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.

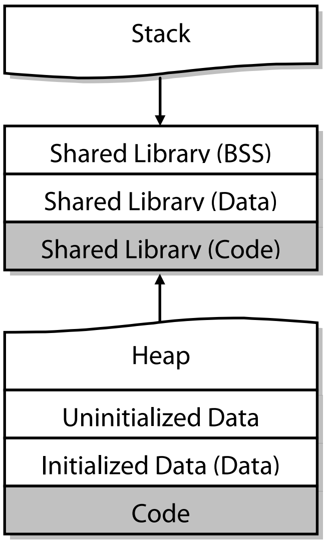


Figure 1: Process Memory Layout

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
pd, version 1.1 tk 2006, www.trapkit.de

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`:

```
C:\>tasklist /FI "IMAGENAME eq lsass.exe"
lsass.exe                  1008 Console                    0         1.368 K
```

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:\>pd -p 1008 > lsass.dump
pd, version 1.1 tk 2006, www.trapkit.de

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:\>pd -p 1008 | nc 192.168.0.100 7000
pd, version 1.1 tk 2006, www.trapkit.de

Dump finished.
```

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

```
C:\>pd -v -p 1008 > lsass.dump
pd, version 1.1 tk 2006, www.trapkit.de

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).

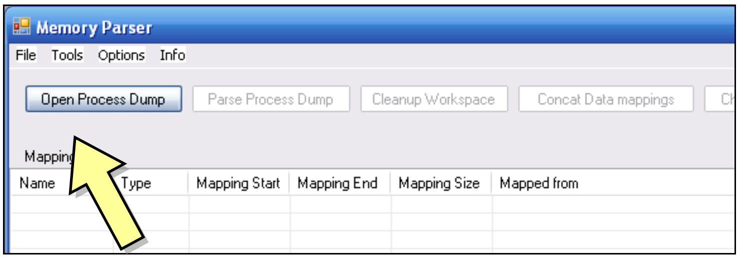


Figure 2: Open a process dump

To parse the process dump press the “Parse Process Dump” button (see Figure 3).

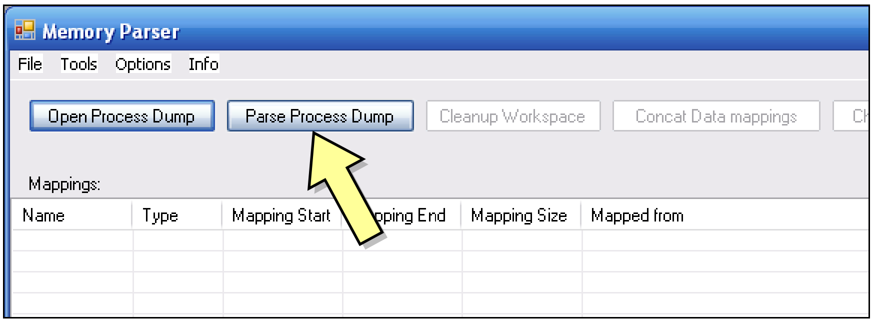
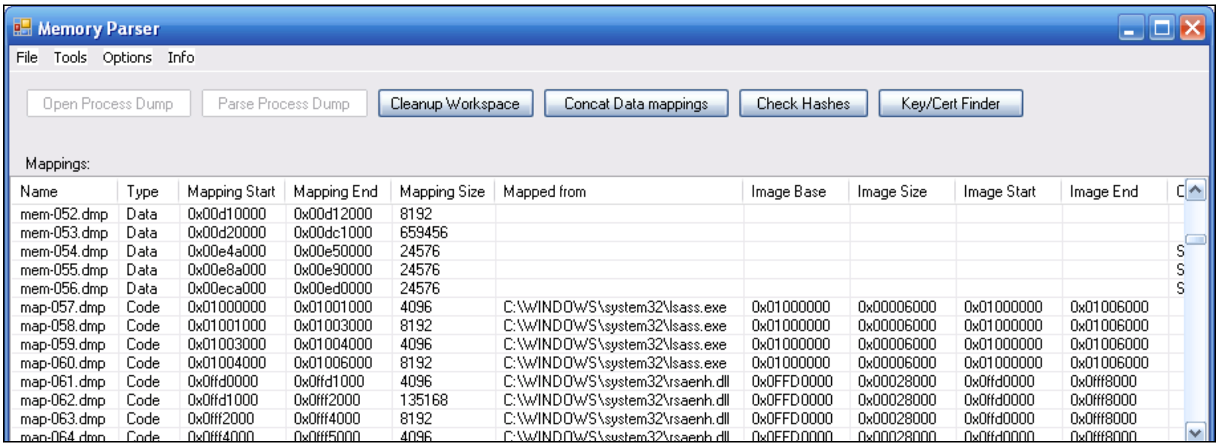


Figure 3: Parse a process dump

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).



Name	Type	Mapping Start	Mapping End	Mapping Size	Mapped from	Image Base	Image Size	Image Start	Image End	C
mem-052.dmp	Data	0x00d10000	0x00d12000	8192						
mem-053.dmp	Data	0x00d20000	0x00dc1000	659456						
mem-054.dmp	Data	0x00e4a000	0x00e50000	24576						S
mem-055.dmp	Data	0x00e8a000	0x00e90000	24576						S
mem-056.dmp	Data	0x00eca000	0x00ed0000	24576						S
map-057.dmp	Code	0x01000000	0x01001000	4096	C:\WINDOWS\system32\lsass.exe	0x01000000	0x00006000	0x01000000	0x01006000	
map-058.dmp	Code	0x01001000	0x01003000	8192	C:\WINDOWS\system32\lsass.exe	0x01000000	0x00006000	0x01000000	0x01006000	
map-059.dmp	Code	0x01003000	0x01004000	4096	C:\WINDOWS\system32\lsass.exe	0x01000000	0x00006000	0x01000000	0x01006000	
map-060.dmp	Code	0x01004000	0x01006000	8192	C:\WINDOWS\system32\lsass.exe	0x01000000	0x00006000	0x01000000	0x01006000	
map-061.dmp	Code	0x0ffd0000	0x0ffd1000	4096	C:\WINDOWS\system32\rsaenh.dll	0x0FFD0000	0x00028000	0x0ffd0000	0x0fff8000	
map-062.dmp	Code	0x0ffd1000	0x0fff2000	135168	C:\WINDOWS\system32\rsaenh.dll	0x0FFD0000	0x00028000	0x0ffd0000	0x0fff8000	
map-063.dmp	Code	0x0fff2000	0x0fff4000	8192	C:\WINDOWS\system32\rsaenh.dll	0x0FFD0000	0x00028000	0x0ffd0000	0x0fff8000	
map-064.dmp	Code	0x0fff4000	0x0fff5000	4096	C:\WINDOWS\system32\rsaenh.dll	0x0FFD0000	0x00028000	0x0ffd0000	0x0fff8000	

