

TZWorks® Yet Another Registry Utility (*yaru*) Users Guide



Abstract

yaru is a GUI registry utility that can display the internal registry hive components and structures. **yaru** can operate on a registry hive directly from a live volume, an image of a volume or a *VMWare* volume. **yaru** runs on Windows, Linux and Mac OS-X.

Copyright © TZWorks LLC

www.tzworks.com

Contact Info: info@tzworks.com

Document applies to v1.90 of **yaru**

Updated: Apr 15, 2024

Table of Contents

1	Introduction	2
2	Registry Hive and Components.....	3
3	Location of Hives.....	4
4	How to Use <i>yaru</i>	4
4.1	Reading Registry Hives from Logical Images.....	5
4.2	Parsing Hives from a Live Volume.....	6
4.3	Common Registry Artifacts useful to Forensic Investigators.....	6
4.4	Searching for Text Patterns.....	8
4.5	Searching for Binary Patterns	9
4.6	Searching for Entries exceeding some threshold size.....	11
4.7	Searching for High Entropy data	11
4.8	Searching for Time Ranges.....	12
5	Unlinked Allocated Chunks	13
6	Deleted Registry Keys	14
7	Exporting Keys and Data	16
8	Brute Force Extraction of Keys – Carving.....	18
9	Validation of Parsed Residuals	20
10	Logging of Activities	22
11	Creating a “Send To” Shortcut for <i>yaru</i>	24
12	Command Line Options.....	24
13	User Defined Templates.....	25
14	Known Issues.....	25
15	X-Window Dependencies.....	25
16	Authentication and the License File.....	25
16.1	<i>Limited</i> versus <i>Demo</i> versus <i>Full</i> in the tool’s Output Banner.....	26
17	References	27

TZWorks® Yet Another Registry Utility (*yaru*) Users Guide

Copyright © TZWorks LLC

Webpage: http://www.tzworks.com/prototype_page.php?proto_id=3

Contact Information: info@tzworks.com

1 Introduction

yaru is a platform independent Windows registry viewer. Inspired by the desire to look into the Windows registry metadata, so as to better forensically analyze the registry hives, **yaru** was designed with a portable and extensible architecture in mind so that it could be compiled to run on various operating systems. The registry parsing engine is written in standard C/C++ and has no dependencies on the Windows registry API functions. This means that the parsing may have trouble on certain untested boundary conditions.

The GUI portion of **yaru** leverages off the FOX (Free Objects for X) library, which was designed to be cross platform. The FOX library is freely available and is distributed in source form under Library GNU Public License (LGPL). Currently, there are compiled versions of **yaru** that will run on Windows, Linux and OS-X.

The Windows version of **yaru** has the ability to take a snapshot of any of the active hives and examine the internal structure of the hive. Since the Windows operating system locks down the active hives from other processes reading them, **yaru** can resort to raw *NTFS* disk reads to read any of the desired hives. Consequently, this requires the user to run this tool with administrative privileges. While this approach adds complexity to **yaru**, it ensures that all metadata is available for analysis, as well as ensures that there is no corruption or changes to the active hive during analysis.

Some other rudimentary functionality includes:

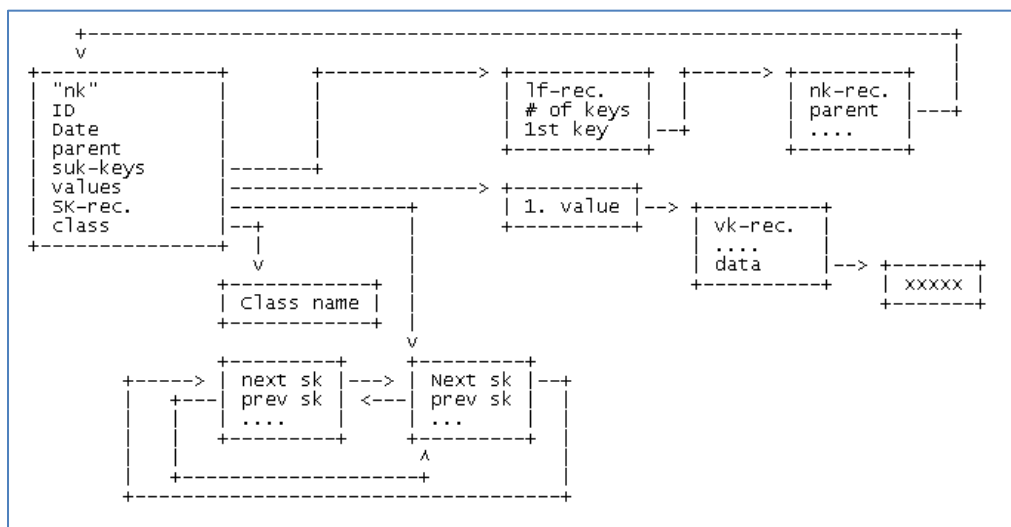
- Show allocated (but unused) key value data space [referred to here as cell slack space].
- Show unallocated hive space [referred to here as hive slack space].
- Able to traverse the hive slack space and enumerate deleted keys.
- Report generation capability. For common registry forensics artifacts, a number of options are available to generate reports from the live hives, copies of hives or hives from unmounted partition files. The latter requires a bit-for-bit (uncompressed) copy of the partition image.
- Optional logging capability that records the user selections along with data values into a separate XML file for later review. A separate XML file is created for each session.
- Ability to export any key in the hive under evaluation to a registration (.reg) file to be used for analysis. The format tries to mimic the version 5.00 of the Windows registry editor, with some additional metadata in commented form.

- Ability to process any hive using user defined templates. These templates allow one to customize what data is to be extracted. While these templates have a very primitive set of commands, they can be useful for repetitive tasks.
- Simple search capability: (a) key names, (b) value names, (c) date ranges, and (e) strings (that greater than 4 characters)
- The ability to verify that all allocated chunks have valid links to the registry. This was discussed in Timothy Morgan's paper [ref 8] as an anti-forensics technique.

2 Registry Hive and Components

When talking about the forensics artifacts in the Windows registry, some discussion on the architecture of the registry is in order. According to the Windows 2000 server resource kit ⁽¹²⁾, the registry “is a hierarchical database that contains the value of variables in Windows ... and in the applications and services that run on Windows... The registry consists of nested containers known as subtrees, keys, and subkeys. These are like folders. The data is... stored in the registry entries, the lowest element in the registry. The entries are like files.... An entry consists of a name, a data type, which defines the length and format of data that the entry can store, and a field known as the value of a registry entry.”

Unfortunately, the Windows registry internals are Microsoft proprietary. Therefore, finding an open source document that accurately documents the internals without error, is difficult and any data in the open is most likely derived from empirical results from looking at hexadecimal dumps of raw registry hives. The first attempt to document the internals of the registry was from a document written many years ago (circa 1998) that was distributed on the Internet from an author identified with only the initials ‘BD’ ¹⁰. Below is a screen shot of a diagram used in BD’s document. While the figure was titled a “*Greatly Simplified Structure of the NT Registry*”, it appeared to have accurately shown the major key/value/data components and their interrelationships. Along with the diagram contained some definitions of the structures for each of the blocks.



BD's diagram refers to registry key name structures as “nk” and key value structures as “vk”. The security key associated with each registry key is shown as “sk”. This nomenclature was based on each structure’s respective signature when looking at a binary dump of each type (eg. “nk”, “vk”, “sk”). This is consistent with what other authors have published nearly a decade later. This included various articles in the Microsoft Development Network, Harlan Carvey’s section on registry analysis in his book on Windows Forensic Analysis, and more recently, published papers by Thomassen¹¹ and Norris¹² to name a few.

Taking the results from all these open sources and arming oneself with a hex editor, one can use the structures documented thus far to manually walk the entire registry with reasonable accuracy.

3 Location of Hives

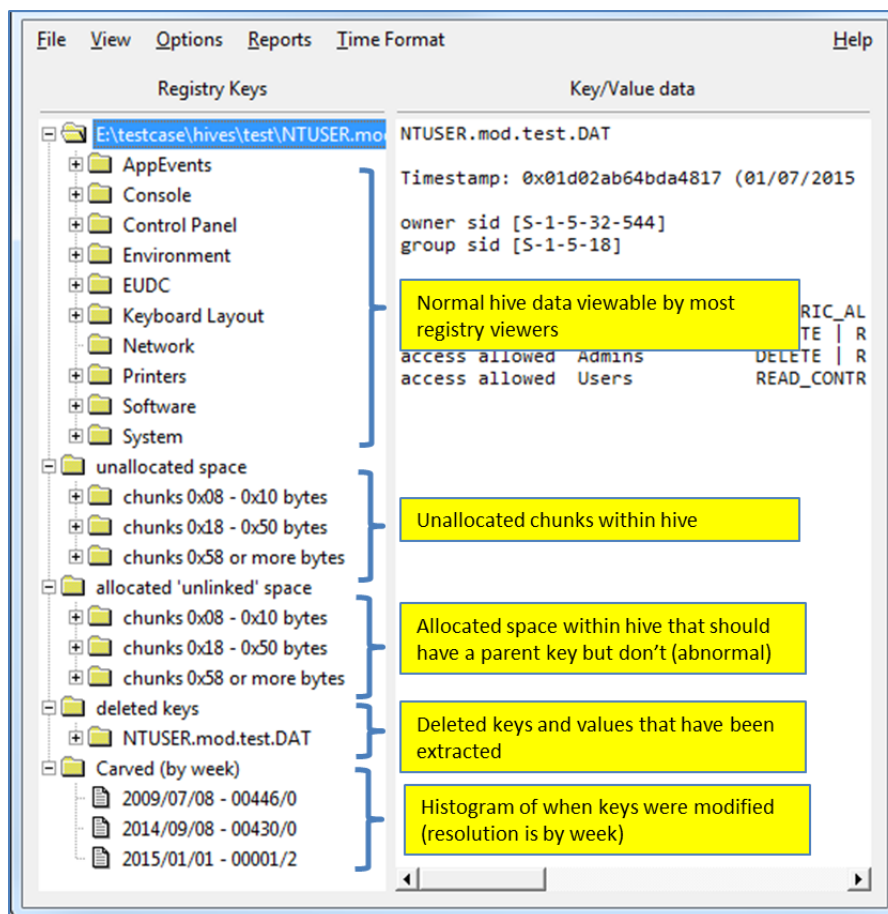
The registry hives are in various locations, depending whether they are system related or user account related. Some of the more common registry hives can be found in the following locations:

