

Forensic Analysis of a Live Linux System, Pt. 2

Mariusz Burdach 2004-04-12

1. Introduction to part II

Last month in the [first part](#) of this article series, we discussed some of the preparation and steps that must be taking when analyzing a live Linux system that has been compromised. Now we'll continue our analysis by looking for malicious code on the running system, and then discuss some of the searches that can be done with the data once it has been transferred to our remote host.

Note:

Some readers, after reading the first part of this article, pointed out that before transferring any digital data from the compromised machine, the trusted shell should be run. I think that it's good idea, so we should compile statically for instance the bash shell and then copy them to our removable media. The step 2a should instead look as follows:

```
(compromised)# /mnt/cdrom/bash
```

We have to remember that almost any command interpreter writes every typed command to a history file. We don't want to modify the local file system so the best solution is to turn off history recording.

Please review [part one](#) of this article, for sections 2, 2.1, and 2.2 and 2.3 (steps 1 to 5) before continuing on.

2.3 Data collection for a live system - continued

Step 6: Physical memory image

One of the step in the digital data collection process is performing a copy of the whole system's memory. We can access physical memory directly by copying the `/dev/mem` device or by copying the `kcore` file. The `kcore` file represents the RAM in a Linux operating system. It can be found in the pseudo file system, which is mounted in the `/proc` directory. The size of this file is equal to the presented physical memory, extended by about 4KB. The advantage of using `kcore` is that it is presented in the ELF core format, so it can be debugged later by the `gdb` tool. The `gdb` utility is a great tool when we need to trace or analyze small pieces of code or memory. In section 2.7 I will show how to use this tool for more advanced offline forensic analysis. But for basic analysis we will use more universal tools such as `grep`, `strings` and hex editors.

Additionally, to properly analyze the whole physical memory we have to know some information from the page tables - the data structures that map virtual (linear) addresses to physical addresses. We have to know that physical memory contains of pages of virtual memory. In page tables we can find information about the order of pages (4 KB in Intel processors per page) written to the physical memory.

As mentioned earlier, we have to remember that by acquiring volatile memory using software tools we will by nature modify the content of that memory. Even worse, we can overwrite possible evidence.

In the example below have I copied a memory image using the `/proc` pseudo file system.

```
(remote)#nc -l -p port > kcore_compromised
(compromised)#/mnt/cdrom/dd < /proc/kcore | /mnt/cdrom/nc (remote) port
(remote)#md5sum kcore_compromised > kcore_compromised.md5
```

When copying the whole system's memory we copy both allocated and unallocated data because the process just copies the entire image of physical memory. In step 9 we will try to copy a particular process using the /proc pseudo file system. I have to mention that by using /proc, we can also modify the swap space. By copying the suspect process we will force some pages of the main memory to be read from the swap space, and other pages to write to it. Another important piece to remember is that we will copy only memory allocated by the process data.

Step 7: List of modules loaded to kernel memory of an operating system

We have to be sure that the collected data is complete, and that results from the netstat or lsof commands are not modified at the kernel level. So, we have to check to see which modules are currently loaded into memory.

```
(remote)# nc -l -p port > lkms_compromised
(compromised)#/mnt/cdrom/cat /proc/modules | /mnt/cdrom/nc (remote) port
(remote)# nc -l -p port > lkms_compromised.md5
(compromised)# /mnt/cdrom/md5sum /proc/modules | /mnt/cdrom/nc (remote) port
```

Unfortunately, some of malicious modules that may be present cannot be listed at all. To verify information about the loaded modules from /proc/modules I will use the method described in a recent issue of Phrack Magazine in the article, "Finding hidden kernel modules (the extrem way)". The hunter.o module checks a chain of modules loaded into the kernel.

```
(compromised)#/mnt/cdrom/insmod -f /mnt/cdrom/hunter.o
```

I will use the "-f switch" in this case, which forces the loading of the hunter.o because of version mismatch. It happens when the version of the kernel of the compromised system is different from the version of the kernel of the system which the hunter.o module was compiled on. The best solution is to recompile module code for each version of the kernel that it will be linked to. If we know which kernel version is used on the compromised machine we can download the proper source code from www.kernel.org and include the specific header files for that kernel in our hunter.o module.

```
(remote)#nc -l -p port > modules_hunter_compromised
(compromised)#/mnt/cdrom/cat /proc/showmodules && /mnt/cdrom/dmesg | /mnt/cdrom/nc (remote) port
(remote)#md5sum modules_hunter_compromised > modules_hunter_compromised.md5
```

Now we can compare the results. We also have to pay attention to the size of the modules. Sometimes a malicious code is stuck to a legal module.