Figure 4: Process dump mappings

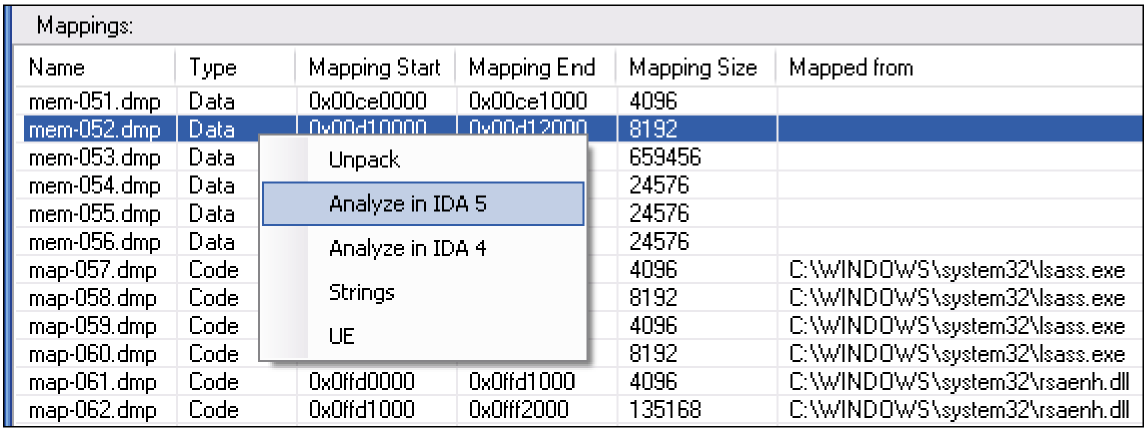
The information about every mapping is displayed in the following rows:

Row name	Information
`Name`	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.
`Type`	Data or Code
`Mapping Start`	The start address of the mapping within the virtual address space of the dumped process.
`Mapping End`	The end address of the mapping within the virtual address space of the dumped process.
`Mapping Size`	The size of the mapping.
`Mapped From`	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.
`Image Base`	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.
`Image Size`	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.
`Image Start`	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.
`Image End`	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.
`Comment`	This row is used for further descriptions of the mapping.

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).



Mappings:

Name	Type	Mapping Start	Mapping End	Mapping Size	Mapped from
mem-051.dmp	Data	0x00ce0000	0x00ce1000	4096	
mem-052.dmp	Data	0x00d10000	0x00d12000	8192	
mem-053.dmp	Data			659456	
mem-054.dmp	Data			24576	
mem-055.dmp	Data			24576	
mem-056.dmp	Data			24576	
map-057.dmp	Code			4096	C:\WINDOWS\system32\lsass.exe
map-058.dmp	Code			8192	C:\WINDOWS\system32\lsass.exe
map-059.dmp	Code			4096	C:\WINDOWS\system32\lsass.exe
map-060.dmp	Code			8192	C:\WINDOWS\system32\lsass.exe
map-061.dmp	Code	0x0ffd0000	0x0ffd1000	4096	C:\WINDOWS\system32\rsaenh.dll
map-062.dmp	Code	0x0ffd1000	0x0fff2000	135168	C:\WINDOWS\system32\rsaenh.dll

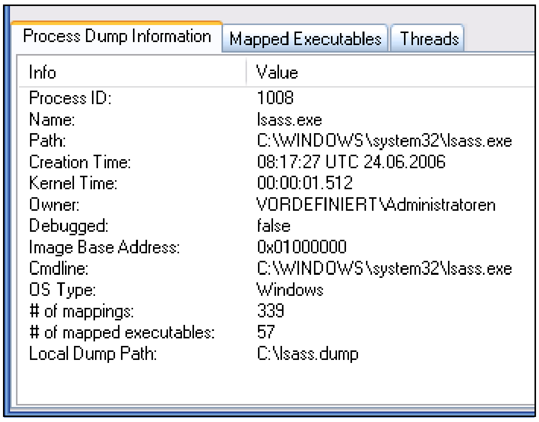
Figure 5: Working with mappings

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.

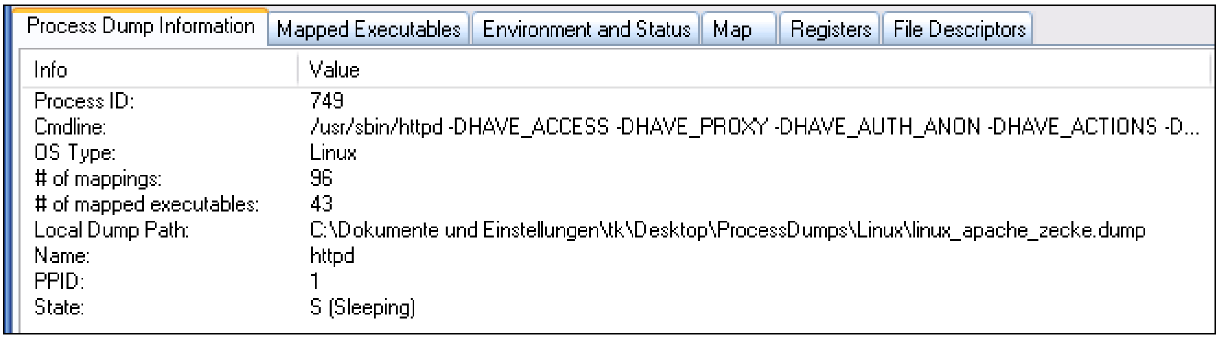
Process Dump Information (Linux/Windows)

This tab shows general information about the dumped process.