Hive	Location
Ntuser.dat	%userprofile%\ntuser.dat
UsrClass.dat	(xp) %userprofile%\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat (vista and later) %userprofile%\AppData\Local\Microsoft\Windows\UsrClass.dat
System	%systemroot%\system32\config\system
Sam	%systemroot%\system32\config\sam
Software	%systemroot%\system32\config\software
Security	%systemroot%\system32\config\Security
Components	(vista and later) %systemroot%\system32\config\Components
BCD	(vista and later) %systemdrive%\boot\bcd
Syscache.hive	(vista and later) System Volume Information\ Syscache.hive
Schema.dat	%systemroot%\System32\SMI\Store\Machine\schema.dat
AmCache.hve	(win 8 and later) %systemroot%\AppCompat\Programs\AmCache.hve
ELAM	(win8 and later) %systemroot%\system32\config\elam
BBI	(win8 and later) %systemroot%\system32\config\bbi
DRIVERS	(win8 and later) %systemroot%\system32\config\drivers

4 How to Use *yaru*

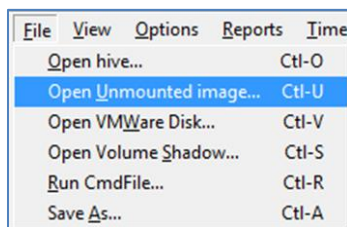
When a hive is loaded into *yaru*, the hive is broken up into 4 main segments: (a) the normal hive data that is viewable by the normal registry editors, (b) the unallocated space within the hive, (c) any allocated space that should have a parent key but does not, and (d) any deleted keys and their associated values that have not been overwritten. The 5 segment shown is for an experimental option to carve the hive that is invoked via the *Options Menu*. The carve option will group all the keys it finds

as a function of their modification date forming a quasi-histogram. The histogram is broken up into valid and deleted keys giving additional insight to the registry changes on a particular period. The carving option is discussed in more detail in a later section in this user guide.



4.1 Reading Registry Hives from Logical Images

Under certain conditions, **yaru** can read the registry hives directly from a logical image that was saved as a file (without mounting the image as a file system). There is one basic assumption that **yaru** makes when reading the unmounted partition, is that the NTFS unmounted partition is a single file and is a binary match of the original logical partition. One can do this via the File -> Open Unmounted Image:



The other option is to open the hive via the command line via the following switch:

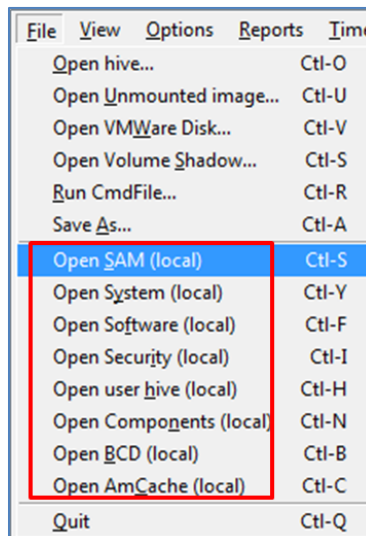
-ntfsimage <unmounted partition> <path\file of the hive>

Here's an example. Note that since the registry path of the hive is not mounted, it does not have a drive letter when specifying where the hive file is. Thus the path is relative to root.

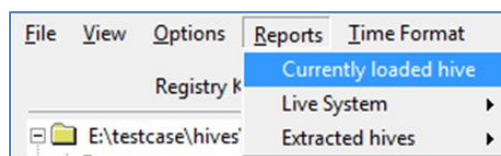
-ntfsimage c:\test\image1.dd \Windows\System32\Config\system

4.2 Parsing Hives from a Live Volume

To load a hive from a system volume one can use the shortcuts in the menu. There are options for each of hives. For those hives that are in more than one location, such as the user hives, if they are selected a menu will allow the user to choose which user hive to load.



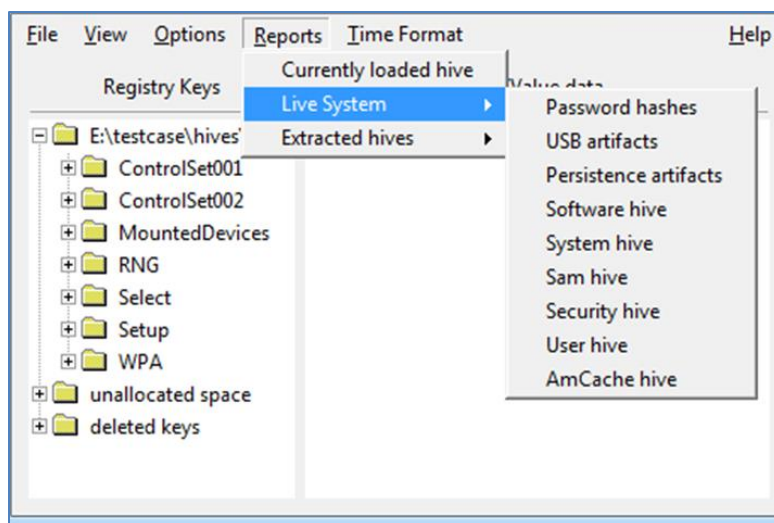
During the load operation, **yaru** scans the hive for deleted registry entries as well as indexes the hive for faster searching. Once the load operation completes one can view any entry or scan for artifacts by selecting the Reports-> Currently loaded hive.



4.3 Common Registry Artifacts useful to Forensic Investigators

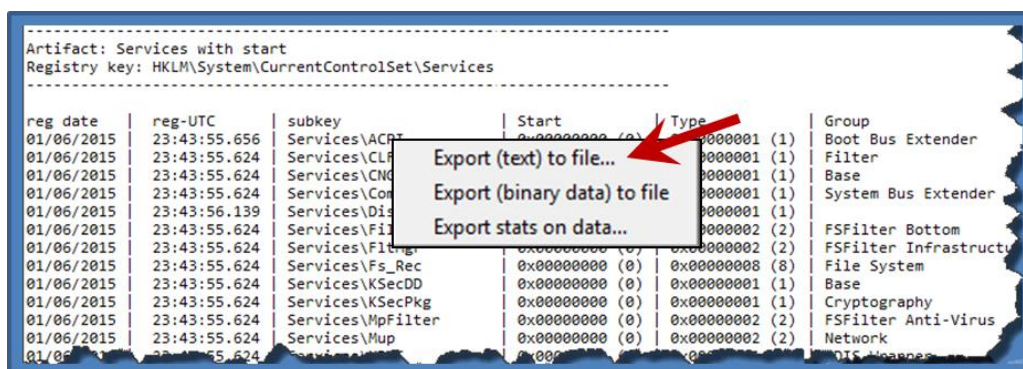
While one can dump the data associated with many keys that are of interest to investigators, it is useful to know the relationships between certain raw data and how these bits of data can paint a story of a

sequence of events. **yaru** groups some of the more common artifacts into canned reports. Shown below are the current groups for various hives.

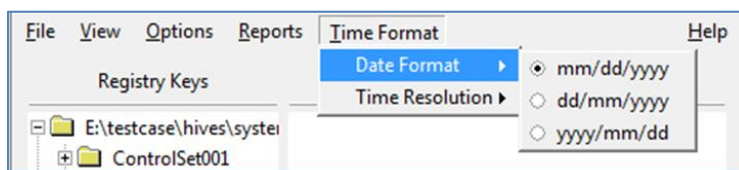


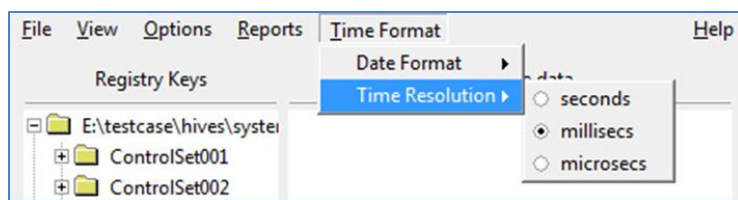
Selecting a report from the “Currently loaded hive” will review the current hive loaded in **yaru** and generate a report specific for that hive. Selecting a report from the “Live System” will load the proper hive based on the report selected and then generate a report.

Whichever report is selected, the results that are generated separate each field with a pipe delimiter to allow for easy viewing as well as inclusion into another tool (such as excel) for analysis. Below is a portion of a report from a system hive showing the various services. After the report is generated one can ‘right-click’ on the report output and select the “export text to file” option to copy the data to a file, which can be used elsewhere.



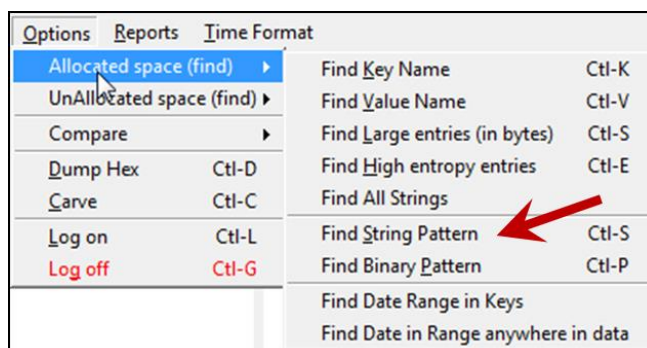
If one wishes to display the date/time in a different format, one can use the following menu options.





