Process Dump Analyses

Forensical acquisition and analyses of volatile data

Tobias Klein
tk@trapkit.de

Version 1.0, 2006/07/22.
1 Overview

There is a general lack of techniques and tools today which can be used to assist the acquisition as well as the analyses of volatile data of a live system. This paper discusses some new techniques and tools that can be used to acquire and analyse process dumps of Microsoft Windows and Linux operating systems in a forensical manner. As an example the introduced tools are used to analyse and reconstruct (remote) code injection attacks that are using anti forensic techniques in order to circumvent classical post-mortem analyses.

1.1 Advanced (Remote) Code Injection Attacks

To clarify the term “advanced (remote) code injection attack” I will give a short example: A remote attacker knows about a Memory Corruption Vulnerability (Buffer Overflow, Format String Vulnerabilities, etc.) within an offered service (e.g. Apache, IIS, etc.). He exploits this vulnerability in order to inject and execute malicious code in the context of that vulnerable service. The injected code first ensures that protective measures (such as Firewalls, Reverse Proxies, etc.) are successfully circumvented using techniques like “connection reusing” and “protocol encapsulation”. Subsequently, the code offers different possibilities to the attacker for further control of the compromised system. All these steps are performed in the process memory of the exploited service. There is no interaction with the filesystem or the harddisk at all (anti forensic technique).

1.2 How does Computer Forensic work today?

If an incident is identified the system is usually switched off. Then the data media (normally the harddisk) is duplicated and secured for later analyses (computer forensic).

This procedure has several weak points especially if the incident involved a (remote) code injection attack. The first problem is that it’s very difficult to identify such an attack as an incident. In addition, even if the incident is identified it cannot be reconstructed by analysing the data media, since all hints and traces are only contained in the memory of the exploited process.

In the following paragraphs a partial discipline of computer forensic called live analysis will be described. This technique can be used to identify and analyse advanced attack techniques like (remote) code injection.

2 Process Dump Analyses

Before we can start with the actual analysis it is necessary to dump the memory of running processes in a forensical manner. There are some freely available tools which are able to take such a snapshot of a running process (see [1] and [2]). However, these tools have some disadvantages. On the one hand, the tools usually write the dumps to the harddisk so the content of the data medium gets corrupted. On the other hand the process memory gets dumped as a non-coherent data blob, which makes a meaningful analysis practically impossible. To solve these issues I developed a new tool called Process Dumper (pd) (see [3]). Process Dumper doesn’t touch the harddisk at all and can be combined with other tools such as Netcat [4], in order to transmit the collected data over the network. Beyond that Process Dumper preserves not only the data of the process memory, but also the associated metadata to support the later analysis. To analyse the process dumps I’ve developed a second tool called Memory Parser (MMP) (see [5]).
2.1 Process Memory Layout

The process memory is usually divided into several different sections or mappings. There are two kinds of mappings: data and code mappings. Figure 1 shows a meta view of a process memory layout.

![Process Memory Layout Diagram]

2.2 Process Dumper

This tool can be used to take a snapshot of the memory of a running process. The tool is currently available for Linux and Microsoft Windows.

Features of version 1.1:

- Dumps the whole process space (all data and code mappings)
- Uses meta information to describe the different mappings (needed for advanced analysis)
- Saves the process environment and state
- Outputs to stdout, so it’s possible to combine it with other tools (netcat etc.)
- Doesn’t touch the harddisk at all

Usage

C:\>pd

Usage: pd [-v] -p pid

Options:
- -v - be verbose

Examples:

```
  pd -p pid > pid.dump
  pd -p pid | nc 10.0.0.1 7000
```

Example

This example shows how to dump the lsass process on a Windows XP SP2 system.

First it is needed to get the process ID (PID) of lsass:
The output of `tasklist` shows that the PID of `lsass` is 1008. The next command will dump the process memory of `lsass` to the harddisk:

```c:
C:\> pd -p 1008 > lsass.dump
Dump finished.
```

To transfer the process memory to a remote computer without touching the local harddisk(s) it is possible to pipe the output of Process Dumper to a tool like netcat.

The following command can be used to transfer the dump to a remote computer:

```c:
C:\> pd -p 1008 | nc 192.168.0.100 7000
Dump finished.
```

The command line option “-v” enables verbose output:

```c:
C:\> pd -v -p 1008 > lsass.dump
Mapping: 0x00010000-0x00011000 Size: 4096
Mapping: 0x00011000-0x00020000 Size: 61440 -> not dumped!
Mapping: 0x00020000-0x00021000 Size: 4096
Mapping: 0x00021000-0x00030000 Size: 61440 -> not dumped!
Mapping: 0x00030000-0x00031000 Size: 4096
[..]
```

## 2.3 Memory Parser

The tool Memory Parser (MMP) can be used to analyse the process dumps.

