often found towards the end of a hive file. This can be explained by the functionality of Configuration

Manager, as described by Russinovich (1999), where new cells are stored in a first found unallocated cell of a sufficient size. Since key cells are always larger than 0x50 bytes (key name starts at offset 0x50), while searching for deleted keys the algorithm can be optimized by skipping all unallocated cells that are too small to hold a key cell. For the same reason, the algorithm can stop if the remaining space of unallocated cell to be still examined becomes too small. Finally, since the size of a cell is always a multiple of 0x8 bytes, the algorithm can skip 0x8 bytes of input at a time while looking for a cell signature. Once a signature "nk" is found, the algorithm proceeds in a similar manner to the registry parser, with the exception that key path is computed by following parent links. The final part of the algorithm displays all unallocated space in both binary and hex formats.

```
for each offset in Unallocated_space[]
{
      if Unallocated_space [offset] > 0x50
      {
            max_ offset =
            offset + Unallocated_space [offset] - 0x50
            while (offset < max_ offset)
            {
                  if key_cell_signature { Parse_Key }
                  offset = offset + 0x8
            }
      }
}

for each offset in Unallocated_space[]
{
      display cell in binary and hex format
}

Parse_Key
{
      extract key name
      extract time stamp
      key_path = key_name
      while key_cell at parent_key_offset
      {
            key_path = parent_key_name . "\" .  key_path;
            offset = parent_key_offset
      }
      Parse_Value_List (value_list_offset)
}

Parse_Value_List (offset)
{
      for each value { Parse_Key_Value (value offset) }
}

Parse_Key_Value (offset)
{
      extract value name and type
```

```
      if value_data_type == 0x80 { extract value data }
      else { Parse_Value_Data (value offset) }
}

Parse_Value_Data (offset)
{
            extract and translate value data
}
```

Functionality not shown above includes determination whether a full key path was successfully computed; if the algorithm reached the root of the tree following parent links then the key type of the key is either 0x2c or 0xac. If that is not the case, question marks will precede the key path in order to show that the full key path could not be recovered. Just like in the parser algorithm, data needs to be formatted and translated depending on its type, for example timestamps, Unicode, ROT-13, etc.

3.6.2 Data validation

Deleted keys and values no longer reside in separate cells, because Windows kernel adjoins neighboring deleted cells, and the information about the size of the previously deleted keys and value cells is no longer available. This complicates matters because it is not possible to determine where data belonging to a deleted unit stops and whether or not it has been overwritten. Very careful validation of input is therefore necessary in order to avoid incorrect outputs.

In order to reject corrupted data, retrieved key cells can be validated by checking if each field of data has a vaid value. As already mentioned, offsets to a security descriptor and a sub key list are set to 0xffffffff, while number of sub keys is set to 0x0 when a key is deleted. If a key contains class name, it has to contain a valid offset to a class name cell, and if there is no class name (class name length equals 0x0), a class name offset should be set to 0xffffffff. Similar relation occurs between a number of values and a value list offset.

It can be difficult to decide valid ranges of data. Very relaxed validation might result in many false positives, while too strict validation might result in false negatives, where valid data is rejected. A valid timestamp can be defined as being in the range of 0x0 - hive file timestamp, because a file timestamp represents the last time the file was written to, therefore a key timestamp can never be larger than the file timestamp. The timestamp range could be further improved by selecting the earliest possible timestamp - for example the time when the Windows operating system was installed. Valid offsets can be defined to be in range of 0x1000 - file size, to exclude the base block, but since all offsets are relative to the first bin, as described previously, the actual valid range is 0x0 … file size - 0x1000. Furthermore, Microsoft defines maximum key name length to be 255 (0xff) characters, while maximum value name length is 16,383 (0x3ff). (MSDN, 2008b)

Table 12 Deleted key with values

Offset:	Size (bytes):	Contents:	Valid data:
0x0004	0x2	signature	nk
0x0008	0x8	timestamp	0x0 … file timestamp
0x0014	0x4	parent offset	0… file_size - 0x1000
0x0018	0x4	number of sub keys	0x0
0x0020	0x4	sub key list offset	0xffffffff
0x0028	0x4	number of values	0x1 … 0xffffffff
0x002a	0x4	value list offset	0… file_size - 0x1000
0x0030	0x4	security descriptor offset	0xffffffff
0x0034	0x4	class name offset	valid offset (0xffffffff if none)
0x004c	0x2	key name length	0x1 … 0xff
0x004e	0x2	class name length	0x0 if no class name

Some additional fields, which were not used for parsing of the registry tree, can be used for validation of input. As shown in Table 13, keys without any values (a number of values field is set to 0x0), have their maximum value name length and maximum value data size fields set to 0x0.

Table 13 Deleted key without values

Offset:	Size (bytes):	Contents:	Valid data:
0x0004	0x2	signature	nk
0x0008	0x4	timestamp	0x0 … file timestamp
0x0014	0x4	parent offset	0 … file_size - 0x1000
0x0018	0x4	number of sub keys	0x0
0x0020	0x4	sub key list offset	0xffffffff
0x0028	0x4	number of values	0x0

0x002c	0x4	value list offset	0xffffffff
0x0030	0x4	security descriptor offset	0xffffffff
0x0034	0x4	class name offset	0xffffffff
0x0040	0x4	max value name length	0x0
0x0044	0x4	max value data size	valid offset (0xffffffff if none)
0x004c	0x2	key name length	0x1 … 0xff
0x004e	0x2	class name length	0x0 if no class name

Similarly, recovered value cells can be validated in order to reject corrupted data. Data type can have a value of 0x0 or 0x80, in the former case value data is stored separately and the value data field should contain a valid offset. If data is stored in a value cell, the maximum length of value data is 0x4 bytes. In both cases, value types can be in the range 0x0 … 0xb, with the exception of values in SAM\Domains keys, where value types can be in 500 and 1000 ranges, because account numbers are sometimes used as value types. Named value field is set to 0x0 if value has no name (which Windows translates to "Default"), in which case value name length should equal 0x0. If a value has a name, named value field should be set to 0x0.

Table 14 Deleted value linking to value data

Offset:	Size (bytes):	Contents:	Valid data:
0x0004	0x2	signature	vk
0x0006	0x2	value name length	0x0 … 0x3ff
0x000b	0x1	data type	0x0
0x000c	0x4	value data offset	0… file_size - 0x1000
0x0014	0x2	named value	0 … 1

Table 15 Deleted value containing value data

Offset:	Size (bytes):	Contents:	Valid data:
0x0004	0x2	signature	vk
0x0006	0x2	value name length	0x0 … 0x3ff
0x0008	0x2	value data length	0x0 … 0x4
0x000b	0x1	data type	0x80
0x000c	0x4	value data	0 … 0xffffffff
0x0014	0x2	named value	0 … 1

In cases where value data is stored outside the value cell, there is no trivial way to decide if the value data has been overwritten, as the cell does not contain any extra fields that can be validated.