4.4 Searching for Text Patterns

One can search for partial names using the “*Find String pattern*” option. The string that is entered will be interpreted by yaru as ‘case insensitive’ and will scan for both Unicode and ASCII strings that have this partial string pattern. Note: for case sensitive searches use the “*Find Binary Pattern*” option.



The output results show (a) the offset of the string found, (b) whether the match was Unicode or ASCII, (c) the string that caused a hit, as well as (d) the governing path/key that encapsulated the value containing the data.

For example, if I wanted to find all the keys and values that have the letters “USB”, I would get something like this.

```

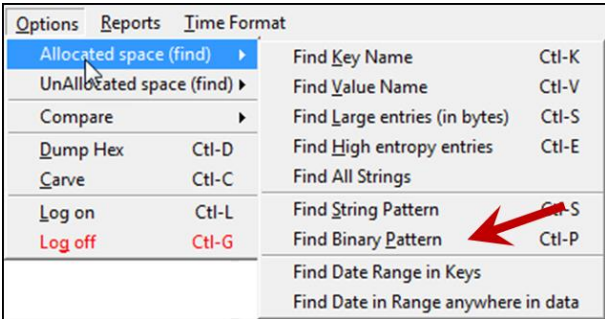
<offset>      : <string>      : <governing key [may be buried in one of the key/value struc
0x00098260 [a] :      USB Monitor : SYSTEM\ControlSet001\Control\Print\Monitors\USB Monitor\
0x000ab638 [a] :      wudfusbccidDriver : SYSTEM\ControlSet001\Control\SafeBoot\Network\wudfusbccid
0x000c2f28 [a] :      TsusbFlt : SYSTEM\ControlSet001\services\TsusbFlt\
0x000fd288 [a] :      usbhub : SYSTEM\ControlSet001\services\usbhub\
0x000fd340 [a] :      usbperf : SYSTEM\ControlSet001\services\eventlog\Application\usbperf\
0x0013ab38 [a] :      USB#VID_04F9&PID_014A&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013aea8 [a] :      BrusbSer : SYSTEM\ControlSet001\services\BrusbSer\
0x0013b138 [a] :      USB#VID_04F9&PID_0146&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013b310 [a] :      USB#VID_04F9&PID_0157&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013b518 [a] :      USB#VID_04F9&PID_010F&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013b720 [a] :      USB#VID_04F9&PID_0121&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013b928 [a] :      USB#VID_04F9&PID_0122&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013bb30 [a] :      USB#VID_04F9&PID_010E&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013bba0 [a] :      USB#VID_07CF&PID_1002 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013bcd0 [a] :      USB#VID_04CB&PID_0100 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013be88 [a] :      USB#VID_04F9&PID_0120&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013c070 [a] :      USB#VID_04F9&PID_013A&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013c280 [a] :      USB#VID_04F9&PID_0135&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013c488 [a] :      USB#VID_04F9&PID_0136&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013c690 [a] :      USB#VID_04F9&PID_013E&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013c898 [a] :      USB#VID_04F9&PID_013F&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013caa0 [a] :      USB#VID_04F9&PID_0111&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013cca8 [a] :      USB#VID_04F9&PID_011E&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d070 [a] :      USB#VID_04F9&PID_0125&MI_02 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d0e0 [a] :      USB#VID_04F9&PID_0110&MI_0210&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d2e8 [a] :      USB#VID_04F9&PID_0110&MI_0210&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d4f0 [a] :      USB#VID_04F9&PID_012B&MI_022B&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d6f8 [a] :      USB#VID_04F9&PID_012F&MI_022F&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013d900 [a] :      USB#VID_04F9&PID_0130&MI_0230&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013db08 [a] :      USB#VID_04F9&PID_0173&MI_0273&MI : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013e170 [a] :      USB#VID_054C&PID_0010&REV_0328 : SYSTEM\ControlSet001\Control\CriticalDeviceDatabase\USB#VID
0x0013f308 [a] :      USBSTOR : SYSTEM\ControlSet001\services\USBSTOR\

```

As can be seen from the output above, what is different about *yaru's* search engine, as opposed to a hex editor, is when the pattern is found the output displays the governing key that the pattern is in.

4.5 Searching for Binary Patterns

The binary pattern search option is shown below:



To use it properly, input the hex values delimited by spaces.

If one understands the internals of the registry, various structures can be searched for within the registry. Below is an example of searching for the 'db' type data structure on a system hive along with the results that are returned:

Search [quoted string or hex pattern delimited by spaces]

String/Pattern to find:

f0 ff ff ff 64 62

Optional offset:

0

Cancel

Find

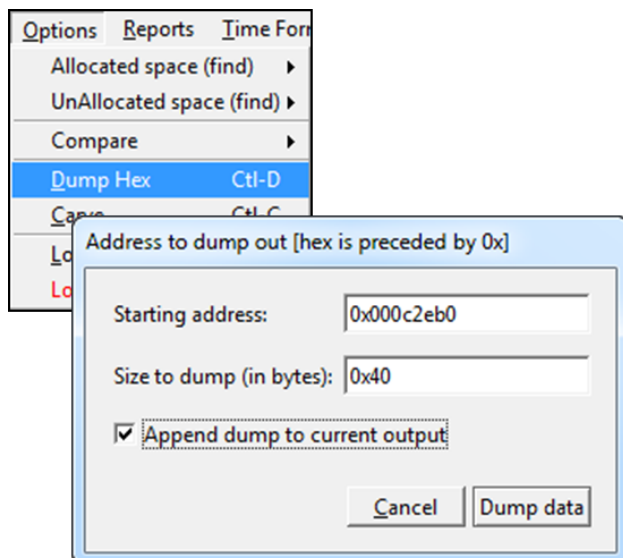
Since the 'db' structure is 0x10 bytes long with a 'db' signature, we crafted the pattern of bytes to be *negative* 0x10 bytes, which is 0xffffffff0 or in little endian format, f0 ff ff ff and the signature for 'd' is 0x64 and 'b' is 0x62. This allows the search to only return those allocated cells chunks that are 0x10 bytes in size and contain the 'db' signature following the negative size, which will return all the 'db' type chunks in the hive. The optional offset field is usually 0, but if one wanted to see the preceding bytes, one can put a negative offset if desired.

```
Binary pattern {f0 ff ff ff 64 62} in allocated space [8]

<offset> :      <pattern>                                     : <governing key [may be buried in one of the key values]>
-----
000c2eb0 : f0 ff ff ff 64 62 0d 00 e0 e5 e0 00 01 00 00 00 : SYSTEM\ControlSet001\Control\Session Manager\AppCompatCache
005a9568 : f0 ff ff ff 64 62 0d 00 00 72 c3 00 40 88 5a 00 : SYSTEM\ControlSet002\Control\Session Manager\AppCompatCache
005f45f0 : f0 ff ff ff 64 62 02 00 a0 37 5f 00 55 52 4c 00 : SYSTEM\ControlSet002\Control\ProductOptions\
006fc310 : f0 ff ff ff 64 62 04 00 60 b3 6f 00 f0 e1 6f 00 : SYSTEM\ControlSet002\services\ialm\Device0\
007522c8 : f0 ff ff ff 64 62 17 00 b0 da e7 00 00 00 00 00 : SYSTEM\ControlSet002\services\rdyboost\Parameters\
00945fe8 : f0 ff ff ff 64 62 02 00 10 51 94 00 00 00 00 00 : SYSTEM\ControlSet001\Control\ProductOptions\
0095c5c8 : f0 ff ff ff 64 62 15 00 20 f0 e8 00 00 20 00 00 : SYSTEM\ControlSet001\services\rdyboost\Parameters\
00bc4c18 : f0 ff ff ff 64 62 04 00 e8 b1 26 00 70 c4 bc 00 : SYSTEM\ControlSet001\services\ialm\Device0\
```

From the output above, one can see the governing keys that use this very large datatype (*AppCompatCache*, *ReadyBoost* parameters, etc.).

If desiring to see the raw data at one of the locations, one can select the “*Dump Hex*” choice from the “Options” menu, and the following dialog will pop up. After entering the offset to view, size to dump and whether you want the hex dump to be appended to the current output, one can see the desired data at the specified offset. For this example we just selected the first returned offset from the 'db' structure.



The result will show 0x40 bytes at offset 0xc2eb0 appended to data in the current view. In this case we were only interested in the first 0x10 bytes and added a few more to see the data that followed the

structure. This approach to reviewing the internal data is quick and provides immediate context of what subkeys the data is associated with.

Binary pattern {f0 ff ff ff 64 62} in allocated space [8]

<offset> :	<pattern>	:	<governing key [may be buried in one of the key values]>
000c2eb0 :	f0 ff ff ff 64 62 0d 00 e0 e5 e0 00 01 00 00 00 :		SYSTEM\ControlSet001\Control\Session Manager\AppCompatCache\
005a9568 :	f0 ff ff ff 64 62 0d 00 00 72 c3 00 40 88 5a 00 :		SYSTEM\ControlSet002\Control\Session Manager\AppCompatCache\
005f45f0 :	f0 ff ff ff 64 62 02 00 a0 37 5f 00 55 52 4c 00 :		SYSTEM\ControlSet002\Control\ProductOptions\
006fc310 :	f0 ff ff ff 64 62 04 00 60 b3 6f 00 f0 e1 6f 00 :		SYSTEM\ControlSet002\services\ialm\Device0\
007522c8 :	f0 ff ff ff 64 62 17 00 b0 da e7 00 00 00 00 00 :		SYSTEM\ControlSet002\services\rdyboost\Parameters\
00945fe8 :	f0 ff ff ff 64 62 02 00 10 51 94 00 00 00 00 00 :		SYSTEM\ControlSet001\Control\ProductOptions\
0095c5c8 :	f0 ff ff ff 64 62 15 00 20 f0 e8 00 00 20 00 00 :		SYSTEM\ControlSet001\services\rdyboost\Parameters\
00bc4c18 :	f0 ff ff ff 64 62 04 00 e8 b1 26 00 70 c4 bc 00 :		SYSTEM\ControlSet001\services\ialm\Device0\
000c 2eb0 :	f0 ff ff ff 64 62 0d 00 e0 e5 e0 00 01 00 00 00 :	db.....
000c 2ec0 :	e8 ff ff ff 6c 68 02 00 40 31 e6 00 fa b6 86 b4 :	lh...@1.....
000c 2ed0 :	78 03 77 00 a8 df 22 c5 a8 ff ff ff 6e 6b 20 00 :		x.w...".....nk .
000c 2ee0 :	34 64 d6 40 0c 9e cf 01 00 00 00 00 d0 01 00 00 :		4d.@.....

