Digital Forensics VII:
Windows Memory Analysis

Cameron Quitugua
cjq061000@utdallas.edu

Department of Computer Science
The University of Texas at Dallas

October 7th, 2011

Outline

1 Papers
2 Schuster's Tool
3 Examining History
4 Recovering Registry Keys
5 Performance
6 Summary

Some Kernel Information

1 Windows kernel maintains two heap-like pools
   - Paged Pool: contents may be swapped into a file if it should become necessary to reclaim physical memory
   - Non-Paged Pool: contains the most important and frequently accessed objects

2 Kernel memory can survive over 14 days

3 Kernel frees up memory for destructed objects

4 Kernel does not necessarily overwrite any part of an object after destruction!

5 However, current (as of 2005) kernel analysis tools walk the kernel like a tree, so they do not detect these destructed objects

Source: Schuster 2006

Outline

1 Papers
2 Schuster's Tool
3 Examining History
4 Recovering Registry Keys
5 Performance
6 Summary

Papers

June 2006 - Searching for processes and threads in Microsoft Windows memory dumps
Andreas Schuster

June 2007 - Forensic memory analysis: From stack and code to execution history
Ali Reza Arasteh, Mourad Debbabi

March 2005 - Digital forensics of the physical memory
Mariusz Burdach

May 2008 - Forensic analysis of Windows registry in memory
Brendan Dolan-Gavitt

May 2008 - The impact of Microsoft Windows pool allocation strategies on memory forensics
Andreas Schuster
Introducing PTfinder

- Advances through dump one step at a time, with each step equal to kernel's memory allocation granularity
- Follows a set of 20 rules in order to identify process and thread objects
- Always assumes it is looking at a valid process or thread structure in order to catch partially overwritten or concealed objects

Source: Schuster 2006

PTfinder vs. Other Tools at the DFRWS 2005 Memory Analysis Challenge

- Storyline: Unusual traffic was seen while monitoring a worker's computer. The incident response coordinator used Helix 1.6 to dump the physical memory of the machine. However, while attempting to create a forensic duplicate of the drive, the system rebooted unexpectedly. Once the system was back up, another physical memory dump was created.
  - kntlist reports that the system boot time of the first image was 2005-06-05 00:32:27Z
  - PTfinder, however, finds processes started before that:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Image Name</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-06-06</td>
<td>01:39:15Z</td>
<td>ciscorun</td>
<td>SB</td>
</tr>
<tr>
<td>2005-06-05</td>
<td>02:20:14Z</td>
<td>Valgrind</td>
<td>SB</td>
</tr>
</tbody>
</table>

PTfinder’s Graph Functionality

- PTfinder uses Graphviz to render graphs expressing relations between threads and their process owners

Source: Schuster 2006

Overall Analysis of PTfinder

- However, the author specifies that PTfinder is not meant to be a complete solution to the complex task of Windows memory analysis.
- Instead, PTfinder should be used as an improvement to other tools such as kntlist

Source: Schuster 2006

Stack Refresher

- The stack mechanism is used to make structured programming possible
- A stack frame is created and stored on the stack for every function call made by a process
- The stack frame contains parameters passed to a function, the return address, and the previous value of the EBP register and local variables of that function
- Tracing these function calls creates a history of what a process has done during execution

Source: Arasteh & Debbabi 2007
Stack Growth is Important!

1. After a function returns, the stack pointer moves down to point to the previous stack.
2. This leaves the returned function stack frame still in memory!
3. And it will stay there until the stack grows high enough to overwrite that frame.
4. This leaves a large number of previously called function stack frames on top of the stack, partially overwritten or even untouched.

Source: Arasteh & Debbabi 2007

Arasteh’s and Debbabi’s Stack Analysis Technique

1. Parse the internal memory structures
2. Retrieve the process assembly code and stack from memory
3. Construct a control flow graph (CFG) from the executable code
4. Model the program execution by transforming the CFGs to local automata and combining these models into a Push Down System (PDS)
5. Model the stack residues properties using process logic
6. Extract all possible execution paths by correlating the stack residues properties and the program execution model

Source: Arasteh & Debbabi 2007

Visual Representation of the Technique

Source: Arasteh & Debbabi 2007

Control Flow Graph

Source: Arasteh & Debbabi 2007

Sample Local Automata Model

Source: Arasteh & Debbabi 2007

Windows Thread Stacks During Kernel Execution

Source: Arasteh & Debbabi 2007
Windows Thread Stacks During Kernel Execution

1. Note that OLD_EBP points to the address of the previous frame’s OLD_EBP.
2. This chains stack frames together, allowing a parser to correctly identify each stack frame.
3. However, if EBP is used within the function, parsers lose this ability.
4. Instead, the authors identify stack frames by looking for return addresses that point to just after a call instruction. This allows retrieval of addresses of call instructions as well as the arguments passed to functions.

Source: Arasteh & Debbabi 2007

How the Registry is Stored

1. The registry is made up of a number of different binary files called hives.
2. Hives are broken down into fixed sized bins of 0x1000 bytes.
3. Bins contain variable-length cells which hold the actual registry data.
4. References in hive files are made by cell index, a value that can be used to derive the location of a cell containing the referenced data.
5. The registry data structure has two distinct data types: key nodes (directories) and value data (files).

Source: Dolan-Gavitt 2008

Two Ways of Locating the Hives

1. Scan physical disk for 0xbee0bee0 (the constant signature that indicates a hive). Then, use this hive’s HiveList to locate the other hives on the disk.
2. Note that while hives will be contiguous on the hard disk, they need not be this way in memory. To deal with this, Windows Configuration Manager creates a mapping between cell indexes and addresses in virtual memory. We can follow this mapping to take us to every hive in the memory.