Features of version 0.2:

- Parsing of process dumps made with Process Dumper v1.1
- Interpreting the meta data of process dumps made with Process Dumper v1.1
- Hash checking of the code sections of the mapped executables (DLLs etc.)
- Concatenate all data mappings to one reference data mapping
- RSA certificate and key finder (see [6])
- Flexible configuration via XML

### 2.3.1 Memory Parser: Overview

To open a process dump press the “Open Process Dump” button (see Figure 2).

![Figure 2: Open a process dump](image-url)
To parse the process dump press the “Parse Process Dump” button (see Figure 3).

![Figure 3: Parse a process dump](image)

### 2.3.1.1 The Mappings

After Memory Parser has finished parsing you can find the different mappings listed in the upper list view (see Figure 4).

![Figure 4: Process dump mappings](image)

The information about every mapping is displayed in the following rows:

<table>
<thead>
<tr>
<th>Row name</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the mapping. All mappings that contain executable code are prefixed by the string “map-“ while data mappings start with “mem-“ The individual mappings can be found as separate files in the same directory as the process dump itself.</td>
</tr>
<tr>
<td>Type</td>
<td>Data or Code</td>
</tr>
<tr>
<td>Mapping Start</td>
<td>The start address of the mapping within the virtual address space of the dumped process.</td>
</tr>
<tr>
<td>Mapping End</td>
<td>The end address of the mapping within the virtual address space of the dumped process.</td>
</tr>
<tr>
<td>Mapping Size</td>
<td>The size of the mapping.</td>
</tr>
<tr>
<td>Mapped From</td>
<td>If it is a code mapping this row contains the path of the mapped binary image. If it is a data mapping this row remains empty.</td>
</tr>
<tr>
<td>Image Base</td>
<td>If it is a code mapping this row contains the start address of the mapped binary image within the virtual address space of the dumped process. If it is a data mapping this row remains empty.</td>
</tr>
<tr>
<td>Image Size</td>
<td>If it is a code mapping this row contains the size of the mapped binary image. If it is a data mapping this row remains empty.</td>
</tr>
<tr>
<td>Image Start</td>
<td>If it is a code mapping this row contains the start address of the mapped binary image.</td>
</tr>
</tbody>
</table>
image within the virtual address space of the dumped process. If it is a data mapping this row remains empty.

<table>
<thead>
<tr>
<th>Image End</th>
<th>If it is a code mapping this row contains the end address of the mapped binary image within the virtual address space of the dumped process. If it is a data mapping this row remains empty.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td>This row is used for further descriptions of the mapping.</td>
</tr>
</tbody>
</table>

Table 1: Mapping information

To analyse a specific mapping right-click the appropriate row and choose one of the analyse tools (Hint: In the default configuration only the notepad utility will show up. See Section 2.3.1.3 Configuration for an example of how to configure Memory Parser).

<table>
<thead>
<tr>
<th>Mappings:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>mem-051.dmp</td>
</tr>
<tr>
<td>mem-052.dmp</td>
</tr>
<tr>
<td>mem-053.dmp</td>
</tr>
<tr>
<td>mem-054.dmp</td>
</tr>
<tr>
<td>mem-055.dmp</td>
</tr>
<tr>
<td>mem-056.dmp</td>
</tr>
<tr>
<td>map-057.dmp</td>
</tr>
<tr>
<td>map-058.dmp</td>
</tr>
<tr>
<td>map-059.dmp</td>
</tr>
<tr>
<td>map-060.dmp</td>
</tr>
<tr>
<td>map-061.dmp</td>
</tr>
<tr>
<td>map-062.dmp</td>
</tr>
</tbody>
</table>

Figure 5: Working with mappings

2.3.1.2 Information Tabs

In the lower pane of Memory Parsers main view are several tabs that provide additional information about the process dump. These tabs will be described in the following.

Process Dump Information (Linux/Windows)

This tab shows general information about the dumped process.

Figure 6: Process Dump Information (Windows process)
### Process Dump Analyses

#### Mapped Executables (Linux/Windows)

This tab shows all the mapped executables of the process.