4.6 Searching for Entries exceeding some threshold size

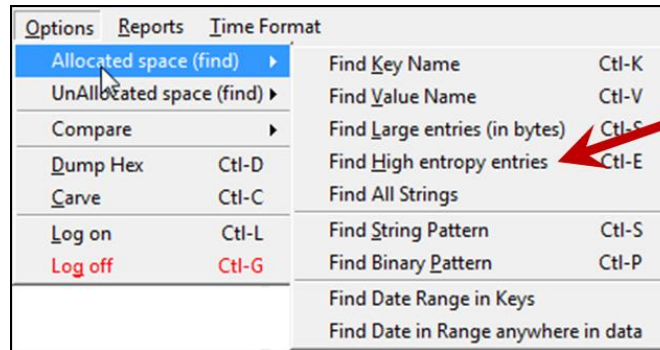
In one is concerned about searching on large registry values, one can use the “Find Large entries (in bytes)” option. This is shown below.

Options	Reports	Time Format
Allocated space (find) ▶	Find Key Name	Ctl-K
UnAllocated space (find) ▶	Find Value Name	Ctl-V
Compare ▶	Find Large entries (in bytes)	Ctl-S
Dump Hex	Find High entropy entries	Ctl-E
Carve	Find All Strings	
Log on	Find String Pattern	Ctl-S
Log off	Find Binary Pattern	Ctl-P
	Find Date Range in Keys	
	Find Date in Range anywhere in data	

When this option is selected one will be able to specify the number of bytes that is the threshold. The operation will search all the values in the registry returning those at or above the number specified.

4.7 Searching for High Entropy data

High entropy is another way to specify randomness in the data. Randomness is one of the artifacts in a dataset whenever it is encrypted or compressed, so computing the entropy of dataset is one of the ways to find encrypted values. In yaru, the option is “Find High entropy entries” and is shown below.



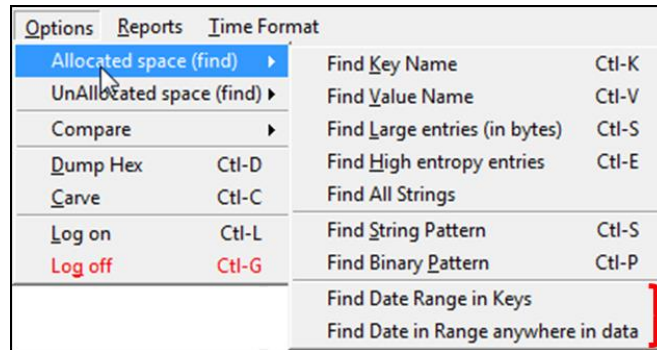
When this option is selected one will be able to specify the percent entropy desired to be the threshold as well as how many bytes to examine. The operation will search all the values in the registry returning those at or above the percent entropy specified. Other statistics will also be displayed such as the mean and standard deviation of the dataset. The data will be ordered with the highest entropy values first.

Randomness search results, from: 80% entropy and higher; examining all bytes in data.
Algorithm used is the Shannon-Wiener Diversity Index adjusted for Evenness converted to a percent.

reg date [UTC]	offset	size	%entropy	mean	std dev	value	subkey
06/03/2016 17:42:15.408	0x00142ad8	0x000001fc	93	119.65	79.922	PowerKey-SF-RC6-29	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x00142d30	0x000001fc	92	119.26	77.437	PowerKey-SF-QP-0C	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x001428a8	0x000001fc	92	118.18	77.483	PowerKey-SF-RC6-0C	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x00142658	0x000001fc	92	127.49	80.692	PowerKey-SF-SS-00	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x001421a8	0x00000154	90	118.06	75.927	PowerKey-BB-RC6-29	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x00142330	0x00000154	89	115.92	79.159	PowerKey-BB-QP-0C	SYSTEM\ControlSet001\service
06/03/2016 00:48:04.758	0x00eac310	0x00000100	88	120.20	80.260	Mapping	SYSTEM\ControlSet001\service
06/03/2016 00:48:04.758	0x00ba36f0	0x00000100	88	120.34	80.216	Mapping	SYSTEM\ControlSet002\service
06/03/2016 17:42:15.408	0x001424d0	0x00000154	88	113.98	77.987	PowerKey-BB-QP-29	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x00141e68	0x00000154	88	115.55	78.240	PowerKey-BB-SS-00	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x007c8020	0x000001fc	85	109.34	85.311	PowerKey-SF-QP-0C	SYSTEM\ControlSet002\service
06/03/2016 00:39:09.744	0x00baf520	0x00000166	84	107.00	83.663	EncryptedSettings	SYSTEM\ControlSet002\service
06/03/2016 00:39:09.744	0x00b15130	0x00000166	84	106.56	83.384	EncryptedSettings	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x00142020	0x00000154	83	104.81	81.215	PowerKey-BB-RC6-0C	SYSTEM\ControlSet001\service
06/03/2016 17:42:15.408	0x007c76d0	0x000001fc	80	99.27	85.636	PowerKey-SF-RC6-29	SYSTEM\ControlSet002\service
07/14/2009 04:49:21.130	0x00574a88	0x00000160	80	102.05	83.004	AuditPolicySD	SYSTEM\ControlSet002\Control

4.8 Searching for Time Ranges

yaru has two options for searching for timestamp ranges. The first is to scan through all the key/subkey timestamps. The second is to scan through all the binary data looking for timestamp signatures and displaying the governing key for the binary data. The governing key would include child values that have timestamps embedded into their data. The date range that is inputted by the user is in terms of UTC (prior versions to v1.39, used local time).



If using the first option, then the output will include only timestamp and path of the key. If using the second option, "Find Date in Range anywhere in the data", then the output will include the raw offset of the data along with timestamp and path of the governing key. The first option will only show one timestamp per key, which is what you would expect. The second option may show many timestamps per key. Below is some sample output for a sample query.

FILETIME search results, range: 01/07/15 00:00:00.000 to 01/08/15 00:00:00.000

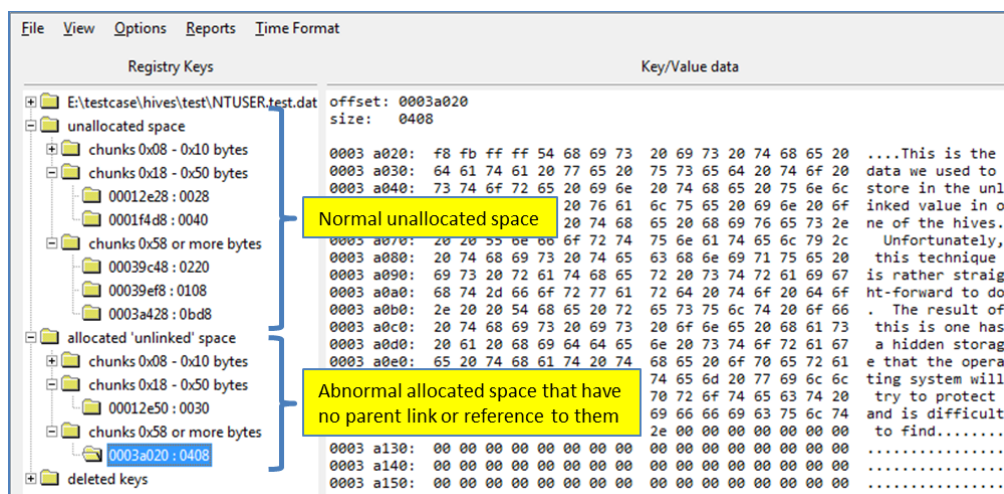
offset	time [raw value]	equivalent time	governing key [may be buried in one of the key values]
0x0001c418	0x01d02a9284b5dd5a	01/07/2015 15:56:37.384	SYSTEM\ControlSet001\services\BITS\
0x0007a0a8	0x01d02a2eb14979f2	01/07/2015 04:02:02.498	SYSTEM\ControlSet001\services\Dnscache\Parameters\DnsC
0x006db6c0	0x01d02a8a256b98a4	01/07/2015 14:56:41.540	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006db754	0x01d02a8a22f83a4f	01/07/2015 14:56:37.429	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006db7d4	0x01d02a8a22f84dd8	01/07/2015 14:56:37.429	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006db85c	0x01d02a8a233a4e85	01/07/2015 14:56:37.862	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006db904	0x01d02a8a249ad45b	01/07/2015 14:56:40.172	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006db994	0x01d02a8a249ad45b	01/07/2015 14:56:40.172	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x006dba4c	0x01d02a8a24acc2c8	01/07/2015 14:56:40.289	SYSTEM\ControlSet001\services\VSS\Diag\SystemRestore
0x011e4b94	0x01d02a9ba1cdfb7e	01/07/2015 17:01:51.667	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4bf4	0x01d02aa6e9c9864b	01/07/2015 18:22:36.898	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4c54	0x01d02aa68bd565e9	01/07/2015 18:19:59.270	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4c24	0x01d02aa03a629cc3	01/07/2015 17:34:45.641	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4cb4	0x01d02aa037346053	01/07/2015 17:34:40.305	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4ce4	0x01d02aa017fc41dc	01/07/2015 17:33:47.928	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4d14	0x01d02aa014c9e69a	01/07/2015 17:33:42.565	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4d44	0x01d02aa00b345b80	01/07/2015 17:33:26.485	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4d74	0x01d02aa004034da6	01/07/2015 17:33:14.420	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4d44	0x01d02aa000cc71e7	01/07/2015 17:33:09.027	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4dd4	0x01d02a9f7084d4cc	01/07/2015 17:29:06.966	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e4e04	0x01d02a9b2314f53f	01/07/2015 16:58:19.061	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5374	0x01d02a9a26296eb9	01/07/2015 16:51:14.732	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e53a4	0x01d02a9abeca76af	01/07/2015 16:55:30.801	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e53d4	0x01d02a99868ada2e	01/07/2015 16:46:46.935	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5404	0x01d02a995d055fdb	01/07/2015 16:45:37.274	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5434	0x01d02a8cc9a22fb1	01/07/2015 15:15:36.038	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5764	0x01d02a89e4aa18f6	01/07/2015 14:54:52.898	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5854	0x01d02a73c47ee300	01/07/2015 12:16:30.000	SYSTEM\ControlSet001\services\ialm\Device0\
0x011e5884	0x01d02a5a000cb380	01/07/2015 09:12:03.000	SYSTEM\ControlSet001\services\ialm\Device0\
0x01230970	0x01d02a88939b8cd9	01/07/2015 14:45:27.410	SYSTEM\ControlSet001\services\nt01ssndis\Parameters\Wd
0x01230a14	0x01d02a88939b8cd9	01/07/2015 14:45:27.410	SYSTEM\ControlSet001\services\nt01ssndis\Parameters\Wd
0x012c1a04	0x01d02ab481233780	01/07/2015 19:59:54.279	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c20c4	0x01d02ab481233780	01/07/2015 19:59:54.279	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2484	0x01d02ab42ef6193	01/07/2015 19:57:35.108	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c24b4	0x01d02ab347c46609	01/07/2015 19:51:08.531	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2514	0x01d02ab31a07aea0	01/07/2015 19:49:51.796	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2544	0x01d02ab292bc16c3	01/07/2015 19:46:04.808	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c25d4	0x01d02a9ba1cdfb7e	01/07/2015 17:01:51.667	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2634	0x01d02aa6e9c9864b	01/07/2015 18:22:36.898	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2664	0x01d02aa68bd565e9	01/07/2015 18:19:59.270	SYSTEM\ControlSet001\Control\Session Manager\AppComp
0x012c2694	0x01d02aa03a629cc3	01/07/2015 17:34:45.641	SYSTEM\ControlSet001\Control\Session Manager\AppComp