The algorithm described in chapter *3.6.1 Recovery of keys and their values* can thus be improved by adding the following validation statements:

```
Parse_Key
{
      return if timestamp > file_timestamp
      return in not Valid_offset (parent_offset)
      return if number_of_sub_keys <> 0x0
      return if subkey_list_offset <> 0xffffffff
      return if security_descriptor_offset <> 0xffffffff
      return if class_name_offset == 0xffffffff
            and class_name_length <> 0x0
      return if class_name_offset <> 0xffffffff
            and class_name_length == 0x0
      return if key_name_length == 0x0 or key_name_length > 0xff
      return if class_name_length <> 0x0
      if number_of_values == 0x0
      {
            return if value_list_offset <> 0xffffffff
            return if max_value_name_length <> 0x0
            return if max_value_data_size <> 0x0
      }
      else
      {
            return if not Valid_offset (value_list_offset)
            return if max_value_name_length > 0x3ff
      }
      Parse_Value_List (value_list_offset)
}

Parse_Value_List (offset)
{
      for each value
      {
            if Valid_offset (value_offset)
                  { Parse_Key_Value (value offset) }
      }
}

Parse_Key_Value (offset)
{
      return if signature <> "vk"
      return if value_name_length > 0x3ff
      return if data_type <> 0x0 and data_type <> 0x80
      if data_type == 0x80
      {
            return if value_data_length > 0x4
      }
      else if data_type == 0x0
      {
            return if not Valid_offset (value_data)
      }
      return if named_data == 0x0 and value_name_length > 0
      return if named_data == 0x1 and value_name_length == 0
}

Valid_offset (offset)
{
```

```
        return offset < file_size - 0x1000
}
```

3.6.3 Improved algorithm

The tests run by the project sponsor have shown that the algorithm retrieved keys that are also present in the "live" registry. Restored keys had the same names and values, and sometimes-even timestamps, as "live" keys but were stored at different offsets in registry hive file. The study of how Windows registry functions perform key updates and deletions is out of the scope of the project. However, it is not optimal to list recovered keys without any indication of whether or not they still exist in the "live" registry. The algorithm will be therefore improved to differentiate between deleted and updated keys, in order to provide more complete information. The algorithm has to perform the following functions: compute unallocated space, recover keys and their values from the unallocated space, and finally to examine if recovered keys reside in the allocated space.

In order to provide as much documentation as possible for recovered data, the algorithm will display offsets to recovered key and values. Since one cannot guarantee that all, if any, values can be retrieved, the algorithm will display the number of values field, to document how many values a key had upon deletion. Finally, each recovered key will be looked up to determine if the key has been deleted or updated (still resides in live registry). If the key still resides in the registry, the live key and its values will be parsed as well to show what, if anything, has changed.

The desired output of the algorithm is as follows:

Deleted key

- key offset
- key name, including its full path name
- key timestamp (in square brackets)
- number of values

- list of key values: →offset; value type; value; name; value data

  …

Updated key

- key offset
- key name, including its full path name
- key timestamp (in square brackets)
- number of values
- list of key values: →value type; value; name; value data

Corresponding live key

- key offset
- key timestamp (in square brackets)
- number of values
- list of key values: →offset; value type; value; name; value data

  …

Unallocated space

- offset range
- binary data
- hex dump

  …

There are two different methods for calculation of unallocated space: by either parsing of the registry tree or based on cell sizes. There are also two different ways of deciding whether a restored key has a duplicate in the "live" registry. A key can be looked up in a registry when it is recovered, or a hash of key names and their offsets can be created beforehand and used for a quick look up of restored keys.

In a recent paper, Morgan (2008) has described how data hiding in a registry can easily be accomplished by storing data in an unallocated cell and changing the size of the cell to a negative number, thus marking it as used. The manipulated cell would not be

referenced by the registry tree and would not be discovered if unallocated space were computed based on cell sizes (looking for cells with a positive number in the size field). Based on that valid observation, the final algorithm will therefore compute allocated space by parsing the registry tree, and consequently any data hidden by manipulation of a cell size will be recovered.

Since the algorithm already computes allocated space by parsing the registry tree, a hash Live_key [] can be created, and store both key paths and offset, for efficient look up of recovered keys later:

```
Parse_Key_Cell (offset)

{
      …
      Live_key [key_path] = offset
      …
}
```

The final version of the algorithm combines previously described algorithms to generate output as described above, and can be summarized as follows:

```
Parse registry tree
      Compute allocated space
      For each key { Live_key [key_path] = offset }
Unallocated space = file space – allocated space
For each unallocated cell
      Search for "nk" signature
      Validate all fields
      Print key data
      For each value
            Validate all fields
            Print value data
      If Live_key[key]
            Print live key data
            For each live key value
                  Print value data
Display unallocated space as binary and hex dump
```

Chapter 4. METHODS AND REALIZATION

4.1 Methodology

There exist many different methodologies for approaching analysis and design in software projects in the IT industry. Most of these based on an assumption that the structure of input data and functional requirements can be discovered during an analysis phase and the software design can then be derived from the analysis. The nature of this project is more exploratory than most software projects, and it takes an experimental approach with continual refinement of successive prototypes.

As mentioned earlier, the existing documentation of the Windows registry is incomplete. This meant that the analysis had to be based not only on existing documentation but also has to include knowledge gathered by thorough examination of Windows hive files. Taking a start point in the existing information about the Windows registry hives a first prototype of the registry parser was constructed. This prototype was used to analyze test hive files from Windows XP and Windows Vista machines. The results were used to refine the analysis and design and to improve the prototype. The final analysis and design therefore represent the combined knowledge obtained in these iterations.

4.2 Tools

Microsoft's registry API functions (MSDN, 2008c) allow applications to retrieve and modify or delete data. Those API functions have severe limitations in that their detailed functionality and their implementation are not fully documented and they therefore operate as black box functions. Furthermore, no functions are available to peek into the

slack space of the registry. Finally, using Windows based API functions would limit the tools to execution on a single platform.

Perl has been chosen as a programming language, because it is very flexible and platform independent, and to accommodate the sponsor of the project, who implements all his forensic tools in Perl. The functions that access input hive files have been placed in a separate library, in order to keep scripts simple and to facilitate reuse of code.

`Hex Workshop` (Break Point Software, 2008) hex editor has been used to manually examine binary files. Hive files have been collected from several machines by booting them from a Linux disc, as hive files are locked while the Windows operating system is running. On one occasion, `ERUNT` (Hederer, 2005) tool was used to obtain files without shutting down the system. Perl scripts were interpreted using `ActivePerl 5.10.0` (ActiveState, 2008) distribution, and screen shots of tests show outputs displayed in a `Perl Express` (2005) editor.