The last thing which we have to do is copy the symbols exported by kernel modules. Sometimes the weak LKM based rootkit can export its symbols. By analyzing the ksyms file we can detect the presence of an intruder in

the system.

```
(remote)#nc -l -p port > ksyms_compromised
(compromised)#/mnt/cdrom/cat /proc/ksyms | /mnt/cdrom/nc (remote) port
(remote)# nc -l -p port > ksyms_compromised.md5
(compromised)#/mnt/cdrom/md5sum /proc/ksyms | /mnt/cdrom/nc (remote) port
```

We can instead use other tools which detect malicious modules (such as a kstat or kern_check), but unfortunately all of them use the System.map file. This file is generated after kernel compilation. If an administrator doesn't copy this file and doesn't create the checksum we should not trust the system call addresses which are presented there. Even when the system call addresses are valid an intruder can use another sophisticated method of hiding of a malicious module in kernel memory. For instance the adore-ng rootkit replaces the existing handler routines for providing directory listings.

If we are sure that system calls addresses are not modified in the System.map file or the ksyms file we can verify if some of system call addresses are not changed in the system call table by the intruder. For more details see the section 2.7.

Step 8: The list of active processes

Information about all processes, open ports and files can be collected by the lsof tool. Of course, this is true only when we don't detect any LKM based rootkits in memory. Additionally, in the next step we will copy suspicious processes. By copying a particular process we can better separate malicious data from the compromised system's memory, which we copied in its entirety. As you will recall, we collected the whole memory (kcore or /dev/mem) in step 6.

```
(remote)#nc -l -p port > lsof_compromised
(compromised)#/mnt/cdrom/lsof -n -P -l | /mnt/cdrom/nc (remote) port
(remote)#md5sum lsof_compromised > lsof_compromised.md5
```

Now, we have to analyze the result from the lsof tool. If any of the active processes are suspicious, now is a good time to copy them. Also, if we see in the results from lsof that the program which initiated the process is deleted by an intruder, we still have a chance to recover it. I will show you how to do this in the next section.

I have listed a few examples of suspicious processes below:

- o One process is listening on an atypical TCP/UDP port or open raw socket;
- o A process has an active connection with a remote host;
- o A program that was previously run has since been deleted;
- o A file, opened by a process, is deleted (for instance: a log file);
- o A strange process name;
- o A process was initiated by a user that does not exist, or by an unprivileged user.

Step 9: Collecting of suspicious processes

I will use the pcat tool to copy the entire memory allocated by a process.

```
(remote)#nc -l -p port > proc_id_compromised
(compromised)#/mnt/cdrom/pcat proc_id | /mnt/cdrom/nc (remote) port
(remote)#md5 proc_ip_compromised > proc_ip_compromised.md5
```

Additionally we can copy just certain data from a process. More information about it will be found in the next section.

Step 10: Useful information about the compromised system

Our next step is an accumulation of some useful information from the compromised system. This information is necessary to prepare a proper description of the incident and to perform a copy of all system files. Please remember that all results have to be sent to the remote host. In Table 2, I have placed commands which must be run on the compromised system.