5 Unlinked Allocated Chunks

For certain malware, there is a technique to hide data in a hive by taking an unallocated chunk of the space and changing the metadata to make it an allocated chunk. This in effect allows the chunk of space from being reused by the registry, however it is difficult to find these chunks and identify them as

'unlinked' to the hive tree. The older versions of **yaru** had an option to scan for these unlinked allocated chunks via the menu entry and the resulting output would show the offset and size of the chunk.

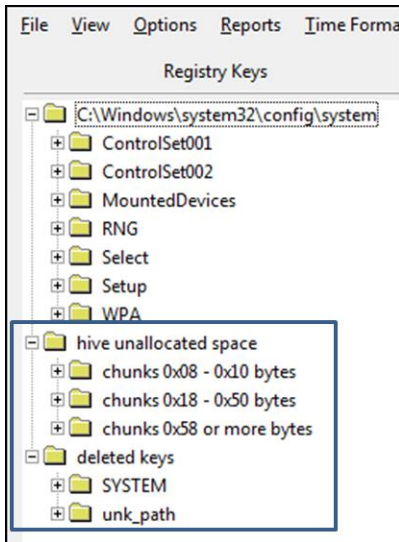
Starting with version 1.39, this menu option is deprecated and any unlinked 'allocated' chunks found during the initial load get reported as part of the hive tree. Since having unlinked 'allocated' chunks is *not* a normal occurrence, we needed to create a contrived example to show how **yaru** reports these artifacts. For this example, we took one of our hives and created various blocks of different sizes and then we just simply unlinked them by deleting all references to them. **yaru** easily finds them and reports them as follows:.



6 Deleted Registry Keys

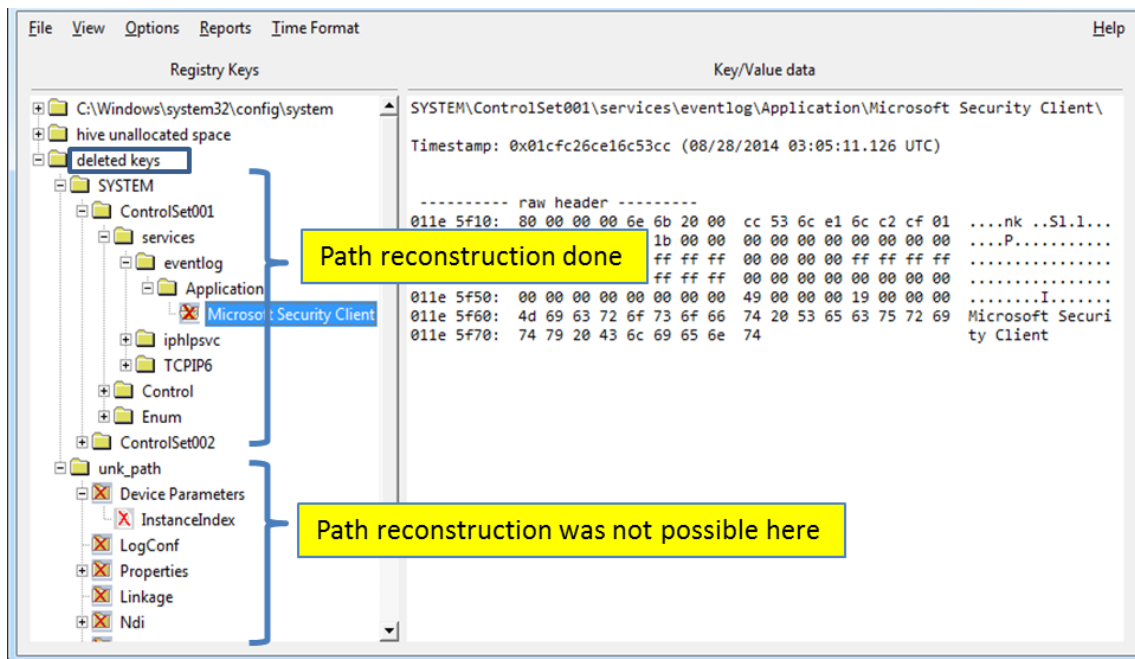
yaru only pulls out deleted key names and any data associated with those keys as opposed to blindly pulling out all signed components (eg. deleted values, security keys, etc) without having an association to a parent key name.

After selecting which file/hive to analyze, **yaru** traverses the hive for both allocated and unallocated space. Of the allocated space identified, **yaru** reconstructs the registry hive within a tree view structure similar to how Microsoft's *regedit* displays a registry hive. For the unallocated space identified, **yaru** categorizes each of the chunks into one of three bins: (i) chunks between 0x08 and 0x10 bytes, (ii) chunks between 0x18 and 0x50 bytes and (iii) chunks greater than 0x50 bytes. One can view each unallocated chunk in the form of a hex dump by selecting the desired chunk. The latter bin is the most important for carving out deleted keys, since registry key chunks require at least 0x50 bytes of space to store the common key header information (more if there is a name for the key).



yaru traverses all the unallocated chunks of greater than 0x50 bytes and looks for the 'nk' magic signature which denotes the chunk may have contained a registry key prior to being unallocated. Of those keys determined to be possible deleted keys, a number of boundary condition tests are performed to minimize the number of false positives. Tests such as date range checking, size checking, and whether valid offsets specified in the header are conducted. If the boundary checks are passed, **yaru** then proceeds to see if it can enumerate any values for the deleted key as well as try to locate the parent key. If a parent key is found, **yaru** recursively traverses up the parent hierarchy to find the entire path up to the root.

Once completed, **yaru** outputs the resulting deleted keys in the form of a tree view. If it was possible for **yaru** to reconstruct the parent hierarchy from a deleted key, then the hierarchy is shown for that key as part of the tree. To visually delineate between deleted and undeleted keys, a **red x** is overlaid over the folder or file icon for deleted keys/values. For those keys where the parent could not be determined, they are collected in a catchall tree node titled "**unk_path**".

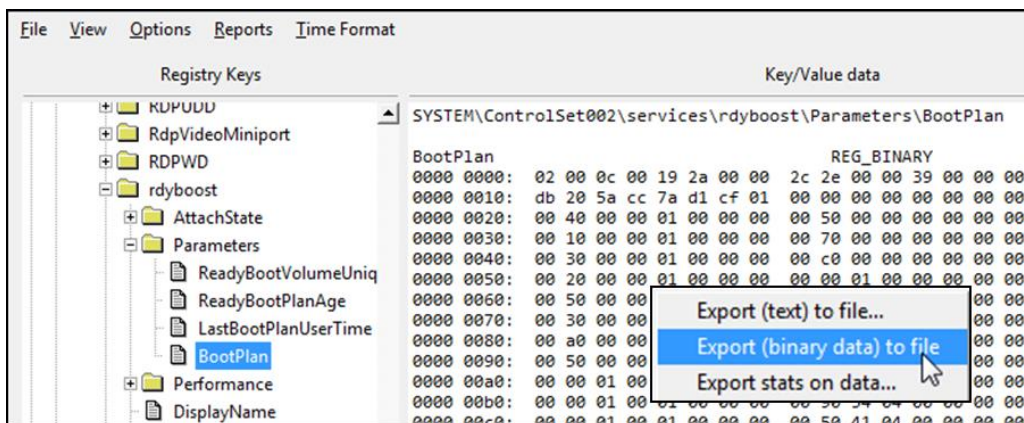


On the left pane is an expand view of one section of the deleted keys. Notice there are combinations of folder icons that do not have a *red x* with those overlaid with a *red x*. This representation was meant to help show the context of where a deleted key might have been deleted from. Also keep in mind, **yaru** generates these results from a deterministic, best guess standpoint. Thus, for example, if the 'nk' signature was deleted from the chunk, the key will not be found using this algorithm. On the right pane, the details about a selected deleted key along with the relevant hex dump of the key header are shown.