4.3 Realization

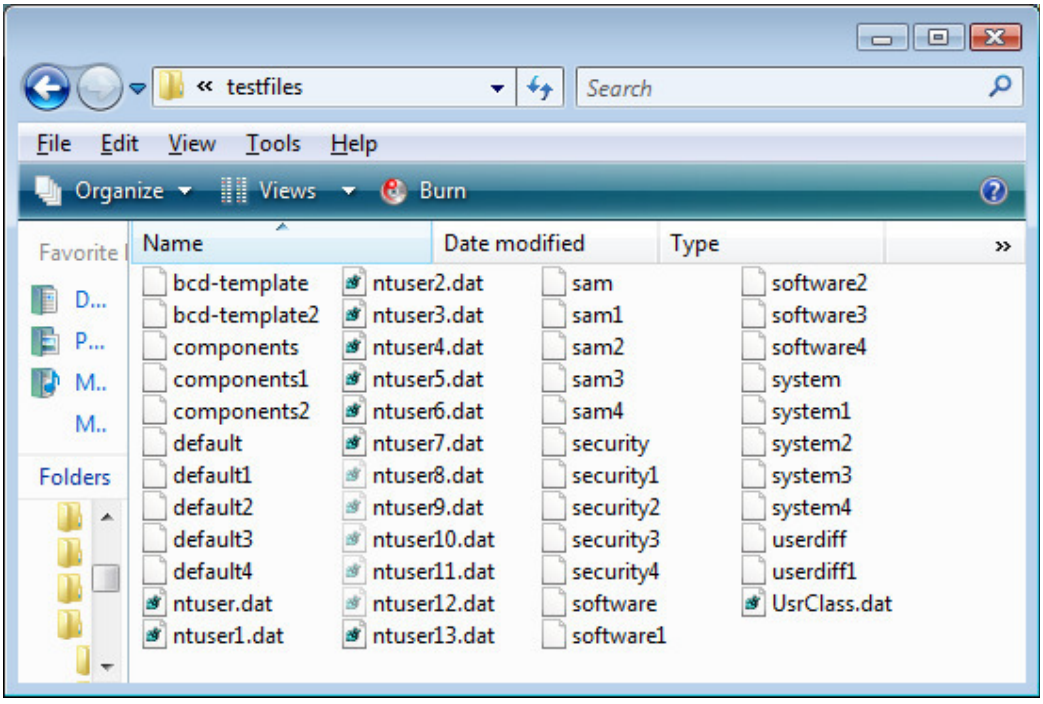
The design has been changed numerous times. Once the algorithm that parses down the registry tree was implemented, the original assumptions about the structure of registry cells have been confirmed, rejected or modified by outputting one field at a time for each registry cell. The discovery that unallocated cells have positive sizes resulted in the redesign of the methodology to compute unallocated space in a more efficient manner. There have been no assumptions about recovered data until that data was actually found and examined, because at the beginning of the project it was unknown whether anything useful will be found in the unallocated space at all. The surprising fact that recovered keys can have duplicates in the "live" registry led to yet another change in the design, as it was now necessary to verify whether keys were deleted or only updated. Once data was recovered, it became clear that data was frequently overwritten. To avoid recovery of

corrupted data, key and value cells had to be reexamined in order to find valid ranges for as many fields as possible, thus including fields that have not previously been of much interest. Valid ranges of data were defined by running tests on "live" data in order to identify upper and lower bounds. Additionally outputs from both "live" and recovered keys have been compared, in order to determine how key cells are modified upon deletion. The final change in the design occurred when it became clear that computing of unallocated space based on cell sizes has a weakness, in that an adversary can easily manipulate cell sizes in order to hide data.

4.4 Tests

Test data consists of 47 hive files, collected from Windows Vista and Windows XP machines, supplied by or copied with permissions from their owners. Hive files of types `COMPONENTS`, `SYSTEM` and `SOFTWARE` were the most difficult to obtain due to their huge sizes.

Figure 8 Test hive files



Tests were performed throughout the whole life cycle of the project, but tests were typically run on smaller hive files for efficiency reasons. The final tests were run on the complete set of test files. The testing has been very successful, as it not only helped to find small errors in the program code, but also led to new discoveries about Windows registry hive files. The main limitation of the testing phase is the small set of hive files, which were difficult to obtain.

The final tests cover the assumptions made about the registry hive files, parsing of the registry tree, computation of unallocated space, and finally recovery of keys and their values from unallocated space, including validation and interpretation of the recovered data.

4.4.1 Hive file structure

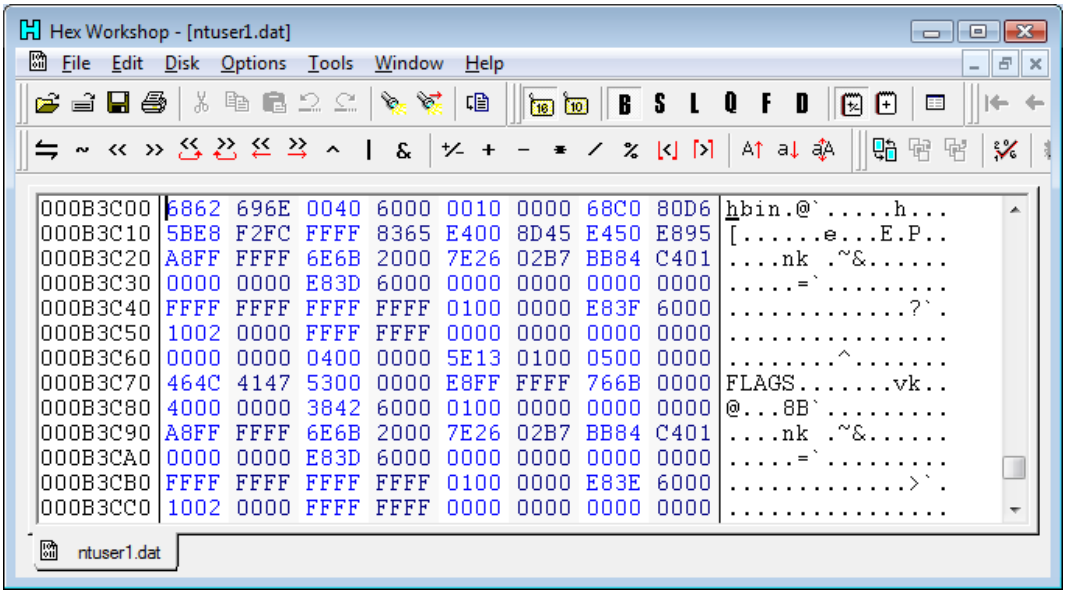
To test the general assumptions about registry hive file's building blocks, such as base block, bins and cells, a script `hivestructure.pl` (source code is listed in appendix A.4) was developed. The script checks the size of each hive file and parses all files sequentially in order to determine sizes of base blocks, bins, bin headers and cells. Test numbers are included in the script to indicate which code segment performs the given test.

Table 16 Hive file structure

Element	Assumption	Test number
file size	multiple of 0x1000 bytes	1
base block	0x1000 bytes	2
bin	multiple of 0x1000 bytes	3 (failed)
bin header	0x20 bytes	4
cell	multiple of 0x8 bytes	5

As Figure 9 shows, the assumption about a bin size being always a multiple of 0x1000 bytes was incorrect, as in a single file a bin header existed at an offset divisible by 0x400 bytes. Once this number was corrected, all test files passed the test successfully.

Figure 9 Bin header at an offset not divisible by 0x1000



4.4.2 Data fields

A script `datafields.pl` (source code is listed in appendix A.5) was implemented in early stages of the project, to determine valid ranges of data fields, and most false assumptions have been corrected at the early stages of the project. The final test was run on the complete set of test files, and a single new error has been discovered; in few cases, bins did not correctly link to each other, and broken chains were identified. This caused a correction of the algorithm to compute allocated space described in chapter *3.3 Allocated space*. Instead of searching for bins following bin header offsets to following bins, the algorithm now searches for bin headers looking for their signatures at relevant offsets. All other tests have completed successfully.