Info	Value
Process ID:	1008
Name:	lsass.exe
Path:	C:\WINDOWS\system32\lsass.exe
Creation Time:	08:17:27 UTC 24.06.2006
Kernel Time:	00:00:01.512
Owner:	VORDEFINIERT\Administratoren
Debugged:	false
Image Base Address:	0x01000000
Cmdline:	C:\WINDOWS\system32\lsass.exe
OS Type:	Windows
# of mappings:	339
# of mapped executables:	57
Local Dump Path:	C:\lsass.dump

Figure 6: Process Dump Information (Windows process)

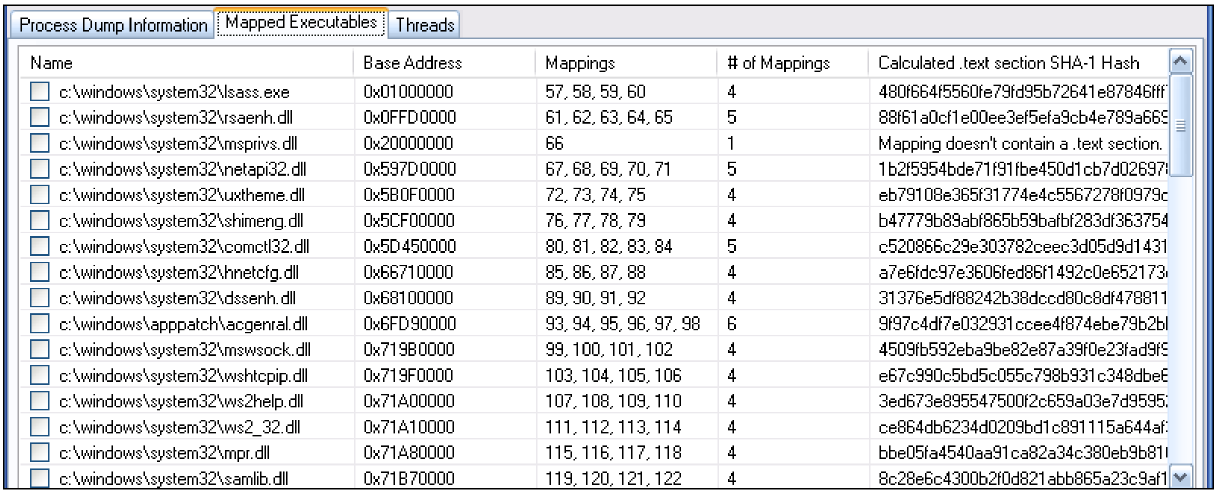


Info	Value
Process ID:	749
Cmdline:	/usr/sbin/httpd -DHAVE_ACCESS -DHAVE_PROXY -DHAVE_AUTH_ANON -DHAVE_ACTIONS -D...
OS Type:	Linux
# of mappings:	96
# of mapped executables:	43
Local Dump Path:	C:\Dokumente und Einstellungen\tk\Desktop\ProcessDumps\Linux\linux_apache_zecke.dump
Name:	httpd
PPID:	1
State:	S (Sleeping)

Figure 7: Process Dump Information (Linux process)

Mapped Executables (Linux/Windows)

This tab shows all the mapped executables of the process.



Name	Base Address	Mappings	# of Mappings	Calculated .text section SHA-1 Hash
□ c:\windows\system32\lsass.exe	0x01000000	57, 58, 59, 60	4	480f664f5560fe79fd95b72641e87846fff
□ c:\windows\system32\rsaenh.dll	0x0FFD0000	61, 62, 63, 64, 65	5	88f61a0cf1e00ee3ef5efa9cb4e789a66
□ c:\windows\system32\msprivs.dll	0x20000000	66	1	Mapping doesn't contain a .text section.
□ c:\windows\system32\netapi32.dll	0x597D0000	67, 68, 69, 70, 71	5	1b2f5954bde71f91fbe450d1cb7d02697
□ c:\windows\system32\uxtheme.dll	0x5B0F0000	72, 73, 74, 75	4	eb79108e365f31774e4c5567278f0979c
□ c:\windows\system32\shimeng.dll	0x5CF00000	76, 77, 78, 79	4	b47779b89abf865b59bafbf283df363754
□ c:\windows\system32\comctl32.dll	0x5D450000	80, 81, 82, 83, 84	5	c520866c29e303782ceec3d05d9d1431
□ c:\windows\system32\hnetcfg.dll	0x66710000	85, 86, 87, 88	4	a7e6fdc97e3606fed86f1492c0e652173
□ c:\windows\system32\dssenh.dll	0x68100000	89, 90, 91, 92	4	31376e5df88242b38dccd80c8df478811
□ c:\windows\apppatch\acgenral.dll	0x6FD90000	93, 94, 95, 96, 97, 98	6	9f97c4df7e032931ccee4f874ebe79b2b
□ c:\windows\system32\mswsock.dll	0x719B0000	99, 100, 101, 102	4	4509fb592eba9be82e87a39f0e23fad9f
□ c:\windows\system32\wshtcpip.dll	0x719F0000	103, 104, 105, 106	4	e67c990c5bd5c055c798b931c348dbe
□ c:\windows\system32\ws2help.dll	0x71A00000	107, 108, 109, 110	4	3ed673e895547500f2c659a03e7d9595
□ c:\windows\system32\ws2_32.dll	0x71A10000	111, 112, 113, 114	4	ce864db6234d0209bd1c891115a644a
□ c:\windows\system32\mpr.dll	0x71A80000	115, 116, 117, 118	4	bbe05fa4540aa91ca82a34c380eb9b81
□ c:\windows\system32\samlib.dll	0x71B70000	119, 120, 121, 122	4	8c28e6c4300b2f0d821abb865a23c9af1

Figure 8: Mapped Executables

The information of every mapped executable is displayed in the following rows:

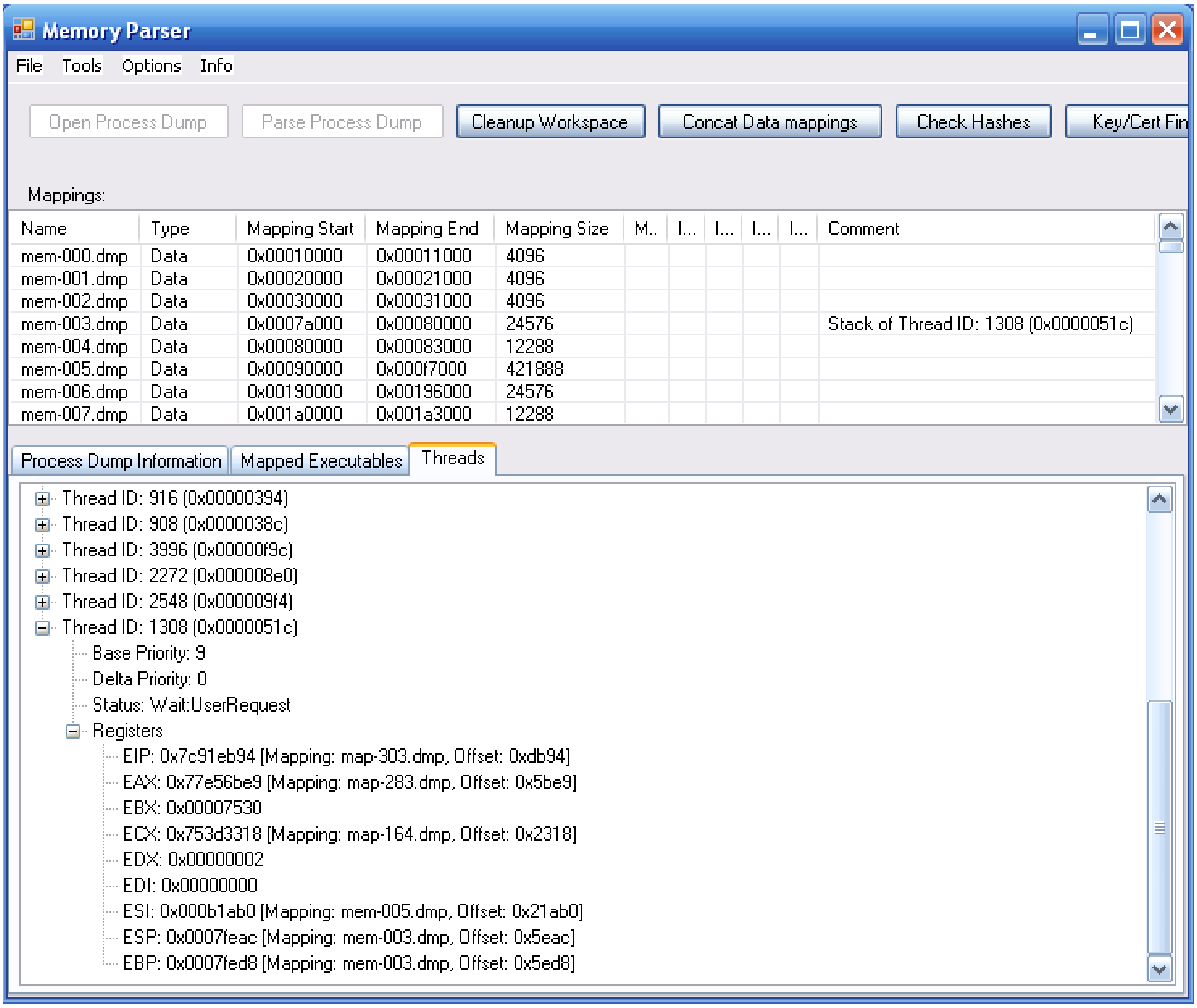
Row name	Information
`Name`	The path of the mapped executable image.
`Base Address`	The base address of the executable.
`Mappings`	A list of the individual mappings (code, data) of the executable.
`# of Mappings`	The number of mappings (code, data) of the mapped executable.
`Calculated .text section 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.

`DB Description`	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.



Name	Type	Mapping Start	Mapping End	Mapping Size	M..	I...	I...	I...	I...	Comment
mem-000.dmp	Data	0x00010000	0x00011000	4096						
mem-001.dmp	Data	0x00020000	0x00021000	4096						
mem-002.dmp	Data	0x00030000	0x00031000	4096						
mem-003.dmp	Data	0x0007a000	0x00080000	24576						Stack of Thread ID: 1308 (0x0000051c)
mem-004.dmp	Data	0x00080000	0x00083000	12288						
mem-005.dmp	Data	0x00090000	0x000f7000	421888						
mem-006.dmp	Data	0x00190000	0x00196000	24576						
mem-007.dmp	Data	0x001a0000	0x001a3000	12288						

```
⊞ Thread ID: 916 (0x00000394)
⊞ Thread ID: 908 (0x0000038c)
⊞ Thread ID: 3996 (0x00000f9c)
⊞ Thread ID: 2272 (0x000008e0)
⊞ Thread ID: 2548 (0x000009f4)
⊟ Thread ID: 1308 (0x0000051c)
    Base Priority: 9
    Delta Priority: 0
    Status: Wait:UserRequest
  ⊟ Registers
      EIP: 0x7c91eb94 [Mapping: map-303.dmp, Offset: 0xdb94]
      EAX: 0x77e56be9 [Mapping: map-283.dmp, Offset: 0x5be9]
      EBX: 0x00007530
      ECX: 0x753d3318 [Mapping: map-164.dmp, Offset: 0x2318]
      EDX: 0x00000002
      EDI: 0x00000000
      ESI: 0x000b1ab0 [Mapping: mem-005.dmp, Offset: 0x21ab0]
      ESP: 0x0007feac [Mapping: mem-003.dmp, Offset: 0x5eac]
      EBP: 0x0007fed8 [Mapping: mem-003.dmp, Offset: 0x5ed8]
```