Table 3: Useful information about the compromised host

Command	Description
/mnt/cdrom/cat /proc/version	Version of the operating system
/mnt/cdrom/cat /proc/sys/kernel/name	Host name
/mnt/cdrom/cat /proc/sys/kernel/domainname	Domain name
/mnt/cdrom/cat /proc/cpuinfo	Information about hardware
/mnt/cdrom/cat /proc/swaps	All swap partitions
/mnt/cdrom/cat /proc/partitions	All local file systems
/mnt/cdrom/cat /proc/self/mounts	Mounted file systems
/mnt/cdrom/cat /proc/uptime	Uptime

Step 11: Current time

The last step is an accumulation of information about the current time.

```
(remote)#nc -l -p port > end_time
(compromised)# /mnt/cdrom/date | /mnt/cdrom/nc (remote) port
```

Now we have reached the point where we can switch off the compromised system. Remember not to shut down the system by using any shutdown or init commands. We have to pull the power cable from the system or UPS device.

2.4 File system images

Before switching off of the compromised system we could have created a copy of all file systems and swap partitions, but we chosen not to. I suggest performing this task after switching off of the system. This way, we can be sure that

the contents of the compromised file system is no longer being modified. Once again, I have to mention that the swap space was modified during our acquisition process and some of evidences could have been overwritten.

A next step is to put a bootable media to a drive and to run the operating system from the media. We can use any Linux distribution which will not mount local file systems by default. Now from this point we can make a copy of all local partitions or the whole hard disk.

2.5 Basic data analysis

For the moment let's consider again the process we used in step 8, where we used the lsof tool. Let's look at this situation in more depth and consider two examples:

Example one: The program, which has initiated the process, is deleted.

In this case the Linux file, which has initiated the process, is still allocated in a memory. To recover the file we have to know the ID process created by the program. In the /proc/ directory we can find the executable file. This file is a copy of the deleted one. We have to send this file to the remote host.

When we recover the deleted file we can then extract information from it. We can choose one of two options: a static analysis by disassembling it, or a dynamic analysis by running this unknown file in a closely controlled environment.

Example two: The file, opened by an active process, is deleted (for instance a log file).

A fragment of the lsof result is presented below:

```
smbd      3137  root  rtd   DIR      8,1      4096          2 /
smbd      3137  root  txt   REG      8,1     672527      92030 /usr/bin/smbd -D
smbd      3137  root  mem   REG      8,1     485171      44656 /lib/ld-2.2.4.so
...
smbd      3137  root  16u   IPv4     976                TCP *:https (LISTEN)
smbd      3137  root  17u   IPv4     977                TCP *:http (LISTEN)
...
smbd      3137  root  20w   REG      8,1        253      46934 /var/log/httpd/access_log (deleted)
...
```

To recover this file we have to list the contents of the fd subdirectory (file descriptors) from the /proc/3137 directory.

```
(remote)# nc -l -p port > ls_from_proc_3137
(compromised)# /mnt/cdrom/ls -la /proc/3137/fd/ | /mnt/cdrom/nc (remote) port
(remote)# more ls_from_proc_3137

l-wx----- 1 root root 64 Aug 10 21:03 12 -> /var/log/httpd/access_log (deleted)
...
```

As we can see, the first file descriptor is deleted. All we have to do is to copy this file to the remote host by using the

netcat tool.

```
(remote)# nc -l -p port > deleted_access_log
(compromised)# /mnt/cdrom/cat /proc/3137/fd/1 | /mnt/cdrom/nc (remote) port
```

This is not the sole method of recovering this file. We can also recover the file during offline analysis when we are looking for unallocated i-nodes and data blocks.

2.6 Keyword searching

In this section of the article I would like to present one method of analysis for processes and memory images. Remember that this entire step is done on a remote host, after the imaging has been completed. The biggest advantage of this method is that we are not required to know the low level language used. I will use the following simple tools to search for signs of an intrusion:

- strings
- less
- grep

We will gather all printable characters from the image file by using the strings tool. The default settings let us view printable character sequences, which are at least four characters long. We will use the `-t` switch to add an offset from the beginning of the file.

```
$ strings -t d kcore > kcore_strings
$ md5sum kcore_strings > kcore_strings.md5
```

The `grep` tool and regular expressions are important in this initial analysis. In a few minutes we can find the evidence of an intrusion. We have to think what kind of data we are going to look for -- for instance, are we looking for commands typed by an intruder, IP addresses, passwords or even decrypted part of malicious code?