Table 17 Base block validation

Field	Assumption	Test number
signature	"regf"	1
timestamp	>= all key timestamps	6
root key offset	points to a root key	2

Table 18 Bin header validation

Field	Assumption	Test number
signature	"hbin"	3
next bin offset	points to next bin	failed

Table 19 Key validation

Field	Assumption	Test number
cell size	negative	4
signature	"nk"	5
timestamp	<= file timestamp	6
key type	0x2c and 0xac reserved for root keys	7
parent offset	points to a key cell (except for root key)	8
subkey list offset	0xffffff if number of subkeys == 0x0	9
subkey list offset	points to subkey list if number of subkeys > 0x0	10
value list offset	0xffffff if number of values = 0x0	11
value list offset	valid offset if number of values > 0x0	12
security descriptor offset	0xffffffff or points to security descriptor cell	14
class name offset	0xffffffff or valid offset	15
key name length	> 0x0	17

Table 20 Security descriptor validation

Field	Assumption	Test number
cell size	negative	13
signature	"sk"	14

Table 21 Class name

Field	Assumption	Test number
cell size	negative	16

Table 22 Subkey list validation

Field	Assumption	Test number
cell size	negative	18
signature	"lf", "lh", "ri" or "li"	19
number of subkeys	> 0x0	20
offsets	point to subkey or subkey list	21

Table 23 Value list validation

Field	Assumption	Test number
cell size	negative	22
offsets	point to values	23

Table 24 Value validation

Field	Assumption	Test number
cell size	negative	24
signature	"vk"	25
value name length	<= 0x3ff	26
data type	0x0 or 0x80	27
value data length	<= 0x4 if data type 0x80, > 0x4 otherwise	28
named value	0x0 if name length == 0x0, 0x1 otherwise	29
value data	valid offset if data type == 0x0	30

Table 25 Linked data validation

Field	Assumption	Test number
cell size	negative	31

4.4.3 Registry parser

The source code of the registry parser script called `regparser.pl` is listed in appendix A.1, and sample outputs from the script (one screen shot per each input hive file) are included in appendix B.1.

The functional test of the registry parser was performed by constructing a set of cases that cover different input types. The script `regparser.pl` was slightly modified in order to enumerate all keys and values for reference purposes. The script also outputs key and sub key lists encountered in each tree path of a key, to identify cases where a given sub key list type was used. Keys and values that satisfy the specified test cases were retrieved from the output (see appendix B.2) and examined; if a key covers a test case all of its values are also shown, and if a value covers a test case, its key and all other values belonging to that key are shown as well. As Table 26 illustrates, the input test set covers the specified test cases, with the exception of a single value type, which was not present in any of the test input files.

Table 26 Functional test of regparser.pl

Cell	Field	Test file	Key/Value
key	number of sub keys == 0	ntuser.dat	key 4
key	number of sub keys > 0	ntuser.dat	key 3
key	class name offset == 0xffffffff	ntuser.dat	key 3
key	class name offset != 0xffffffff	ntuser.dat	key 312
values	number of values == 0	ntuser.dat	key 3
values	number of values > 0	ntuser.dat	key 4
sub key list	signature "lf"	ntuser.dat	key 3
sub key list	signature "lh"	system	key 4181
sub key list	signature "ri"	components	key 510
sub key list	signature "li"	components	key 510
value	name length == 0	ntuser.dat	value 1
value	name length > 0	ntuser.dat	value 2
value	name encrypted with ROT-13	ntuser1.dat	value 3880
value	type REG_NONE	software	value 128458
value	type REG_SZ	ntuser.dat	value 1
value	type REG_EXPAND_SZ	software	value 128462
value	type REG_BINARY	components	value 1746
value	type REG_DWORD	software	value 5
value	type REG_DWORD_BIG_ENDIAN	sam1	value 31
value	type REG_LINK	sam1	value 5
value	type REG_MULTI_SZ	software	value 128698
value	type REG_RESOURCE_LIST	system	value 11436
value	type REG_FULL_RESOURCE_DESCRIPTION	-	N/A
value	type REG_RESOURCE_REQUIREMENTS_LIST	system	value 11435
value	type REG_QWORD	software	value 105162
value	type other	sam	value 12
value	data length == 0	sam	value 12
value	data length <= 4	software	value 5
value	data length > 4	ntuser.dat	value 1

Additionally all the keys and their values used for the purpose of the above test were also retrieved from the output generated by the `regp.pl` offline registry viewer tool by Carvey (2007a), its unaltered source code is listed in appendix A.6. Key names, timestamps, value names, value types and value data output by both scripts (see appendix B.2) were compared and a few differences were found and are explained below. The test has confirmed that the script `regparser.pl` extracts and translates key and values correctly.

Table 27 Comparison of outputs from regparser.pl and regp.pl

Test file	Key/Value	Element	Comparison
components	key 510	name	√
components	key 510	timestamp	√
components	key 510	number of values	√
components	value 1746	value type	√
components	value 1746	value name	√
components	value 1746	value data	√
components	value 1747	value type	√
components	value 1747	value name	√
components	value 1747	value data	√
components	value 1748	value type	√
components	value 1748	value name	√
components	value 1748	value data	√
ntuser.dat	key 3	name	√
ntuser.dat	key 3	timestamp	√
ntuser.dat	key 3	number of values	√
ntuser.dat	key 4	name	√
ntuser.dat	key 4	timestamp	√
ntuser.dat	key 4	number of values	√
ntuser.dat	value 1	value type	√
ntuser.dat	value 1	value name	√
ntuser.dat	value 1	value data	√
ntuser.dat	value 2	value type	√
ntuser.dat	value 2	value name	√
ntuser.dat	value 2	value data	√
ntuser.dat	key 5	name	√
ntuser.dat	key 5	timestamp	√
ntuser.dat	key 5	number of values	√
ntuser.dat	value 3	value type	√
ntuser.dat	value 3	value name	√
ntuser.dat	value 3	value data	√
ntuser.dat	key 312	name	√
ntuser.dat	key 312	timestamp	√
ntuser.dat	key 312	number of values	√
ntuser1.dat	key 900	name	√
ntuser1.dat	key 900	timestamp	√
ntuser1.dat	key 900	number of values	√
ntuser1.dat	value 3880	value type	√
ntuser1.dat	value 3880	value name	√
ntuser1.dat	value 3880	value data	√
ntuser1.dat	value 3881	value type	√
ntuser1.dat	value 3881	value name	√
ntuser1.dat	value 3881	value data	√
ntuser1.dat	value 3882	value type	√
ntuser1.dat	value 3882	value name	√
ntuser1.dat	value 3882	value data	√
ntuser1.dat	value 3883	value type	√
ntuser1.dat	value 3883	value name	√
ntuser1.dat	value 3883	value data	√
ntuser1.dat	value 3884	value type	√
ntuser1.dat	value 3884	value name	√
ntuser1.dat	value 3884	value data	√
sam	key 12	name	√
sam	key 12	timestamp	√
sam	key 12	number of values	√
sam	value 12	value type	see [1]