Figure 9: Thread information

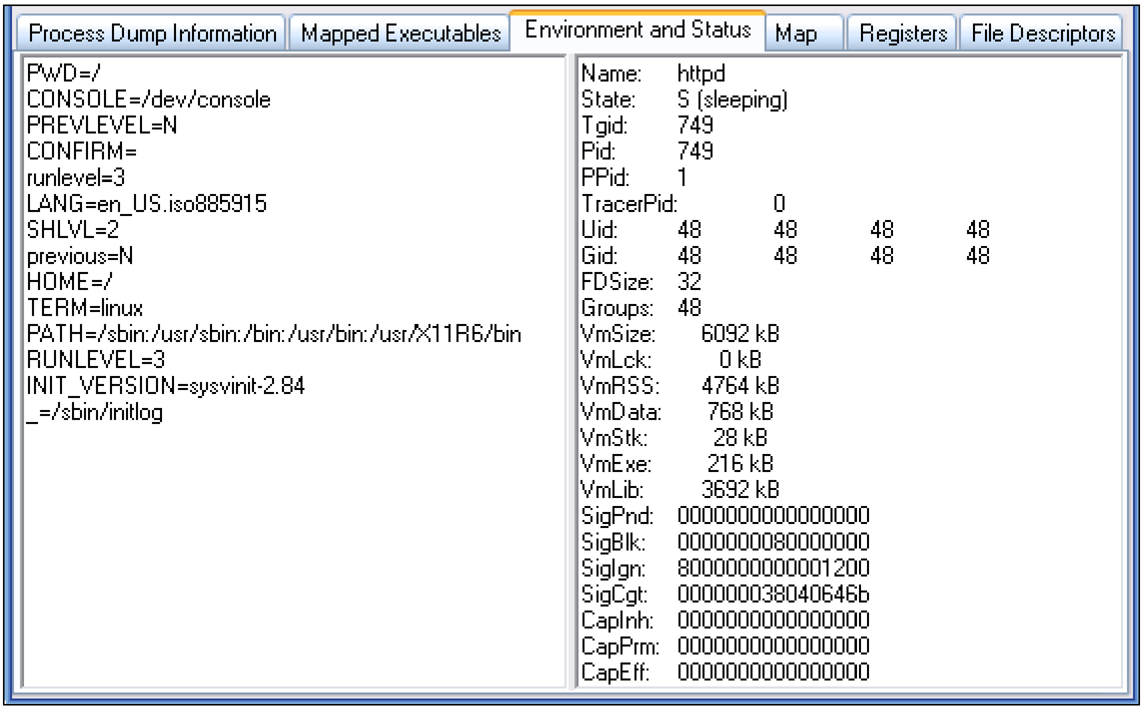
If a register value points into one of the code or data mappings of the process the appropriate information is shown.

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.



```
PWD=/
CONSOLE=/dev/console
PREVLEVEL=N
CONFIRM=
runlevel=3
LANG=en_US.iso885915
SHLVL=2
previous=N
HOME=/
TERM=linux
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
RUNLEVEL=3
INIT_VERSION=sysvinit-2.84
_=/sbin/initlog
```

```
Name:     httpd
State:    S (sleeping)
Tgid:     749
Pid:      749
PPid:     1
TracerPid:        0
Uid:      48        48        48        48
Gid:      48        48        48        48
FDSize:   32
Groups:   48
VmSize:     6092 kB
VmLck:         0 kB
VmRSS:      4764 kB
VmData:      768 kB
VmStk:        28 kB
VmExe:       216 kB
VmLib:      3692 kB
SigPnd:   0000000000000000
SigBlk:   0000000080000000
SigIgn:   8000000000001200
SigCgt:   000000038040646b
CapInh:   0000000000000000
CapPrm:   0000000000000000
CapEff:   0000000000000000
```

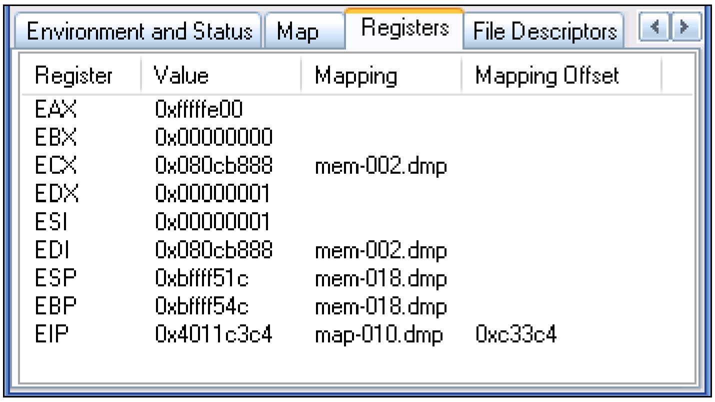
Figure 10: Process environment and status

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.



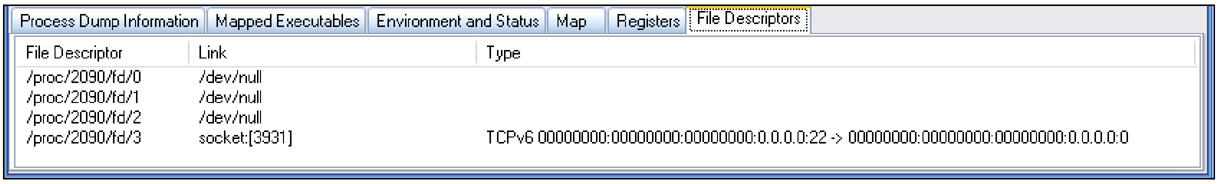
Register	Value	Mapping	Mapping Offset
EAX	0xfffffe00		
EBX	0x00000000		
ECX	0x080cb888	mem-002.dmp	
EDX	0x00000001		
ESI	0x00000001		
EDI	0x080cb888	mem-002.dmp	
ESP	0xbffff51c	mem-018.dmp	
EBP	0xbffff54c	mem-018.dmp	
EIP	0x4011c3c4	map-010.dmp	0xc33c4

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.