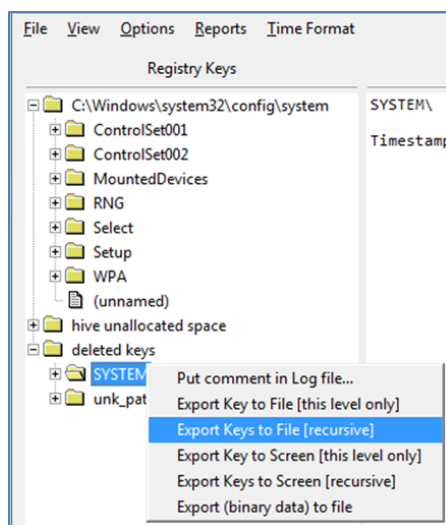
7 Exporting Keys and Data

Occasionally one will need to pull information from the key or value for offline analysis. There are various modes one can pull data from yaru. The two main ones are: (a) extracting the binary data and (b) extracting the subkey hierarchy in a useable format.

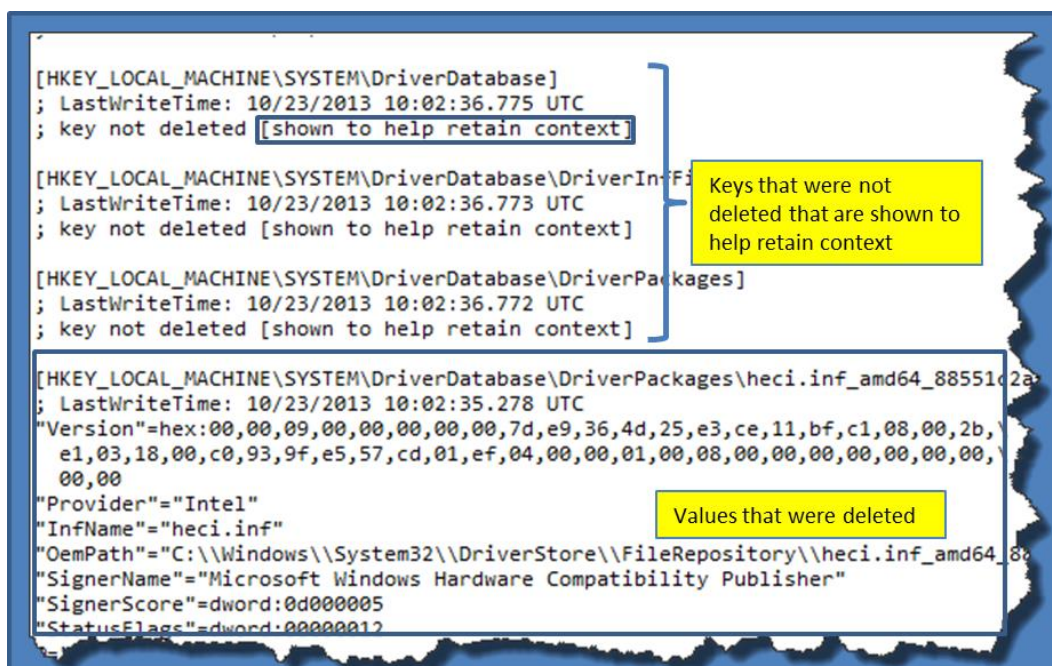
If needing to extract the raw data, one just navigates to the subkey/value that is of interest and then right clicks on the data window, whereupon a pop-up menu will allow on to export the data as text or binary to a desired file. Below is an example of exporting the BootPlan from the ReadyBoost services key. The BootPlan data can get very large and thus to analyze this data, it would be best to export the data and view it in your favorite hex editor.



If one needs to recursively extract the subkey's and values starting with a parent and including all the children, the option to do this is available by right clicking on the parent subkey in the tree view and selecting the various *Export Keys* options. In this example, we are pulling all the subkeys associated with the "deleted keys" and exporting them to a file. The format will use the standard *Windows Registry Editor Version 5.00* format. The file just needs to be renamed with a *.reg* extension, if one wishes to import the same data back into a registry. This, by the way, is not recommended unless you make a backup of your original registry hive. Anytime you add entries in this way can cause your registry to become unstable and hence the reason we put a *.txt* extension on the exported file.



When viewing a portion of the output, one will see both keys that were not deleted as well as keys and values that were deleted. This is purposely done so that the deleted keys/values have some context when viewing the hierarchy and their parent timestamps. Also it is useful when recreating the portion of the hive from scratch when renaming the file to *.reg*. Below is a sample output.



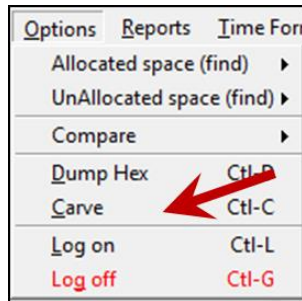
8 Brute Force Extraction of Keys – Carving

Included with version 1.45 of **yaru** is the added ability to carve out keys and values from hives that are only partial. This option is considered prototype. The key extraction is comprehensive in the sense that it will carve out keys that are valid, deleted or in slack space. The value extraction is more limited in the sense it will truncated long runs of binary data.

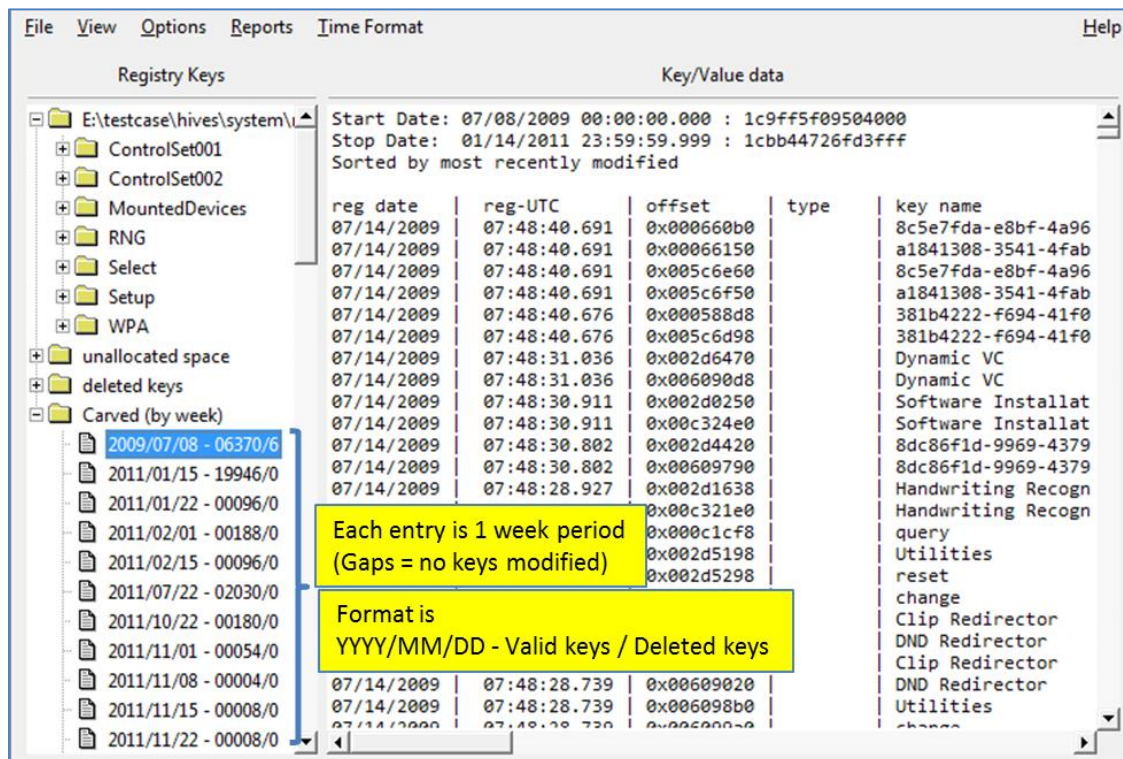
The purpose of adding in this functionality was specifically to pull artifacts from the registry transactional logs; however, it can be used to carve any hive. As background, the registry transactional logs have the same name as the hive counterparts, but have the extension **logX** where the **X** is a number of the log, since there can be more than one. The transactional logs have a valid hive header, but only have a small subset of the hive data; essentially the required data that was used to handle a registry transaction.

In the past, if you tried to open a transactional log with **yaru** it would have trouble parsing it. With this version it will automatically detect whether it is a transactional log and revert to carving the keys and based on the available data in the log, try to reconstruct each of the key's paths.

If you open a normal hive, then the behavior is the same as it was in the past. The registry will be parsed; any slack or unallocated space identified as well as any deleted keys. If one wishes to also carve the keys from this hive one can explicitly invoke it via the menu, under **Options -> Carve**.



After a carving operation, the tree-view pane on the left will create a root entry called “Carved (by week)” and with child entries that form a sort of histogram. Each of the child entries will group keys by date in increments of 1 week per entry. Gaps in time between entries imply there were no keys modified during that week in time. Clicking on any of the entries will list the keys for that time period.



The entries also show after each of the annotated dates, the number of valid keys and the number of deleted keys found during that period. This is useful for a quick triage to see what period of time keys were deleted. Once a time period is selected the keys modified during that time period are displayed on the right window. Not shown in the screenshot above, but if one scrolls to the right, if the full path was able to be constructed, it is displayed as well.

If we pick an entry with many deleted entries and truncate some of the output, to show how the one can use this technique to locate some critical USB entries that were deleted, it would look something like the screenshot below. The entry shows that during the week of 22 July 2013, 29358 keys were modified and 560 keys were deleted.