sam	value 12	value name	√
sam	value 12	value data	see [2]
sam1	key 5	name	√
sam1	key 5	timestamp	√
sam1	key 5	number of values	√
sam1	value 5	value type	√
sam1	value 5	value name	√
sam1	value 5	value data	see [2]
sam1	key 31	name	√
sam1	key 31	timestamp	√
sam1	key 31	number of values	√
sam1	value 31	value type	√
sam1	value 31	value name	√
sam1	value 31	value data	see [2]
software	key 7	name	√
software	key 7	timestamp	√
software	key 7	number of values	√
software	value 5	value type	√
software	value 5	value name	√
software	value 5	value data	√
software	key 86093	name	√
software	key 86093	timestamp	√
software	key 86093	number of values	√
software	value 105162	value type	see [3]
software	value 105162	value name	√
software	value 105162	value data	see [3]
software	key 93378	name	√
software	key 93378	timestamp	√
software	key 93378	number of values	√
software	value 128458	value type	√
software	value 128458	value name	√
software	value 128458	value data	see [4]
software	key 93380	name	√
software	key 93380	timestamp	√
software	key 93380	number of values	√
software	value 128460	value type	√
software	value 128460	value name	√
software	value 128460	value data	√
software	value 128461	value type	√
software	value 128461	value name	√
software	value 128461	value data	√
software	value 128462	value type	√
software	value 128462	value name	√
software	value 128462	value data	√
software	value 128463	value type	√
software	value 128463	value name	√
software	value 128463	value data	√
software	key 93435	name	√
software	key 93435	timestamp	√
software	key 93435	number of values	√
software	value 128697	value type	√
software	value 128697	value name	√
software	value 128697	value data	√
software	value 128698	value type	√
software	value 128698	value name	√
software	value 128698	value data	see [5]
system	key 4181	name	√
system	key 4181	timestamp	√
system	key 4181	number of values	√
system	value 11435	value type	√

system	value 11435	value name	√
system	value 11435	value data	√
system	value 11436	value type	√
system	value 11436	value name	√
system	value 11436	value data	√

[1] `regp.pl` did not display any value type as it only allows value types in ranges 0x9…0xa. The actual value type is 1007, as sometimes account numbers are used as value types in `SAM` hive files.

[2] For some value types `regp.pl` displays value data as 0, even though the value has no data. Tests were rerun to show the length of value data and confirmed that those values do not have any data (data length is set to 0x0) and that therefore `regparser.pl` handles the data field correctly.

[3] `regp.pl` does not implement the value type REG_QWORD and as result displays data as text

[4] In the case of value type REG_NONE regp.pl translated the value data incorrectly

[5] There is a difference in how the two scripts format values of the type REG_MULTI_SZ; `regparser.pl` inserts semicolons to separate the retrieved strings.

Finally, a test was run to compare the numbers of keys and values retrieved by both scripts (key and value counters were inserted in both scripts). In most cases, both scripts retrieved the same amount of keys and values. The script `regp.pl` failed to complete successfully in two cases, therefore test data could not be produced. The script failed on files with root key of type 0xac; the initial loop of the script searches the file for a root key node of type 0xc and could not terminate when the input file contained a root key of type 0xac.

Table 28 Number of keys and values recovered by regparser.pl and regp.pl

Test file	regparser.pl keys/values	regp.pl keys/values	Result
bcd-template	190/151	190/151	√
bcd-template2	99/75	99/75	√
components	61246/126110	61246/126110	√
components1	61660/117349	61660/117349	√
components2	65413/135986	65413/135986	√
default	725/2672	725/2672	√
default1	9200/11430	5204/7446	failed
default2	335/1504	335/1504	√
default3	304/1338	304/1338	√
default4	447/2174	447/2174	√
ntuser.dat	1397/7093	1397/7093	√
ntuser1.dat	1144/5561	1144/5561	√
ntuser2.dat	3062/14524	3062/14524	√
ntuser3.dat	10319/30861	10319/30861	√
ntuser4.dat	17691/65993	17691/65993	√
ntuser5.dat	10249/29992	10249/29992	√
ntuser6.dat	3720/13212	3720/13212	√
ntuser7.dat	473/881	473/881	√
ntuser8.dat	1724/10565	1724/10565	√
ntuser9.dat	3076/17012	3076/17012	√
ntuser10.dat	2488/13536	2488/13536	√
ntuser11.dat	2030/12773	2030/12773	√
ntuser12.dat	2164/11802	2164/11802	√
ntuser13.dat	9844/27648	9844/27648	√
sam	54/61	54/61	√
sam1	78/84	78/84	√
sam2	52/58	52/58	√
sam3	82/97	82/97	√
sam4	52/59	52/59	√
security	225/224	225/224	√
security1	326/335	326/335	√
security2	75/74	75/74	√
security3	90/99	90/99	√
security4	81/80	81/80	√
software	105061/163690	61464/112036	failed
software1	190526/291428	116546/203083	failed
software2	110053/183434	64550/119830	failed
software3	105619/163742	N/A	N/A
software4	150372/228628	N/A	N/A
system	13421/41543	11483/37667	failed
system1	16358/49704	14415/45824	failed
system2	28474/65071	25803/60018	failed
system3	27020/58000	24342/52930	failed
system4	32955/71933	30279/66860	failed
userdiff	425/905	425/905	√
userdiff1	425/905	425/905	√
usrclass.dat	3963/31103	3963/31103	√

In several cases, the script regparser.pl retrieved more keys and values, because test files have a tree path containing "ri"->"lh" subkey list combination, which `regp.pl` does not support. As a result, "lh" sub key list was not parsed and regp.pl did not retrieve any

of the keys residing in this branch of the tree. The test has shown that `regparser.pl` parses the registry tree more successfully than `regp.pl` script.

4.4.4 Calculation of unallocated space

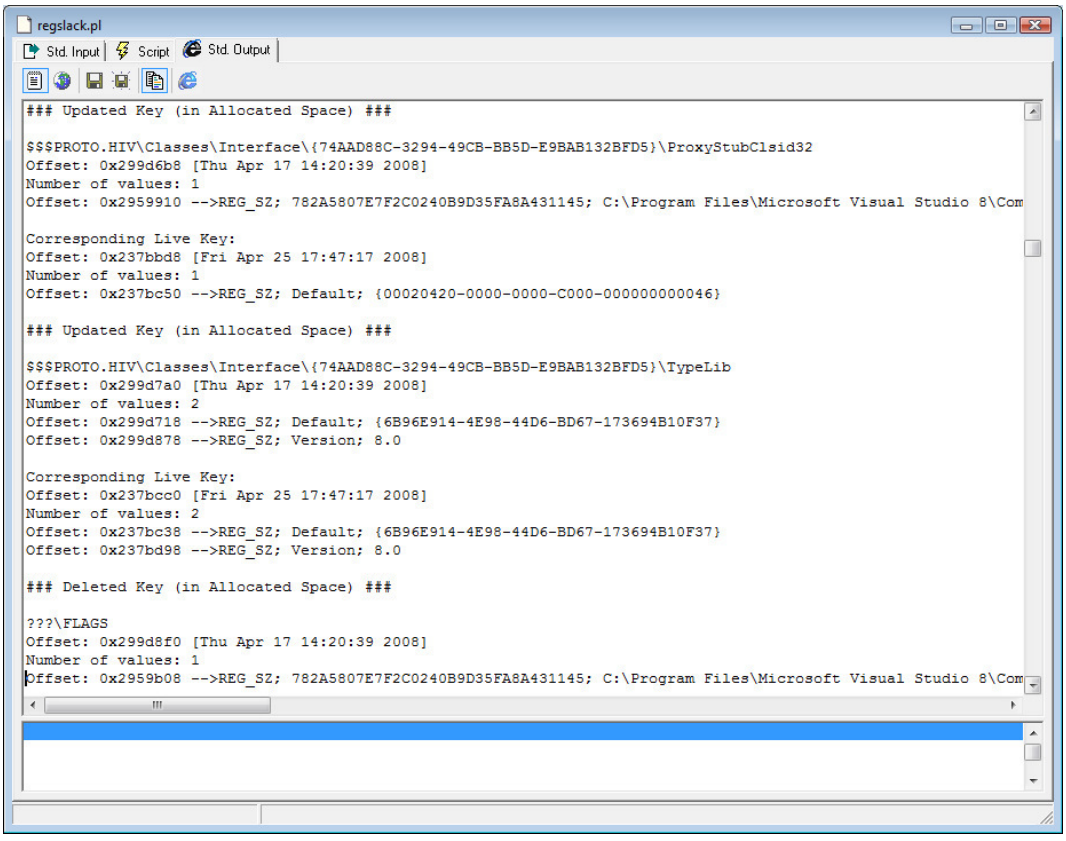
A script called `regslack.pl` (source code is included in appendix A.2) implements both the calculation of the unallocated space and the recovery of deleted keys and values. The chosen method for the calculation of the unallocated space parses the registry tree in order to calculate the space referenced by the tree and computes the unallocated space as all the remaining space of the hive file. If all of the allocated space of the registry was accessed by parsing the registry tree, all the remaining cells should have positive sizes. The tests performed on the complete set of test files have surprisingly shown that it is not always the case.