File Descriptor	Link	Type
/proc/2090/fd/0	/dev/null	
/proc/2090/fd/1	/dev/null	
/proc/2090/fd/2	/dev/null	
/proc/2090/fd/3	socket:[3931]	TCPv6 00000000:00000000:00000000:0.0.0.0:22 -> 00000000:00000000:00000000:0.0.0.0:0

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 version="1.0" ?>
<!-- mmp settings -->

<settings>

        <!-- currently only six external tools are supported          -->
        <!--                                                          -->
        <!-- XML nodes: tool1, tool2, tool3, tool4, tool5 and tool6    -->
        <!--                                                          -->
        <!-- Values:                                                  -->
        <!--   name - name of the external program                    -->
        <!--   path - path of the external program                    -->
        <!--   menu - name of the external program in the context menu -->

  <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.

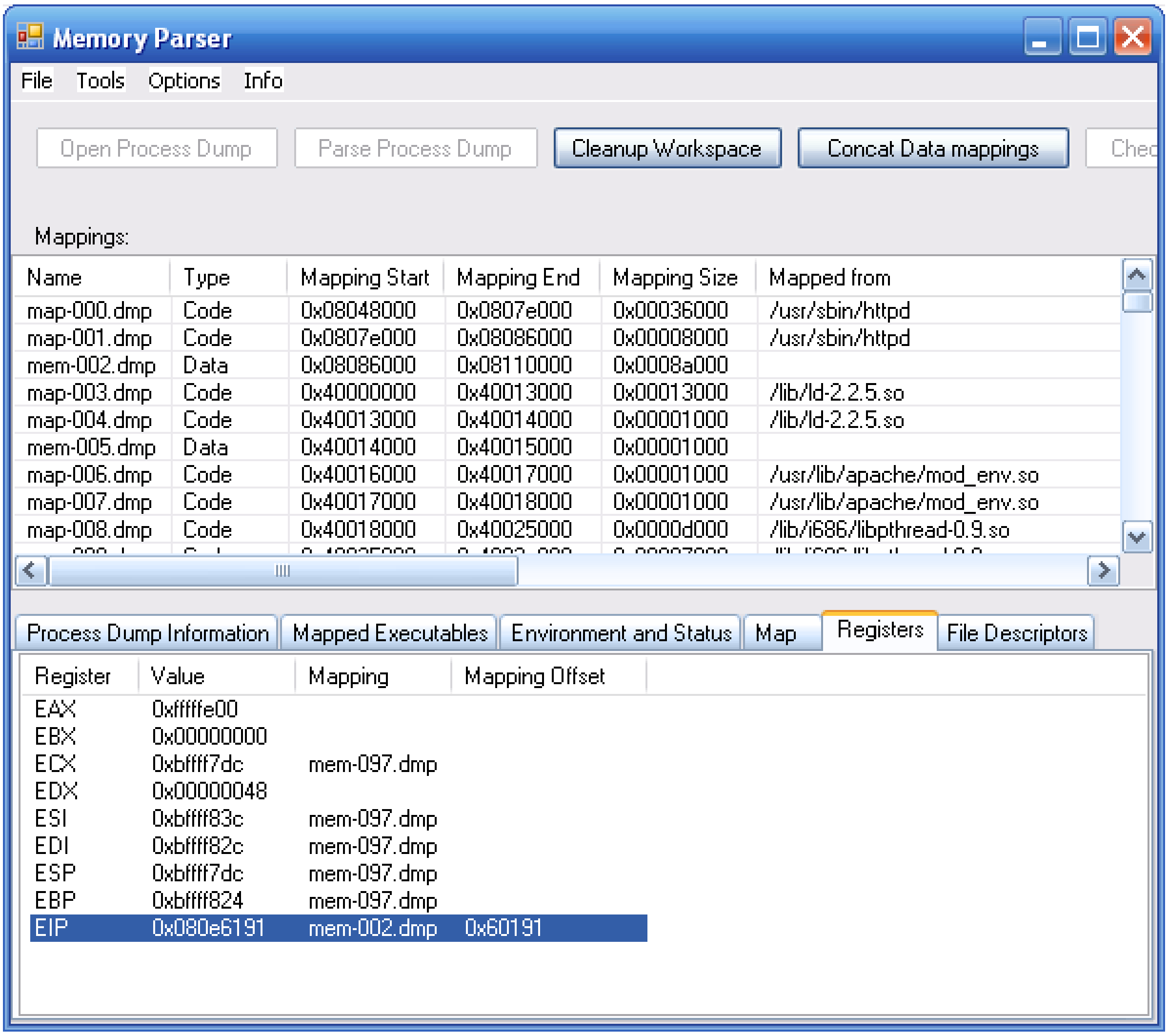
2.4 Example Analyses

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.

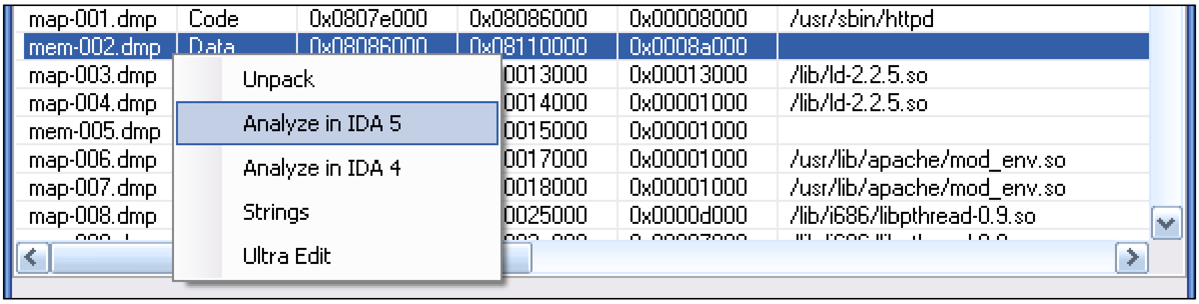


Name	Type	Mapping Start	Mapping End	Mapping Size	Mapped from
map-000.dmp	Code	0x08048000	0x0807e000	0x00036000	/usr/sbin/httpd
map-001.dmp	Code	0x0807e000	0x08086000	0x00008000	/usr/sbin/httpd
mem-002.dmp	Data	0x08086000	0x08110000	0x0008a000	
map-003.dmp	Code	0x40000000	0x40013000	0x00013000	/lib/ld-2.2.5.so
map-004.dmp	Code	0x40013000	0x40014000	0x00001000	/lib/ld-2.2.5.so
mem-005.dmp	Data	0x40014000	0x40015000	0x00001000	
map-006.dmp	Code	0x40016000	0x40017000	0x00001000	/usr/lib/apache/mod_env.so
map-007.dmp	Code	0x40017000	0x40018000	0x00001000	/usr/lib/apache/mod_env.so
map-008.dmp	Code	0x40018000	0x40025000	0x0000d000	/lib/i686/libpthread-0.9.so