<table>
<thead>
<tr>
<th>Name</th>
<th>Base Address</th>
<th>Mappings</th>
<th># of Mappings</th>
<th>Calculated .text section SHA-1 Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\Windows\system32\winreg.exe</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>48013464084187979278819925103361</td>
</tr>
<tr>
<td>c:\Windows\system32\snup.exe</td>
<td>0x04790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>346473280098895298733973</td>
</tr>
<tr>
<td>c:\Windows\system32\uxtheme.dll</td>
<td>0x01790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>276191564192588796191251</td>
</tr>
<tr>
<td>c:\Windows\system32\downlevel.dll</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>123456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\kernel32.dll</td>
<td>0x02790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\lssystemdll.dll</td>
<td>0x01790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\имвол.dll</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Base Address</th>
<th>Mappings</th>
<th># of Mappings</th>
<th>Calculated .text section SHA-1 Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\Windows\system32\winreg.exe</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>48013464084187979278819925103361</td>
</tr>
<tr>
<td>c:\Windows\system32\snup.exe</td>
<td>0x04790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>346473280098895298733973</td>
</tr>
<tr>
<td>c:\Windows\system32\uxtheme.dll</td>
<td>0x01790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>276191564192588796191251</td>
</tr>
<tr>
<td>c:\Windows\system32\downlevel.dll</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>123456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\kernel32.dll</td>
<td>0x02790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\lssystemdll.dll</td>
<td>0x01790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
<tr>
<td>c:\Windows\system32\имвол.dll</td>
<td>0x03790000</td>
<td>09:08:00:00:00</td>
<td>4</td>
<td>23456789012345678901234</td>
</tr>
</tbody>
</table>

### Calculated .text section SHA-1 Hash

If it is a dump of a Windows process, Memory Parser is able to compute a SHA-1 hash of the .text section of the mapped executable. This hash is shown in this row. The hash can be used to identify manipulations of mapped DLLs or to identify injected DLLs.

### DB .text section SHA-1 Hash

If it is a dump of a Windows process and the “Check Hashes” feature was used, this row contains the SHA-1 hash from the reference database.

### Hash Match

If it is a dump of a Windows process and the “Check Hashes” feature was used, this row shows whether the hashes of the “Calculated .text section SHA-1 Hash” and the “DB .text section SHA-1 Hash” rows are matching.

### DB Name

If it is a dump of a Windows process and the “Check Hashes” feature was used, this row contains the executable name from the reference database.
If it is a dump of a Windows process and the “Check Hashes” feature was used, this row contains the executable description from the reference database.

Table 2: Mapped executables information

### Threads (Windows)
This tab contains a list of all threads of the dumped process. Furthermore the priority, the status as well as the register values of each thread are shown.

![Thread information](image)

**Figure 9: Thread information**

If a register value points into one of the code or data mappings of the process the appropriate information is shown.

Example: ESP: 0x0007feac (Mapping: mem-003.dmp, Offset: 0x5eac)

In the example shown in Figure 9 the ESP register points with an offset of 0x5eac into the data mapping mem-003.dmp of the process. Therefore it is very likely that this mapping is the stack of the thread with ID 1308. This information is also shown in the “Comment” row of mem-003.dmp in the upper mapping list view (Stack of Thread ID: 1308 (0x0000051c)).
Environment and Status (Linux)

This tab contains information about the environment as well as the status of the process that was dumped.

![Environment and Status](image)

**Figure 10:** Process environment and status

Map (Linux)

This tab contains the same information as the `/proc/PID/maps` file of the process.

Registers (Linux)

This tab contains the register values of the dumped process.

![Registers](image)

**Figure 11:** Register values

If a register value points into one of the code or data mappings of the process the appropriate information is shown.
File Descriptors (Linux)
This tab contains the file descriptors that were used by the dumped process.

<table>
<thead>
<tr>
<th>File Descriptor</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>/proc/2006/0d2</td>
<td>/dev/null</td>
</tr>
<tr>
<td>/proc/2006/0d3</td>
<td>/dev/null</td>
</tr>
<tr>
<td>/proc/2006/0d2</td>
<td>/dev/null</td>
</tr>
<tr>
<td>/proc/2006/0d3</td>
<td>socket[3001]</td>
</tr>
</tbody>
</table>

Figure 12: File descriptors

2.3.1.3 Configuration
Memory Parser can be configured using the XML file mmp.cfg that can be found in the same directory as the Memory Parser binary. Within this file it is possible to configure the analysis tools which show up when right-clicking a mapping for further analyses.

```xml
<?xml version="1.0" ?>
<settings>
  <!-- currently only six external tools are supported
  <tool1 name="Notepad" path="c:\Windows\notepad.exe" menu="Notepad" />
  <tool2 name="empty" path="empty" menu="empty" />
  <tool3 name="empty" path="empty" menu="empty" />
  <tool4 name="empty" path="empty" menu="empty" />
  <tool5 name="empty" path="empty" menu="empty" />
  <tool6 name="empty" path="empty" menu="empty" />
</settings>
```

The above example shows the default configuration of Memory Parser. Only the notepad utility is specified in the default configuration. To add a new tool, just exchange the “empty” placeholders with the appropriate information. In the current version of Memory Parser six external analyses tools are supported.