Three test files (`ntuser13.dat`, `software4` and `system4`) were copied from the running system using `ERUNT` tool developed by Hederer (2005). Tests have shown that those files have somewhat different structure than files copied from shutdown systems, and contained keys and value cells with negative sizes, that were not reached by parsing the registry tree. Additionally a hive file included in a tool DVD (Carvey, 2007b) called `ntuser1.dat` exhibits the same behavior which indicates that the file was also retrieved from a running system. Since analyzing hive files from a running system is out of the scope of this project, those files were excluded from the following tests. Once those files were excluded, the script `hivestructure.pl` was rerun and bin headers were now only found at offsets divisible by 0x1000 bytes, showing that the original assumption about bin sizes seems to be correct in a post mortem scenario.

Code was added to `regslack.pl` to display cells with negative sizes that were not reached by parsing the registry tree. As sample outputs show (see appendix B.3), even after the exclusion of the files copied from live system, `SOFTWARE` and `SYSTEM` hive files contain cells with negative sizes in the computed unallocated space. Those files do not

seem to contain any offsets that reference those cells. Those mysterious cells do not contain any live keys or values, but in a couple of software hive files deleted keys and values were found. There is not enough test data to conclude whether this mystery only occurs in `SOFTWARE` and `SYSTEM` files. The approach to compute unallocated space as not referenced by the tree is more effective, as those keys and values would not been found if unallocated space was computed based on cell sizes. The algorithm was slightly adjusted to clearly indicate the cases were keys are recovered from cells with negative sizes.

Figure 10 Recovered keys from unallocated cells with negative sizes



4.4.5 Recovery of deleted keys and their values

The table below shows the comparison of total numbers of "live", recovered and rejected keys and values for each test file. There is not enough test data to look for eventual patterns. Since key and values must have been deleted in order to reside in unallocated space, it makes sense that `DEFAULT` files produce small number of recovered keys, as they are updated infrequently. The more keys have been deleted since the operating system was installed, the more keys should be recovered, therefore recently installed or little used systems will produced less recovered keys. The number of rejected keys and values has to be related to the number of inserted keys and values (which have overwritten the previously deleted data). It also seems reasonable that the number of the recovered keys and values influences the number of rejected keys and values.

Table 29 Numbers of "live", recovered and rejected keys and values

Test file	Live keys/values	Recovered keys/values	Rejected keys/values
bcd-template	190/151	0/0	0/0
bcd-template2	99/75	0/0	0/0
components	61246/126110	0/0	0/0
components1	61660/117349	41/58	0/0
components2	65413/135986	125/54	0/0
default	725/2672	0/0	0/0
default1	9200/11430	6/1	0/2
default2	335/1504	0/0	0/0
default3	304/1338	0/0	0/0
default4	447/2174	0/0	0/0
ntuser.dat	1397/7093	15/24	0/2
ntuser2.dat	3062/14524	15/10	0/39
ntuser3.dat	10319/30861	0/0	2/0
ntuser4.dat	17691/65993	1/14	0/0
ntuser5.dat	10249/29992	2/0	0/0
ntuser6.dat	3720/13212	9/35	0/0
ntuser7.dat	473/881	0/0	0/0
ntuser8.dat	1724/10565	0/0	0/0
ntuser9.dat	3076/17012	19/40	0/10
ntuser10.dat	2488/13536	1/0	0/0
ntuser11.dat	2030/12773	1/0	0/0
ntuser12.dat	2164/11802	0/0	0/0
sam	54/61	4/5	0/0
sam1	78/84	0/0	0/0
sam2	52/58	1/1	0/0
sam3	82/97	0/0	0/0
sam4	52/59	6/7	0/0
security	225/224	2/1	0/1

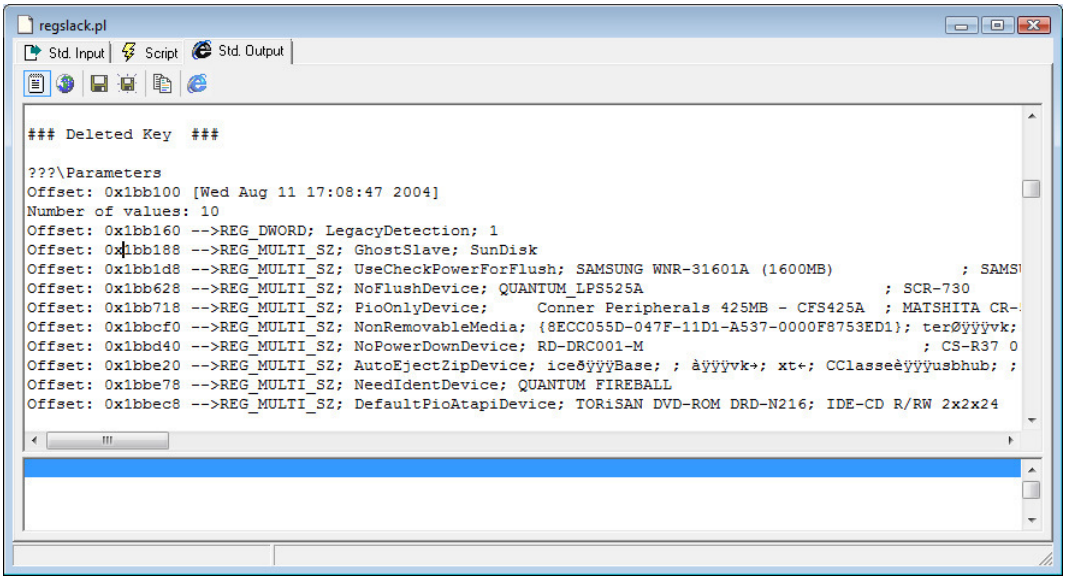
security1	326/335	0/0	0/0
security2	75/74	6/6	0/0
security3	90/99	0/0	0/0
security4	81/80 values	0/0	0/0
software	105061/163690	684/112	2/3
software1	190526/291428	5059/5986	6/147
software2	110053/183434	415/929	7/219
software3	105619/163742	16/30	0/2
system	13421/41543	0/0	0/0
system1	16358/49704	7090/18268	93/2261
system2	28474/65071	14198/31340	328/680
system3	27020/58000	11327/22800	324/43
userdiff	425/905	0/0	0/0
userdiff1	425/905	0/0	0/0
usrclass.dat	3963/31103	0/0	0/0

Also 28 keys in `software1` file and 8 keys in `software2` file were recovered from cells not reached by the tree but carrying negative sizes.