Below there are some examples of keywords to look for. We use these to find evidence of intrusion in our `kcore_strings` file.

- host name or prefix of the compromised system

```
$ grep "root@manhunt" kcore_strings
$ grep "]"# kcore_strings

11921096 [root@manhunt]#
16643784 [root@manhunt root]#
30692969 ]#]
```

In the above result we have listed some of the offsets of the strings. The next step is to open the file with a text editor (such as the `less` tool) and jump into a place in the file, which is close to the direct offset address. If we have some luck, we will find other commands that were run in the past. But we have to remember that

pages of virtual memory in physical memory and the swap space are written in an unorganized manner, and therefore our conclusions could also be completely invalid.

```
$ less kcore_strings
/11921096

11921096 [root@manhunt]#
11921192 /usr/bin/perl
11921288 perl apache_mod_exploit.pl
...
```

The above example presents some of the commands typed on the compromised system.

- file and directory names

```
$ grep -e "\/proc\/" -e "\/bin\/" -e "\/bin\/.*?sh" kcore_strings
$ grep -e "ftp" -e "root" kcore_strings
$ grep -e "rm -" kcore_strings
$ grep -e ".tgz" kcore_strings
```

- ip addresses and domain names

```
$ grep -e "[0-9]\+\.[0-9]\+\.[0-9]\+\.[0-9]\+" kcore_strings
$ grep -e "\.pl" kcore_strings
```

2.7 Advanced forensic analysis

In this final chapter I will show how to analyze copied kcore file by the gdb tool. At the beginning I describe how to check if system call addresses are proper in the syscall table. Changing system call addresses is the easiest method of installing a LKM based rootkit on the compromised system. In the next example I show how to list all processes run on the compromised system in the past. The results from this examples can be compared with results received in steps 7 and 8.

To start analysis we need the following information:

- the memory image in the ELF core format (see step 6)
- a compiled kernel image (we have to mount file system images of the compromised system and copy a file vmlinux-* from /boot directory)
- a list of exported symbols (a System.map file can be found in /boot directory of the compromised system or we can use the ksyms file - see step 7)
- a source code of kernel used on the compromised machine (if the source code is not available on the compromised machine we have to download the same version from the following web site: www.kernel.org)

The next step is to run the gdb tool as follows:

```
#gdb vmlinux kcore
GNU gdb Red Hat Linux (5.1.90CVS-5)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...

warning: core file may not match specified executable file.
Core was generated by `ro root=/dev/sda2 hdc=ide-scsi'.
#0 0x00000000 in ?? ()
(gdb)
```

Now we are ready to start our analysis.

Example one: Verification of system call addresses

Our first step is to find address of the system call table (`sys_call_table`). Almost in every Linux operating system this information is exported and can be found in the `Symbol.map` file.

```
# cat Symbol.map | grep sys_call_table
c02c209c D sys_call_table
```

Now, we can list entries from the `sys_call_table`. Each entry is the address of system call. To proper interpret entries I recommend to look at the `entry.S` file from the kernel source code of the compromised system. This file contains the proper order of system calls.

```
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall) /* 0 - old "setup()" system call*/
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open) /* 5 */
    .long SYMBOL_NAME(sys_close)
    ...
```

The same order is in the `sys_call_table`. To list first 10 addresses of system calls we do as follows:

```
(gdb) x/10x 0xc02c209c
0xc02c209c <sys_call_table>: 0xc01217c0 0xc011ac50 0xc0107510 0xc0138d50
0xc02c20ac <sys_call_table+16>: 0xc0138e50 0xc0138880 0xc01389b0 0xc011b010
0xc02c20bc <sys_call_table+32>: 0xc0138930 0xc01445c0
```

The address 0xc01217c0 is the sys_ni_call system call, the address 0xc011ac50 is the sys_exit system call, etc.

If we trust addresses from the System.map or the ksyms files we can compare almost every address. Of course the intruders don't change every address of system calls, there are a few popular such as a sys_read, sys_getdents or sys_write.