2013/06/08 - 00014/8	07/30/2013	01:25:15.414	0x0154f8d0	del	Device Parameters	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/06/22 - 00022/5	07/30/2013	01:25:15.414	0x0154fb20	del	LogConf	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/07/01 - 00044/2	07/30/2013	01:25:15.414	0x0154fc00	del	{540b947e-8b40-4291-b961-000000000000}	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/07/08 - 00042/21	07/30/2013	01:25:15.414	0x0154fc78	del	00000004	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/07/15 - 00042/12	07/30/2013	01:25:15.414	0x01559968	del	LogConf	SYSTEM\ControlSet001\Enum\USB\VID_0E0F&PID_0001\00D0C9CCD8
2013/07/22 - 29358/560	07/30/2013	01:25:15.414	0x0155cd08	del	00000000	SYSTEM\ControlSet001\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/08/01 - 00484/49	07/30/2013	01:25:15.414	0x0155ed00	del	00000000	SYSTEM\ControlSet001\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/08/08 - 00004/0	07/30/2013	01:25:15.414	0x015b6360	del	Device Parameters	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\001CC0EC35
2013/08/15 - 00002/0	07/30/2013	01:25:15.414	0x015b8168	del	00000000	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
2013/08/22 - 00000/19	07/30/2013	01:25:15.414	0x015b81c0	del	{83da6326-97a6-4291-b961-000000000000}	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
	07/30/2013	01:25:15.414	0x015b8e10	del	00000003	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
	07/30/2013	01:25:15.414	0x015b8e68	del	00000000	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
	07/30/2013	01:25:15.414	0x015b8f50	del	00000004	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3
	07/30/2013	01:25:15.414	0x015b8fa8	del	00000000	SYSTEM\ControlSet002\Enum\USB\VID_0E0F&PID_0001\1C659DAEE3

For similar functionality, but with much more flexibility in output options, one can use the *cafae* tool. It also has the carving functionality but can be scripted and the output easily sent to a post processor or database.

9 Validation of Parsed Residuals

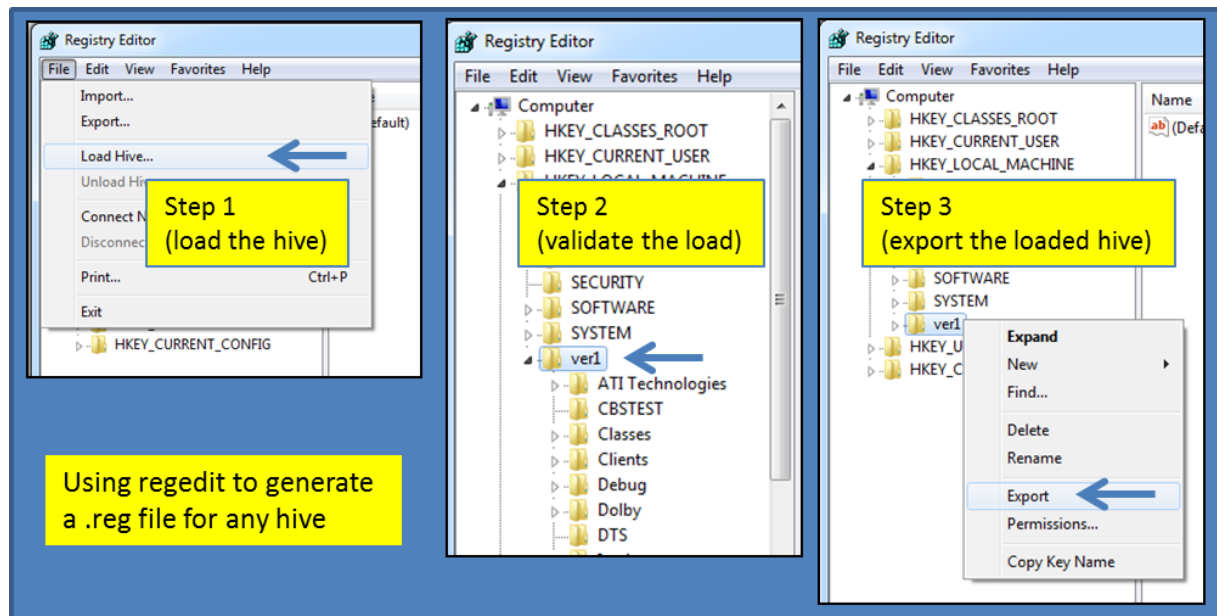
For any tool to be used in forensics, one must ensure the output generated is representative of the true output of the underlying data. All tools that extract data will ultimately format the data from some internal representation into a user readable form. This requires the tool to (a) parse the data accurately and (b) ensure that the data presented to the user is formatted correctly to minimize any misinterpretation of the output generated. When dealing with the Windows registry hives, this is no small feat. There are numerous boundary conditions that need to be taken into account. If one did this type of validation manually it would be close to impossible to compare entries in some of the larger hives, such as the software hive which can easily be larger than 25 MB in size.

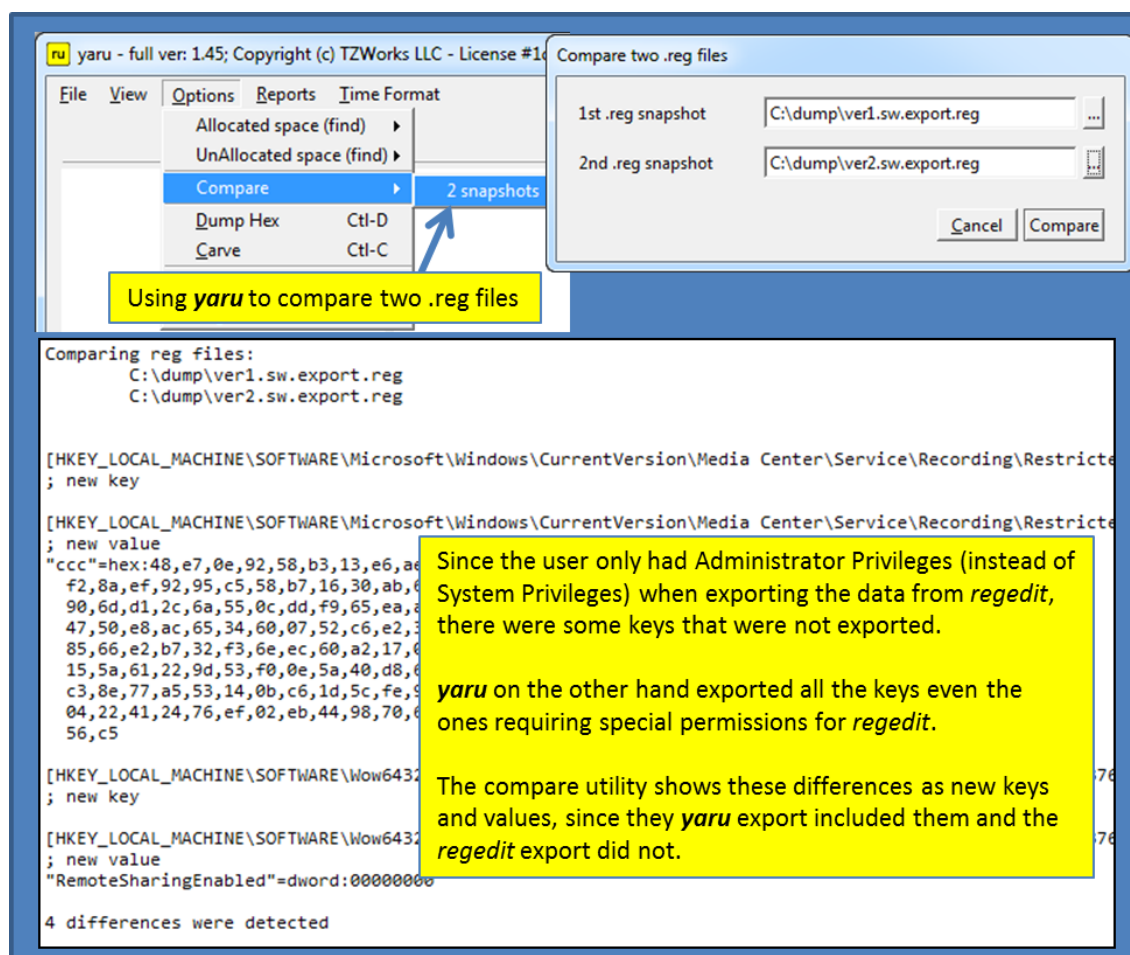
To automate this verification process as much as possible, *yaru* has the capability to output its data in the Microsoft registration file format (.reg format). This is done on a best effort basis and as problems in the output are encountered, bug fixes are applied. Nonetheless, the .reg format offers a way to test the output of *yaru* to that of the Microsoft *regedit* tool. Consider the simple scenario of making a copy of a hive and then importing that hive into the Microsoft *regedit* utility for the sole purpose of exporting the hive data into a .reg file. Repeating this process with *yaru* gives the user two representations of .reg files of the same hive generated by two different parsers. The beauty of this approach is it will validate not only the keys and value names but the underlying data as well. To get a complete list of the key/values in the *regedit* tool, one must have system level permissions. For Windows XP, this is as simple as using the 'at' command to spawn a command prompt and then invoking *regedit* from the newly spawned command prompt.

Once two .reg files are generated from different parsing tools, one needs a tool to compare the files easily. Simple differencing of the files using one's favorite differencing tool will *not* work as expected. There are a number of reasons for this: (a) the order of the data in the .reg files cannot be guaranteed to be the same, (b) the naming convention is affected when importing a hive into *regedit*, since it takes a new unique name which gets imprinted on the resulting data in the .reg file generated, and (c) miscellaneous artifacts that are added by one registry parser are not necessarily accounted for in another registry parser. To help with some of the issues, *yaru* incorporates an option that can take two .reg files, parse each of them, reorder the keys so they are suitable for comparison, remove any

commented fields and display the differences. One caution to keep in mind is that the .reg file uses Unicode as the native file format. Therefore if manually editing a .reg file, do not resave it into an 8 bit ASCII format due to the risk of losing data.