A code segment was inserted into regslack.pl to output the keys and values, which were rejected by the algorithm for being corrupted. . As sample outputs show in appendix B.4, the algorithm does a very good job at rejecting corrupted data. Rejected keys have for the most part no timestamp, and if they do, key names are clearly invalid, as they contain arbitrary data instead of ASCII text. Keys have mostly obscure value types, while valid types are in range 0-11 or 500 and 1000 ranges in SAM files. In cases where value type is within the valid range, the name and the value data show that the retrieved data was corrupted.

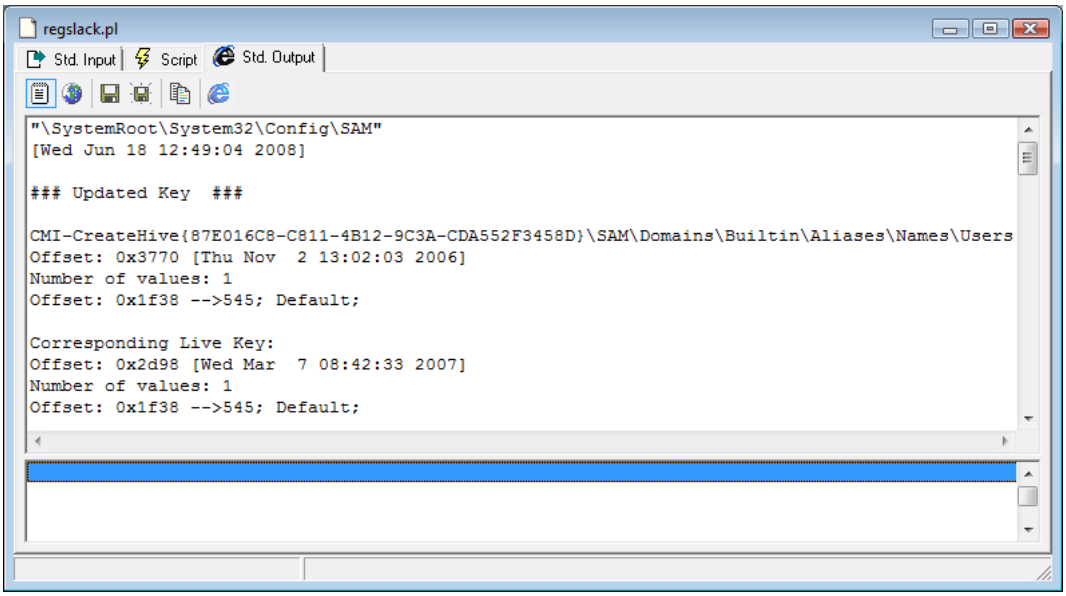
Appendix B.5 shows sample outputs with recovered keys and values, and appendix B.6 list a full output from a single (small) hive file. The recovered data is intact in the majority of cases and is displayed as it was intended. However, as previously discussed, there are no guarantees that recovered key names, value names or value data are not overwritten. Figure 11 Recovered data that is partially overwritten shows an example output were two of the recovered values seem to have been partially overwritten. It is impossible to completely avoid this problem, because tests of "live" registry values have shown that value data strings can contain any characters.

Figure 11 Recovered data that is partially overwritten



Finally, as below figure shows, a recovered key and a "live" key can refer to the same value cell. The two key cells reside at different offsets, but point to a value at the same offset, which indicates that the value resides in the "live" registry and can therefore be updated at any time.

Figure 12 Recovered key and "live" key referring to the same value



The above findings lead to a conclusion that despite the careful validation, it is not possible to guarantee that no corrupted data will be retrieved or that the retrieved data does not belong to another key or value.

Chapter 5. RESULTS AND EVALUATION

5.1 Results

The project delivers a documentation of hive file structure, demonstrates that keys can be recovered from unallocated space of the registry hive files, and implements a tool that can achieve that.

The main limitation of the project is a limited set of test hive files. Files originate from only two Windows platforms: Windows XP and Windows Vista (32-bit version) and all hive files originate from English versions of operating systems.

At the DFRWS conference in August 2008 Morgan (2008) presented a new paper on recovering deleted data from the Windows registry while I presented a demo of the developed prototype. There was a considerable interest in the subject from the attendees of the conference, which confirms the need for tools that recover deleted keys from unused space of the registry.

Morgan's algorithm calculates unallocated space based on cell sizes (positive cell size implies that the cell is unallocated). If a key signature is located in unallocated space, the key is recovered and the data segment is marked as allocated. The algorithm proceeds in a similar manner to recover value cells and security descriptor cells from the remaining unallocated space and finally outputs whatever is left as a hex dump. Morgan's algorithm has a weakness in that if data is partially overwritten, it may contain another key or value that will not be parsed by the algorithm. For example, a key name length that contains an arbitrary large number will result in recovery of a large chunk of data, displaying it as text and marking it as already parsed. Finally, Morgan does not correlate his results with the active registry entries, missing out on the fact that a recoverd key migh not have been deleted but only updated.

Morgan makes a valid observation that cell sizes can be manipulated to hide data, and that therefore calculation of unallocated space based on cell sizes would not discover the hidden data. A comparison of both methods has shown that calculation of unused space as not referred to by registry tree results in recovery of more keys. It is unclear why some cells are marked as allocated but seemingly not referenced by the registry tree; however, this project has shown that those cells can contain deleted keys. This leads to a conclusion the that method proposed in this dissertation for computing unallocated space, as a space not referred to by the registry tree is more effective.

Since any application can access a registry, data does not always conform to specifications. The delivered tool displays all binary data as a hex dump; therefore, any text stored as binary data will be readable. Wong (2007) argues that text strings could be encoded in hexadecimal format and stored as strings. The final algorithm could be easily modified to display string values as both strings and hex dump to eliminate that possibility.

The main challenge in recovery of deleted keys and values is validation of data. Key names and value names are stored as last fields, if only the last field is overwritten the algorithm will not detect that and corrupted data will be recovered. The same problem occurs when value data is stored in a separate cell, as no other fields can be validated.

The final mystery is why deleted keys sometimes refer to "live" values. Although seldom occurring, this phenomenon can result in recovery of values that could have been modified after a key referring to them was deleted. The final algorithm could be modified to indicate if a recovered value is also referred by a "live" key, to alert a forensic investigator that the value might have been changed.

5.2 Evaluation

"Jolanta Thomassen has invested a considerable amount of time and effort into a rather unique thesis project and has accomplished a great deal in a relatively short

amount of time. When she first approached me via email in the Spring of 2008, asking for thoughts or ideas for a thesis project, I offered her the idea of exploring the possibility of locating and extracting deleted Registry keys from within unallocated space within the hive files themselves. At the time, online searches for this subject matter revealed questions asked as far back as 2001, but there had been relatively little work, if any, done in the ensuing time. Ms. Thomassen picked up the subject and delved into what might appear to be extremely technical and "arcane" material and in doing so, has done a fantastic job in not only understanding the material, but also programmatically demonstrating her understanding through the development of Perl code to locate and extract those deleted cells.