Register	Value	Mapping	Mapping Offset
EAX	0xffffe00		
EBX	0x00000000		
ECX	0xbffff7dc	mem-097.dmp	
EDX	0x00000048		
ESI	0xbffff83c	mem-097.dmp	
EDI	0xbffff82c	mem-097.dmp	
ESP	0xbffff7dc	mem-097.dmp	
EBP	0xbffff824	mem-097.dmp	
EIP	0x080e6191	mem-002.dmp	0x60191

Figure 13: Register values

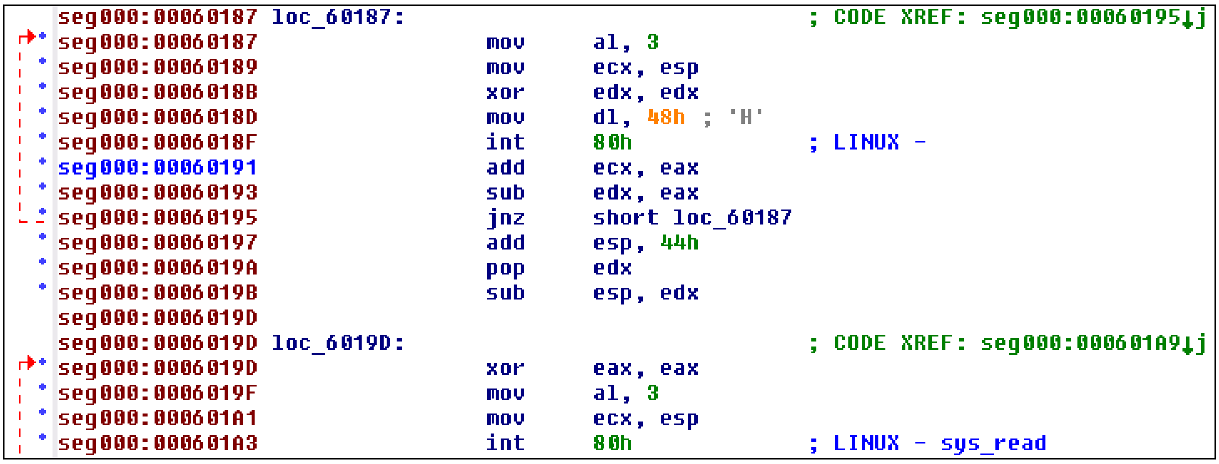
Memory Parser offers the possibility to have a closer look at each individual mapping of the dump. In order to accomplish a deeper analysis of the indicated offset within the referred data mapping `mem-002.dmp` right-click the mapping and choose the appropriate external tool. In this example the mapping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



map-001.dmp	Code	0x0807e000	0x08086000	0x00008000	/usr/sbin/httpd
mem-002.dmp	Data	0x08086000	0x08110000	0x0008a000	
map-003.dmp			0013000	0x00013000	/lib/ld-2.2.5.so
map-004.dmp			0014000	0x00001000	/lib/ld-2.2.5.so
mem-005.dmp			0015000	0x00001000	
map-006.dmp			0017000	0x00001000	/usr/lib/apache/mod_env.so
map-007.dmp			0018000	0x00001000	/usr/lib/apache/mod_env.so
map-008.dmp			0025000	0x0000d000	/lib/i686/libpthread-0.9.so

Figure 14: Analyse individual mappings

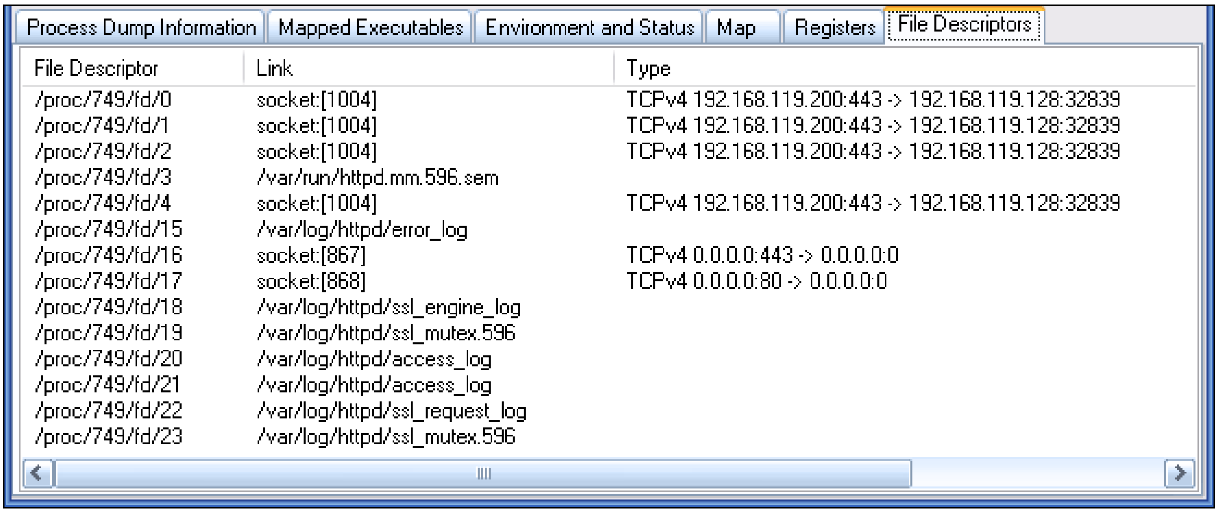
The disassembly of the data mapping shows (see Figure 15) that there is indeed executable assembler code found at the indicated offset (`0x60191`). The functionality of that code can now be analysed.



```
seg000:00060187 loc_60187:                          ; CODE XREF: seg000:00060195↓j
seg000:00060187                 mov     al, 3
seg000:00060189                 mov     ecx, esp
seg000:0006018B                 xor     edx, edx
seg000:0006018D                 mov     dl, 48h ; 'H'
seg000:0006018F                 int     80h             ; LINUX -
seg000:00060191                 add     ecx, eax
seg000:00060193                 sub     edx, eax
seg000:00060195                 jnz     short loc_60187
seg000:00060197                 add     esp, 44h
seg000:0006019A                 pop     edx
seg000:0006019B                 sub     esp, edx
seg000:0006019D
seg000:0006019D loc_6019D:                          ; CODE XREF: seg000:000601A9↓j
seg000:0006019D                 xor     eax, eax
seg000:0006019F                 mov     al, 3
seg000:000601A1                 mov     ecx, esp
seg000:000601A3                 int     80h             ; LINUX - sys_read
```

