Recovering and Analyzing Deleted Registry Files

Andrew Case
Digital Forensics Solutions
andrew@digdeeply.com / @attrc

I had a case recently where I was asked to investigate any signs of data exfiltration by a client's former employee. What I did not realize until I started investigating the case is that the computers in question were reformatted/reinstalled in-between the employee leaving and my investigation [1].

After realizing this, I then needed a plan to recover relevant data knowing that multiple operating systems worth of files would be spread throughout the drive, and that none of the files I was interested in would be present in the filesystem. My first goal of the investigation was to recover USB device activity, and my initial plan for this was to simply file carve for deleted registry files and then manually figure out the ones that belonged to the previous instance of the OS. I could not simply use *blkls* from the Sleuthkit, as I normally would target unallocated files, because I did not know the layout of the previous install of the operating system. This meant the previous registry files could literally be anywhere on the disk.

I knew from past cases that carving for registry files is usually painful because there is a header but no footer, so I needed some extra power besides just header/footer carving. With all this in mind, I came up with and executed the plan I will now explain in order.

Step 1: Use Scalpel to Find All Instances of Possible Registry Files

My first step was to find where all possible registry files could be. Registry files start with the string "regf", and so I created a simple Scalpel config file (see Appendix A – Scalpel configuration file for carving registry contents) that would find all instances of "regf" on the disk image (using Scalpel as an indexer this way was previously explained here). I then ran Scalpel with this invocation:

scalpel -c scalpel.conf -p -o regresults disk.dd

Note that the '-p' (preview) tells Scalpel to only write its audit file and not to actually carve any files out of the disk. This audit file contains all the offsets into the disk where the header ("regf") was found. Scalpel in this instance found a little over 7000 instances of "regf" on disk.

Step 2: Filter Scalpel hits into possible SYSTEM files.

After Step 1, I had an audit file with all the offsets on disk where "regf" appeared. I then wanted to filter those only to possible SYSTEM registry hives as this is where USB information is stored. For this I wrote a script, recover.py (see Appendix B – recover.py), that:

1) Gathered all the offsets from Scalpel's audit file

2) Read 2048 bytes starting at that offset into a separate buffer

3) Checked the separate buffer for “system”/”SYSTEM” in Unicode

4) Print out indexes that pass the check from 3)

I ran recover.py with this invocation:

python recover.py regresults/audit.txt disk.dd > systemidxs-unicode

This then placed all the indexes of possible SYSTEM hives into the *systemidxs-unicode* file, and narrowed down the original Scalpel indexes list from over 7000 to 171.

Step 3: Gathering the Possible Registry Files

The next step was to take the 171 indexes of possible SYSTEM files found in Step 2, and to determine which were actual registry hives. To accomplish this, I wrote grab.py (see Appendix C – grab.py), which simply takes a list of disk offsets and a disk image, and pulls X number of bytes starting at each index and writes the contents out to a file. To ensure that I recovered all data from the gathered SYSTEM files, I set this script to pull 25MB out for each file. I ran the invocation of grab.py with *systemidxs-unicode* from the previous step as:

python grab.py systemidxs-unicode disk.dd grabout/

This placed the possible hives into the "grabout" directory.

Step 4: Filtering Real Hives versus Fragments

I now had 171 possible SYSTEM files written out to the directory "grabout". At this point I needed to determine which hives were valid. To accomplish this, I used reglookup and had it attempt to parse every gathered registry hive. I saved the output of each attempt for the next step. Since *reglookup* only accepts one file at a time, I chose to write a bash one-liner that would automate the process over all of the gathered hives. That invocation looked like this:

for file in $(ls grabout); do reglookup grabout/$file > reglookup-output/$file-output; done > /dev/null

For those not familiar with bash, what this one-liner does is:

1) Gets a list of all the files in the "grabout" directory, this is the "ls grabout" invocation.
2) It then iterates over every file in the directory and runs reglookup on it
3) The output of each reglookup invocation is saved in the "reglookup-output" directory as "<filename>-output".

The result of this process is that each registry file will be processed in one command. If a file is not a valid registry hive then its output file will be of size 0, or, if it is valid, then all of its registry paths, names, and values will be written to the output file.

Step 5: Determining and Acquiring Hives of Interest

After Step 4 completed, I had output files that:

1) Determined if a file was a valid hive or not
2) Could be searched for relevant keys/names/values.

Since my first goal of this investigation was to determine USB activity from recovered hives, I then wrote another bash one-liner to find all the hives that had a USBSTOR key (which signifies that a USB device had been plugged in at some point while the hive was active). The bash invocation looked like this:

for name in $(grep -li USBSTOR *-output | cut -d '-' -f 1); do cp grabout/$name /media/KINGSTON/regfiles/; done

This invocation actually accomplishes two goals. The first is that it filters based on "USBSTOR" being present in a hive, and the second is that it copied each wanted hive to my USB thumb drive. It accomplishes this goal by:

1) Using "grep" to find all of the *reglookup* output files that have USBSTOR in them. Note that the "–l" flag to grep means to only print a matching filename once even if the search term appears multiple times.
2) For each matching output file, copy the actual registry hive, stored in "grabout", to the USB drive folder "/media/KINGSTON/regfiles/"

Step 6: Analyzing the Recovered Hives

After Step 5 completed, I then had 49 intact SYSTEM files that were recoverable from the disk image and that passed the previously discussed validation steps. Remembering that these files were spread out across (at least) two installs of Windows as well as possibly numerous System Restore Points across the installs, I needed a way to filter to just the files from the previous install.