Example two: List of active processes

To create the whole list of run processes we have to find the address of the init_task_union struct. This struct points to first process descriptor which is called "process 0" or "swapper".

```
# cat Symbol.map | grep init_task_union
c02da000 D init_task_union
```

The next step is to find out how looks this structure. The example of init_task struct can be found in header file sched.h in source code of kernel version of the compromised system. The struct task_struct can also be found in this header file.

```
#define INIT_TASK(tsk)
{
    state:      0,
    flags:      0,
    sigpending: 0,
    addr_limit: KERNEL_DS,
    exec_domain: &default_exec_domain,
    ...
    run_list:   LIST_HEAD_INIT(tsk.run_list),
    time_slice: HZ,
    next_task:  &tsk,
    prev_task:  &tsk,
    p_opptr:    &tsk,
    ...
}
```

For us the most important fields are the prev_task and next_task of each process descriptor (the task_struct type). They help us to create the processes list. The next_task field points to the process descriptor of the next process, the prev_task field points to the process descriptor inserted last in the list.

In the example below I list fragments of process descriptor (the task_struct task), which was found at address: 0xc514c000.

```
(gdb) x/180x 0xc514c000
...
0xc514c040:    0x00000000                0xffffffff                0x00000004                0xc1be0000
0xc514c050:    0xc4dac000                0xc5ea8e40                0xc5ea8e40                0xc02c56d4
```

```

...
0xc514c070:      0x00000001          0x00001ace         0x00001ace         0x00000000
...
0xc514c230:      0xffffffff          0xffffffff          0x61620000         0x00006873
0xc514c240:      0x00007974          0x00000000          0x00000000         0x00000000
...

```

where:

0xc1be0000 is the next_task

0xc4dac000 is the prev_task

0x00001ace is the process PID = 6268 in decimal format

0x61620000 and 0x00006873 is the name of process (the comm table in the struct) = "bash" in this particular example

To print the process descriptor of the next process (the task_struct type) we do as follows:

```
(gdb) x/180x 0xc1be0000
```

Taking information from each process descriptor we can build the full list of active processes.

3. Summary

The goal of this article was to present some of the methods used for data collection, basic and advanced analysis. In particular, I have focused on the data which would be completely lost after switching off a system. This part of a forensic analysis and incident response process is very difficult to perform; we have to be very careful during all steps, and also ensure that we document everything. We have also seen some of the advantages and the disadvantages of the software collecting procedure of a live system. Finally, while I have shown how the collection procedure looks in Linux operating system, we can perform almost the same steps on other operating systems as well. A future article will show you how to gather volatile data from a live Windows operating system.

References

- Adore-ng rootkit, <http://stealth.7350.org/rootkits>
- Alessandro Rubini, Jonathan Corbet. Linux Device Drivers, 2nd Edition. O'Reilly; 2001.
- Dan Farmer, Wietse Venema. Column series for the Doctor Dobb's Journal. <http://www.porcupine.org/forensics/column.html>.
- Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 2nd Edition. O'Reilly; 2002.
- Kernel source code. <http://www.kernel.org>
- Linux manual pages.
- National Institute of Standards and Technology. Computer Security Incident Handling Guide. <http://csrc.nist.gov>.
- PHRACK #61. Finding hidden kernel modules (the extrem way) by madsys. <http://www.phrack.org>.
- RFC 3227. Guidelines for Evidence Collection and Archiving.
- Smith Fred, Bace Rebecca. A guide to forensic testimony. Addison Wesley; 2003.
- Symantec Corporation. CodeRed Worm. <http://securityresponse.symantec.com>.

- The HoneyNet Project. Scan 29. <http://www.honeynet.org>
- The SANS Institute. Incident Handling step by step. <http://www.sans.org>

About the author

View [more articles](#) by Mariusz Burdach on SecurityFocus.

Credits

Thanks to Kelly Martin and Dan Hanson for their suggestions, careful reviewing and help.

[Privacy Statement](#)

Copyright 2006, SecurityFocus