Below is an output of analyzing two .reg files from the software hive on a Windows 7 box. One of the .reg files was created with the Microsoft *regedit* utility and the other was created with *yaru*. For this example, the user only logged on to *regedit* with Administrative privileges (as opposed to System privileges) so some of the keys/values will not be accessible from the *regedit* tool. These differences will be clearly shown in the comparison.

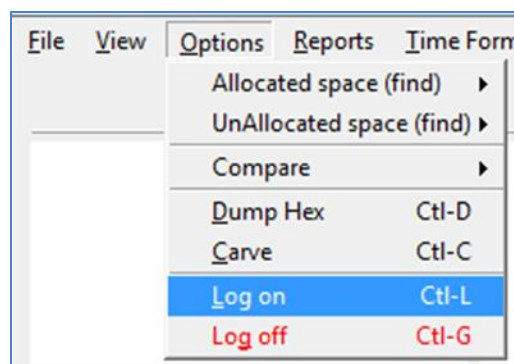




While this technique is great for validating the results of **yaru**, it can also be used for comparing a before and after hive to see the new and deleted keys/values. The only pre-requisite is one must export the keys/values from the same parent key to do a valid comparison.

10 Logging of Activities

To review one's past steps, it is useful to be able to refer to a log file that records all the steps one took during an analysis of a hive. This is also useful for debugging purposes when discovering some new registry key or trying to analyze some new registry format. For this reason, **yaru** incorporates a logging capability. To minimize the cluttering of log files, **yaru** starts off with logging turned off by default. However, when turned on, **yaru** will record all the users' activity, including what selections were made. To keep the log file manageable in size, any output that is sent to a file is not logged.



To make the format as extensible as possible, **yaru** incorporates XML as the file format. The date and time of creation is appended to the log file name to ensure uniqueness. Each log entry is also time stamped. Unfortunately there are no configuration settings to identify where the log file is archived or under what conditions to log data. For now, **yaru** generates a log file in the directory that **yaru** starts in. When logging is stopped, an XSL file will be created that will allow the resulting log file to be rendered in any web based browser. Adhoc comments can be injected into the log at any time by right clicking and selecting "Put comment in Log File".

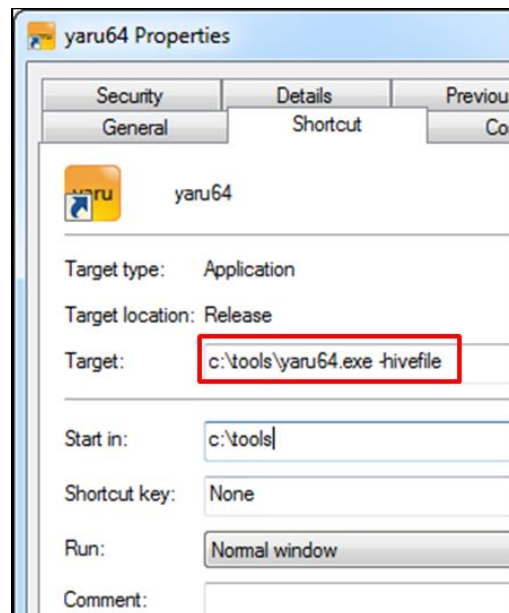
To view the log file when finished and if the logging is turned off, just open the XML file that was created in your favorite browser. Below is an example of the XML output rendered in Internet Explorer. The log includes timestamps and what action transpired. Any comments added are included as well.

Date	Time	Data
09/17/2014	19:23:09	init: yaru - full ver: 1.36; Copyright (c) TZWorks LLC - License #1cf9e36ea6a
09/17/2014	19:23:09	turning log on:
09/17/2014	19:23:15	expanded: ControlSet001
09/17/2014	19:23:21	expanded: services
09/17/2014	19:23:40	expanded: Tcpip
09/17/2014	19:23:48	expanded: Parameters
09/17/2014	19:23:59	examining: Parameters
09/17/2014	19:23:59	SYSTEM\ControlSet001\services\Tcpip\Parameters
		Timestamp: 0x01cfd2a3ef185c36 (09/17/2014 18:19:34.988 UTC)
		owner sid [S-1-5-32-544]
		group sid [S-1-5-32-544]
		Discretionary Access Control List
		access allowed S-1-3-4 READ_CONTROL
		access allowed Local System DELETE READ_CONTROL WRITE_DAC WRITE_OWN
		access allowed Local Service READ_CONTROL
		access allowed Network Service DELETE READ_CONTROL
		access allowed Admins DELETE READ_CONTROL WRITE_DAC WRITE_OWN
		access allowed Users READ_CONTROL
		access allowed Network Configuration Operators DELETE READ_CONTROL
		access allowed S-1-5-80-2940520708-3855866260-481812779-327648279-1710889582
		access allowed S-1-5-80-3081856537-581775623-1136376035-2066872258-400572886

11 Creating a “Send To” Shortcut for *yaru*

A useful shortcut to use *yaru* in a fast seamless way is to create a “Send To” option. This allows ones to right click on any hive in Windows, from the Explorer menu and open the hive in *yaru*. For a typical Windows 7 system, one would create a normal *yaru* shortcut in the following directory:

C:\Users\[desired user acct]\AppData\Roaming\Microsoft\Windows\SendTo. After this is done, edit the properties of the shortcut target to include the option *-hivefile*. This option is required for *yaru* to pull the hive you selected.



12 Command Line Options

When running in Windows, *yaru* cannot output to the console, but one can redirect the standard output (*stdout*) to a file. This is not a limitation with Linux or Mac. One can use this approach when using commands that do not invoke the GUI.

Commands to use with GUI [opens the GUI with the hive specified]

-hivefile <filename>

-ntfsimage <unmounted partition> <path\file of the hive>

Commands that do not invoke the GUI

-cmdfile <filename> = run *yaru* from a cmdfile with a list of !cmds

-cmd <options> = run a command using the *yaru* registry engine.

13 User Defined Templates

These are text files that allow one to automate key/value extraction. The parsing rules for these templates are discussed in more detail in the *cafae* user's guide. The *cafae* user's guide can be downloaded from this URL: <https://tzworks.com/prototypes/cafae/cafae.users.guide.pdf>.

14 Known Issues

1. When running under Vista or Windows 7, any network shares established prior as a regular (non-admin) user, will be isolated from other accounts (including the admin account). This problem occurs because User Account Control (UAC) treats members of the Administrators group as standard users. Therefore, network shares that are mapped by logon scripts are shared with the standard user access token instead of with the full administrator access token.
2. *yaru* may run out of memory processing some very large registry hives with many deleted files. To address this issue, use the 64-bit version of *yaru*.
3. When using *yaru* to compare *.reg* files from two different snapshots in time where the snapshots are generated from tools other than *yaru* (eg. from regedit.exe) one needs to ensure the *.reg* file is saved in the old NT4 format (which is text based) versus the default format (which is binary based). *yaru*'s comparison option only works with text based *.reg* files.

15 X-Window Dependencies

For this tool to work, the X Window System libraries are required for both Linux and macOS (they are not required for Windows). These libraries use the X11 protocol and graphics primitives to render the graphical user interface components. These libraries are common on Unix-like OS's.

If one is unfamiliar with X Windows or the libraries associated with it, one can download an installer package from XQuartz.org, which is an open-source effort to develop a version of the X Windows System that runs on Linux and macOS.

After the X11 libraries are installed, one needs to ensure they are running prior to running this tool.

16 Authentication and the License File

This tool has authentication built into the binary. The primary authentication mechanism is the digital X509 code signing certificate embedded into the binary (Windows and macOS).

The other mechanism is the runtime authentication, which applies to all the versions of the tools (Windows, Linux and macOS). The runtime authentication validates that the tool has a valid license. The license needs to be in the same directory of the tool for it to authenticate. Furthermore, any modification to the license, either to its name or contents, will invalidate the license.

16.1 *Limited versus Demo versus Full* in the tool's Output Banner

The tools from *TZWorks* will output header information about the tool's version and whether it is running in *limited*, *demo* or *full* mode. This is directly related to what version of a license the tool authenticates with. The *limited* and *demo* keywords indicates some functionality of the tool is not available, and the *full* keyword indicates all the functionality is available. The lacking functionality in the *limited* or *demo* versions may mean one or all of the following: (a) certain options may not be available, (b) certain data may not be outputted in the parsed results, and (c) the license has a finite lifetime before expiring.

17 References

- 1 Document on various Internet sites titled "WinReg.txt" by B.D.
- 2 Various articles in MSDN.
- 3 Windows Forensic Analysis DVD Toolkit, Harlan Carvey
- 4 Wikipedia, the free encyclopedia section on [Windows Registry](#).
- 5 Various forensic artifacts discussed in Computer Forensic Essentials from [SANS Institute](#).
- 6 Forensic Analysis of Unallocated Space in Windows Registry Hive Files, by Jolanta Thomassen, Dissertation for Master of Science submitted to The University of Liverpool, dated 04 Nov 2008.
- 7 The Internal Structure of the Windows Registry, by Peter Norris, MSc Thesis submitted Defence College of Management and Technology, Dept of Informatics and Sensors, Cranfield University. Feb 2009.
- 8 Recovering Deleted Data from the Windows Registry, by Timothy D. Morgan, Digital Investigation 5 (2008) S33-S41.
- 9 [FOX-toolkit](#) version 1.6.47.
- 10 B.D. [WinReg.txt](#). <http://home.eunet.no/pnordahl/ntpasswd/WinReg.txt>, 1998.
- 11 Thomassen, Jolanta. [Forensic Analysis of Unallocated Space in Windows Registry Hive Files](#). Dissertation for Master of Science submitted to The University of Liverpool, 2008.
- 12 Norris, Peter. [The Internal Structure of the Windows Registry](#). Defence College of Management and Technology, Dept of Informatics and Sensors, Cranfield University, 2009.
- 13 [X Window System Libraries](#) by XQuartz.org.