To accomplish this, I copied my files to a Windows analysis machine and turned to Registry Decoder. First, I created a new Registry Decoder (RD) case and then loaded all 49 recovered hives into it. Once it was done processing, I then utilized Registry Decoder's search ability to quickly find USB devices of interest. Registry Decoder's search functions allow an investigator to filter by a number of selections, including:

1) A single search term or a newline separated set of search terms in a file
2) The last write time of keys
3) 'Exact' or wildcard search
4) Limiting searching to any combination of keys, names, and data

As part of our pre-investigation phase, the client had given us a date range to be investigated. I was able to filter my search results to these dates by using the "Filter By Last Write Time" fields of RD's search form. To recover USB activity, I targeted the *DeviceClasses* information for USB drives that were used on the machine. To filter to only USB devices, I entered a search term of "\\?\USBSTOR#Disk" and told RD to only search the 'data' field of the analyzed registry hives. Appendix D – Search Form Screenshot has a screenshot of what this setup looks like on a sample case. This approach effectively narrows down the search results to only USB drives found under the *DeviceInstances* path as well as within the appropriate date range.

I then ran this search across all the files loaded into the case, and a number of search result tabs were generated. By looking through the results I could tell that I had recovered a large number of files from the previous Windows install, and that many USB devices had been used on it during the timeframe of interest. I then ran Registry Decoder's USBSTOR plugin, modified to include the last write time of the keys, across all the registry files within my time range. I then cross-compared the timestamps in the USBSTOR keys to those in the associated *DeviceClasses* keys to ensure they matched. This is necessary as USBSTOR keys timestamps are not always reliable.

Finishing this part of the investigation was as simple as using Registry Decoder's automated report feature to write all the USB devices found, including their serial number and last insertion time, into a formatted HTML file.

Final Thoughts

Hopefully this write up was interesting as it documents a process that I do not know of any other references for (at least with only using open source tools). The end result was that I started off with a disk that no longer had my targeted OS installed, and was able to recover a large number of registry files as well as to quickly analyze and report their contents with Registry Decoder.

Note that while I targeted SYSTEM files in this example, recover.py could be modified to target any other type of hive and the process would be the same. I performed this process on the same case with NTUSER files and recovered over 180 valid hives. I then took filenames of interest from our client and used those as my *grep* parameter (in place of USBSTOR) to narrow down to the hives I wanted to analyze. I then loaded the files into Registry Decoder and performed searches specifically for the filenames our client provided.

I would like to thank Wyatt Roersma (@WyattRoersma), Ken Pryor (@kdpryor), and Frank McClain (@littlemac042) for reviewing an initial version of the paper and providing feedback.

Notes

[1] That the OS was reinstalled was determined by noticing strange dates in the "Documents and Settings" folder (XP machine) related to user accounts and then was verified by checking the install date in the Registry. The install date is stored in "\Microsoft\Windows NT\CurrentVersion" -> "InstallDate" of the SOFTWARE hive, and can be recovered automatically using the "Windows Install Information" plugin of Reigstry Decoder or the "winver" plugin of RegRipper.

Appendix A – Scalpel configuration file for carving registry contents

```
reg     y       20      regf
```

Appendix B – recover.py

```
import os,sys,re

# a search term found and its disk location
class scalpel_hit:
    def __init__(self, term, offset, length):
        self.term     = term
        self.offset   = offset
        self.length   = length
        self.filename = ""

# parses the matched headers and offsets from scalpel's audit.txt
# returns a list of 'scalpel_hit' objects (one for each match)
def parse_audit_log(auditfile):

    hits = []
    hitre = re.compile('^\d+\.(\S+)\s+(\d+)\s+\S+\s+(\d+)')

    for line in auditfile.readlines():

        match = re.search(hitre, line)

        if match:
            g = match.groups()
            hits.append(scalpel_hit(g[0], int(g[1]), int(g[2])))

    return hits

def main():

    audit = open(sys.argv[1], "r")
    img   = sys.argv[2]

    hits = parse_audit_log(audit)

    fd = open(img, "rb")

    find = "s\x00y\x00s\x00t\x00e\x00m"

    for offset in [hit.offset for hit in hits]:

        fd.seek(offset, 0)

        buf = fd.read(2048)

        if buf.lower().find(find) != -1:
            print "%d" % offset

if __name__ == "__main__":
    main()
```

Appendix C – grab.py

```
import sys, os

idxs = [int(x) for x in open(sys.argv[1], "r").readlines()]
fd = open(sys.argv[2], "rb")
outdir = sys.argv[3]

ctr = 0

for idx in idxs:

    fd.seek(idx, 0)

    buf = fd.read(25000000)

    outfd = open(os.path.join(outdir, ctr), "wb")
    outfd.write(buf)
    outfd.close()

    ctr = ctr + 1
```

Appendix D – Search Form Screenshot