Figure 15: Disassembly of the data mapping

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.



File Descriptor	Link	Type
/proc/749/fd/0	socket:[1004]	TCPv4 192.168.119.200:443 -> 192.168.119.128:32839
/proc/749/fd/1	socket:[1004]	TCPv4 192.168.119.200:443 -> 192.168.119.128:32839
/proc/749/fd/2	socket:[1004]	TCPv4 192.168.119.200:443 -> 192.168.119.128:32839
/proc/749/fd/3	/var/run/httpd.mm.596.sem	
/proc/749/fd/4	socket:[1004]	TCPv4 192.168.119.200:443 -> 192.168.119.128:32839
/proc/749/fd/15	/var/log/httpd/error_log	
/proc/749/fd/16	socket:[867]	TCPv4 0.0.0.0:443 -> 0.0.0.0:0
/proc/749/fd/17	socket:[868]	TCPv4 0.0.0.0:80 -> 0.0.0.0:0
/proc/749/fd/18	/var/log/httpd/ssl_engine_log	
/proc/749/fd/19	/var/log/httpd/ssl_mutex.596	
/proc/749/fd/20	/var/log/httpd/access_log	
/proc/749/fd/21	/var/log/httpd/access_log	
/proc/749/fd/22	/var/log/httpd/ssl_request_log	
/proc/749/fd/23	/var/log/httpd/ssl_mutex.596	

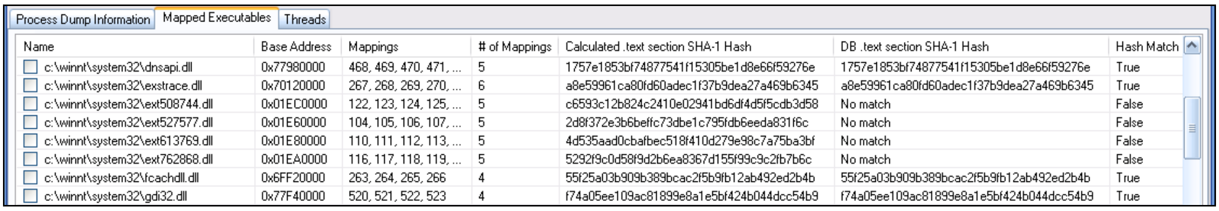
Figure 16: Process file descriptors

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.



Name	Base Address	Mappings	# of Mappings	Calculated .text section SHA-1 Hash	DB .text section SHA-1 Hash	Hash Match
□ c:\winnt\system32\dnsapi.dll	0x77980000	468, 469, 470, 471, ...	5	1757e1853bf74877541f15305be1d8e66f59276e	1757e1853bf74877541f15305be1d8e66f59276e	True
□ c:\winnt\system32\exstrace.dll	0x70120000	267, 268, 269, 270, ...	6	a8e59961ca80fd60adec1f37b9dea27a469b6345	a8e59961ca80fd60adec1f37b9dea27a469b6345	True
□ c:\winnt\system32\ext508744.dll	0x01EC0000	122, 123, 124, 125, ...	5	c6593c12b824c2410e02941bd6df4d5f5cdb3d58	No match	False
□ c:\winnt\system32\ext527577.dll	0x01E60000	104, 105, 106, 107, ...	5	2d8f372e3b6beffc73dbe1c795fdb6eeda831f6c	No match	False
□ c:\winnt\system32\ext613769.dll	0x01E80000	110, 111, 112, 113, ...	5	4d535aad0cbafbec518f410d279e98c7a75ba3bf	No match	False
□ c:\winnt\system32\ext762868.dll	0x01EA0000	116, 117, 118, 119, ...	5	5292f9c0d58f9d2b6ea8367d155f99c9c2fb7b6c	No match	False
□ c:\winnt\system32\fcachdll.dll	0x6FF20000	263, 264, 265, 266	4	55f25a03b909b389bcac2f5b9fb12ab492ed2b4b	55f25a03b909b389bcac2f5b9fb12ab492ed2b4b	True
□ c:\winnt\system32\gdi32.dll	0x77F40000	520, 521, 522, 523	4	f74a05ee109ac81899e8a1e5bf424b044dcc54b9	f74a05ee109ac81899e8a1e5bf424b044dcc54b9	True

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

[1] pmdump, *http://ntsecurity.nu/toolbox/pmdump/*

[2] pcat, *http://www.porcupine.org/forensics/tct.html*

[3] Process Dumper (pd), *http://www.trapkit.de/research/forensic/pd/*

[4] netcat, *http://www.vulnwatch.org/netcat/*

[5] Memory Parser (MMP), *http://www.trapkit.de/research/forensic/mmp/*

[6] Klein, T.: “All your private keys are belong to us - Extracting RSA private keys and certificates from process memory”, *http://www.trapkit.de/research/sslkeyfinder/*

[7] IDA Pro, *http://www.datarescue.com*

[8] Metasploit, *http://www.metasploit.com*

[9] MMPHash, *http://www.trapkit.de/research/forensic/mmp/*

[10] Hoglund, G.; Butler, J.: „Rootkits: Subverting the Windows Kernel“, Addison-Wesley, 2006.