Source: Dolan-Gavitt 2008

Table 1 - Number of Keys/Values accountable using various techniques

<table>
<thead>
<tr>
<th>Metric</th>
<th>Standard Technique</th>
<th>Shell Hash</th>
<th>Shell DSA</th>
<th>Standard DSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keys</td>
<td>3507</td>
<td>3283</td>
<td>3281</td>
<td>3280</td>
</tr>
<tr>
<td>Keys (indexed)</td>
<td>3286</td>
<td>3285</td>
<td>3284</td>
<td>3284</td>
</tr>
<tr>
<td>Keys (non-indexed)</td>
<td>3287</td>
<td>3287</td>
<td>3286</td>
<td>3286</td>
</tr>
<tr>
<td>Values (indexed)</td>
<td>3285</td>
<td>3285</td>
<td>3284</td>
<td>3284</td>
</tr>
<tr>
<td>Values (non-indexed)</td>
<td>3282</td>
<td>3282</td>
<td>3281</td>
<td>3281</td>
</tr>
<tr>
<td>Values (nested)</td>
<td>3283</td>
<td>3284</td>
<td>3283</td>
<td>3283</td>
</tr>
<tr>
<td>Values (flat)</td>
<td>3284</td>
<td>3284</td>
<td>3284</td>
<td>3284</td>
</tr>
<tr>
<td>Values (hash)</td>
<td>3285</td>
<td>3286</td>
<td>3285</td>
<td>3285</td>
</tr>
<tr>
<td>Values (flat)</td>
<td>3286</td>
<td>3286</td>
<td>3286</td>
<td>3286</td>
</tr>
<tr>
<td>Values (hash)</td>
<td>3287</td>
<td>3288</td>
<td>3287</td>
<td>3287</td>
</tr>
</tbody>
</table>

Source: Dolan-Gavitt 2008
Further Examination of the Results

1. The technique of walking the hive starting at the root in memory is more effective than any other existing method of extracting registry keys and values from RAM.
2. Running these techniques on a "real" workstation might result in substantially reduced effectiveness. This is because normal use would cause unused portions of the registry to be paged out from memory to disk.
3. The author notes that an average of 631 keys and 1231 values per image were found in memory only. Another observation is that a small number of these keys had a more recent timestamp in memory than on disk, indicating that a write had occurred that had not yet been recorded on the disk.

Source: Dolan-Gavitt 2008

Review of Dolan-Gavitt's Work

1. In short, registry data in memory shares the same structure as registry data on the hard disk, but uses a different address translation mechanism. Dolan-Gavitt’s work shows how to adapt to this and gain full access to registry data stored only in memory. This is of vital importance, as there are attacks which cannot be detected in any other manner beyond forensic analysis of volatile registry keys and values.

Source: Dolan-Gavitt 2008

Can We Use Our Toolkit?

1. How long can a process structure persist in system pool memory after the process has been terminated?
2. What mechanisms affect the persistence of freed allocations in pool memory?
3. What are the implications on forensic memory acquisition and incident response tools?

Source: Schuster 2008

What Makes Volatile Data Important?

1. Consider the following scenarios:
   - Attacker gains access to kernel memory, altering cached keys and leaving on-disk data unchanged. Operating System will be using the cached keys, so if we only examine the on-disk data, we completely miss the attacker.
   - Tools such as the SQL Slammer worm reside only in memory and do their damage without writing to disk.
   - Thus, being able to analyze volatile memory is vital to a forensic investigation.

Source: Schuster 2008 & Burdach 2005

How Long Does Volatile Data Stick Around?

1. Farmer & Venema (2005): “after 10 minutes, about 90% of the monitored memory was changed”
2. Chow et al. (2005): after 14 days of “everyday work,” 3 MB of an initial 4 marked MB were still available
3. Walters & Petroni (2007): 15 hours of idle activity on a Windows XP Service Pack 2 system left about 85% of 512 MB RAM unchanged
4. Solomon et al. (2007): pages of userland data persist for less than 5 minutes with only single pages lasting longer

Source: Schuster 2008
### Schuster’s Experiments

1. Environment: avoids any unneeded background activity in order to provide an unobstructed, microscopic view of activities associated with allocation/deallocation of pool memory.
2. Persistence of objects in nonpaged pool: 100 processes were created and then terminated in reverse order. Discovered that once the nonpaged pool freed these processes, a significant portion could persist for 24 hours or longer. However, Schuster notes that background activity such as local security authority or svchost.exe would overwrite these processes over time.

Source: Schuster 2008

### Conclusions from the Experiments

1. System pool memory can persist for a long time, probably for as long as the system process is running.
2. The userland portion of the memory, however, will not last this long.
3. Incident response toolkits must be extremely careful to check for resource consumption prior to usage in the field; if not, they could potentially overwrite evidence.
4. Forensic software must use as little of the system resources as possible (again, to prevent overwriting).
5. One solution for large companies would be to force a pre-allocation of system resources (e.g. create threads) on system boot. This will allow for tools to overwrite these threads without affecting the investigation.

Source: Schuster 2008

### Review

1. Schuster introduced PTfinder, a tool that finds processes and threads stored in memory.
2. Arasteh & Debbabi show how to analyze thread stacks in order to figure out what a thread has actually done during its lifetime.
3. Dolan-Gavitt introduced several techniques that allow for recovery of volatile registry data.

Source: Schuster 2008

### Outline

- Papers
- Schuster’s Tool
- Examining History
- Recovering Registry Keys
- Performance
- Summary