Throughout the development of her thesis, Ms. Thomassen shared her Perl script and methodology with me. As an incident responder and forensic analyst in the corporate consulting field in the United States, I could immediately see the usefulness and applicability of the fruits of her labor. Forensics analysts do not rest an entire examination on a single artifact, and Ms. Thomassen's research and development of working code has opened yet another door for us, revealing another avenue of analysis. The code itself allows us to verify both her findings and ours, while automating the collection of data. All of this is extremely valuable to forensic analysts.

At the time she was working on her thesis, to the best of my knowledge, she was the only person conducting research into this field. At the DFRWS conference in Baltimore, MD, in August 2008, another researcher, also working in isolation, presented a paper on his own findings.

Ms. Thomassen's research and dedication to seeing this project through to completion have resulted in a very valuable contribution to the field of computer forensic analysis. I and others will forever be in her debt for taking on this arduous task and providing not only an understanding of the technical aspects of her research but also working code to demonstrate it." (Carvey, 2008b)

Chapter 6. CONCLUSIONS

6.1 Lessons Learned

The project has gathered, verified, and extended the existing knowledge about registry hive files. The project proposes a method for calculation of the unallocated space as space not reffered to by the registry tree and shows that the method is more effective than calculation based on cell sizes. The main observation is that deleted keys can be recovered from the registry and Windows registry forensics should include analysis of the unallocated space to obtain more complete information about the machine and activities performed on it.

6.2 Prospects for Further Work

The project answers some questions and poses several new ones. The documentation of registry hives is still incomplete, many cell fields remain undocumented, and some cells marked as allocated do not seem to be referenced by the registry. There is not enough knowledge about how Windows registry functions perform updates and deletions or how different registry cleaners modify the registry. Applications that access the registry typically use Windows registry functions, but tools that modify registry without use of those functions are being developed. An editor that can modify registry hive files could give a more complete indication of a valid structure of hives and valid ranges of data, by testing if a modified file is accepted or rejected by the Windows operating system. The existing registry forensic tools should include analysis of keys residing in unallocated space to obtain a more complete information about the investigated machine, user activities and timeline information.

REFRENCES CITED

ActiveState (2008) *ActivePerl - The complete and ready-to-install Perl distribution.* [Online] Available from: http://www.activestate.com/Products/activeperl/feature_list.mhtml (Accessed: 02 November 2008)

B.D. (n.d.) *WinReg.txt* [Online] Available from: http://home.eunet.no/pnordahl/ntpasswd/WinReg.txt (Accessed: 02 November 2008)

BreakPoint Software, Inc. (2008) *Hex Workshop, the Professional Hex Editor*. [Online] Available from: http://www.hexworkshop.com/ (Accessed: 02 November 2008)

Carvey, H. (2007a) *regp.pl. Offline Registry Parser.* Windows Forensic Analysis Including DVD Toolkit. Elsevier Science. ISBN-13: 9781597491563

Carvey, H. (2007b) *Windows Forensic Analysis Including DVD Toolkit*. Chapter 4: Registry Analysis. Syngress. ISBN-13: 9781597491563. pp. 125-189.

Carvey, H. (2008a) (keydet89@yahoo.com), 16 March 2008. RE: Would you be my MSc IT dissertation sponsor? Email to JT (jolantathomassen@hotmail.com).

Carvey, H. (2008b) (keydet89@yahoo.com), 09 October 2008. RE: Re: analysis and design Email to JT (jolantathomassen@hotmail.com).

Carvey, H. (2008c) LinkedIn public profile. [Online] Available from: http://www.linkedin.com/in/hcarvey (Accessed: 02 November 2008)

Carvey, H. (2008d) *RegRipper.* Windows Forensic Analysis. Dedicated to incident response and computer forensic analysis topics, with respect to Windows 2000, XP, 2003, and Vista operating systems. [Online] Available from: http://www.regripper.net/RegRipper/Documents/regripper.pdf (Accessed: 02 November 2008)

Chang, K et al. (2007) *Initial Case Analysis Using Windows Registry in Computer Forensics.* Future Generation Communication and Networking. Volume 1, 6-8 Dec. 2007 Page(s):564 – 569. [Online] DOI: 10.1109/FGCN.2007.151 (Accessed: 02 November 2008)

Clark, P. (2005) *Security Accounts Manager* [Online] Available from: http://beginningtoseethelight.org/ntsecurity/index.php (Accessed: 02 November 2008)

Hederer, L. (2005a) *ERUNT*. The Emergency Recovery Utility NT. Registry Backup and Restore for Windows NT/2000/2003/XP/Vista. [Online] Available from: http://www.larshederer.homepage.t-online.de/erunt/ (Accessed: 02 November 2008)

Hederer, L. (2005b) *NTREGOPT*. NT Registry Optimizer. Registry Optimization for Windows NT/2000/2003/XP/Vista. Detailed Information. [Online] Available from: http://www.larshederer.homepage.t-online.de/erunt/ (Accessed: 02 November 2008)

Microsoft (2002) *Microsoft® Computer Dictionary*, Fifth Edition. Microsoft Press. ISBN 9780735614956

Microsoft (2006) *Regedit.exe Cannot Search for DWORD or Binary Data.* KB161678 [Online] Available from: http://support.microsoft.com/kb/161678 (Accessed: 02 November 2008)

Morgan, T.D. (2008) *Recovering deleted data from the Windows registry*. Science Direct. Digital Investigation 5 (2008) S33 – S41 [Online] Available from: http://www.dfrws.org/2008/proceedings/p33-morgan.pdf (Accessed: 02 November 2008)

MSDN (2008a) *FILETIME Structure.* [Online] Available from: http://msdn.microsoft.com/en-us/library/ms724284(VS.85).aspx (Accessed: 02 November 2008)

MSDN (2008b) *Registry Element Size Limits* [Online] Available from: http://msdn.microsoft.com/en-us/library/ms724872(VS.85).aspx (Accessed: 02 November 2008)

MSDN (2008c) *Registry Functions* [Online] Available from: http://msdn.microsoft.com/en-us/library/ms724871%28VS.85%29.aspx (Accessed: 02 November 2008)

Nordahl-Hagen, P. (2008) *ntreg.h - NT Registry Hive access library, constants & structures* [Online] Available from: http://home.eunet.no/pnordahl/ntpasswd/ (Accessed: 02 November 2008)

Perl Express (2005) *A Free Perl IDE/Editor for Windows.* Version 2.5. [Online] Available from: http://www.perl-express.com/ (Accessed: 02 November 2008)

Probert, D.B. (n.d.) *Windows Kernel Internals.* NT Registry Implementation. Microsoft Corporation. [Online] Available from: http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/09-Registry/Registry.ppt (Accessed: 02 November 2008)

Russinovich, M. (1999) *Inside the Registry.* Windows NT Magazine. [Online] Available from: http://technet.microsoft.com/en-us/library/cc750583.aspx (Accessed: 02 November 2008)

Wong, L. W. (2007) *Forensic Analysis of the Windows Registry*. Forensic Focus. [Online] Available from: http://www.forensicfocus.com/index.php?name=Content&pid=73&page=1 (Accessed: 02 November 2008)