2.4 Example Analyses
In order to describe the features of Process Dumper and Memory Parser, two exemplary process memory dumps will be analysed in the following.

2.4.1 Example Analysis 1: Remote Code Injection - Apache /Linux
In this example a dump of an Apache process will be analysed. The Apache process was compromised by a remote code injection attack.

After opening the dump in Memory Parser we first take a look at the register values. What’s very suspicious is that the instruction pointer (EIP register) points into the data mapping mem-002.dmp with an offset of 0x60191 (see Figure 13). This is a very suspicious behavior, since the instruction pointer should normally refer to a code mapping. The data mapping the instruction pointer points to is the heap of the process. It’s very likely that some malicious code was placed onto the heap and executed afterwards.
Process Dump Analyses

Memory Parser offers the possibility to have a closer look at each individual mapping of the dump. In order to accomplish a deeper analysis of the indicated offset within the referred data mapping `mem-002.dmp` right-click the mapping and choose the appropriate external tool. In this example the mapping will be further examined using the Disassembler IDA Pro [7].

![Figure 13: Register values](image)

The disassembly of the data mapping shows (see Figure 15) that there is indeed executable assembler code found at the indicated offset (0x60191). The functionality of that code can now be analysed.

![Figure 14: Analyse individual mappings](image)
Another suspicious behaviour of the dumped process can be found while evaluating the file descriptors used by the process (see Figure 16). The first three file descriptors (stdin, stdout, stderr) are identical with a likewise opened TCP socket (socket: [1004]). This is a usual procedure with remote code injection. In order to be able to communicate with the hijacked process an attacker normally duplicates the network socket to the stdin, stdout and stderr file descriptors of the process.

Apart from these described examples there are many other possibilities to analyse a process dump. For example it is possible to scan all data mappings of the process dump for signs for executable code. As previously mentioned, no executable code should be found in such areas of a process. For this purpose I developed two plugins for the IDA Pro Disassembler (Malicious Code Profiler and NOP Sled Detector).

2.4.2 Example Analysis 2: Remote Library Injection – IIS/Windows

In this second example a dump of an IIS Web server is analysed. The server process was compromised using the Meterpreter functionality of the Metasploit Framework (see [8]). With the help of Meterpreter it is possible to inject a DLL into the vulnerable process. The DLL is only present in memory and will thereby never be copied to the harddisk of the target system. This kind of attack cannot be reconstructed with the help of post-mortem analysis of the systems harddisk(s).

Memory Parser allows to compute SHA-1 hashes of the code areas of the mapped DLLs. Furthermore it is possible to compare these hashes against arbitrary hash databases. So it’s possible to create a reference hash database of the code section of every DLL on a Windows system and then compare the DLLs of a process dump against this baseline database. The tool MMPHash (see [9]) can be used to create such a
hash database. Figure 17 shows the output of a comparison of the mapped DLLs and a reference hash database of known Windows DLLs.

<table>
<thead>
<tr>
<th>Process Dump Information</th>
<th>Mapped Executables</th>
<th>Hash check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Base Address</td>
<td>Mappings</td>
</tr>
<tr>
<td>c:\windows\system32\drop.dll</td>
<td>0x7090000</td>
<td>456</td>
</tr>
<tr>
<td>c:\windows\system32\xwin.exe</td>
<td>0x7013000</td>
<td>262</td>
</tr>
<tr>
<td>c:\windows\system32\webcache.dll</td>
<td>0x01c0000</td>
<td>120</td>
</tr>
<tr>
<td>c:\windows\system32\calc7.dll</td>
<td>0x01e0000</td>
<td>104</td>
</tr>
<tr>
<td>c:\windows\system32\calc.dll</td>
<td>0x01e0000</td>
<td>110</td>
</tr>
<tr>
<td>c:\windows\system32\calc.dll</td>
<td>0x01e0000</td>
<td>120</td>
</tr>
<tr>
<td>c:\windows\system32\calc.dll</td>
<td>0x01e0000</td>
<td>110</td>
</tr>
</tbody>
</table>

**Figure 17:** Hash check

It turns out that four DLLs are not found in the reference database which is very suspicious (see the “Hash Match” row in Figure 17).

This method can be used to identify DLL injection and other manipulation techniques where DLLs are modified in memory (e.g. DLL/API Hooking, see [10]).

## 3 References


