Hands-on Labs

Hands-on Labs of Memory Analysis *August 25th, 2015*

1 Lab Overview

The goal of this project is to get the hands on experience on virtual machine introspection — analyzing memory dump and reconstruct guest abstraction. There will be two tasks: one is to use `crash` tool to introspect a Linux kernel memory dump, and the other is to use `volatility` to inspect a Windows memory snapshot. This lab requires a virtual machine installed in your environment. Please first download the VM that has installed everything you need for this lab at `http://www.utdallas.edu/~zhiqiang.lin/file/mem-analysis-vm.tar.gz`. Note that you can execute the VM by using VMware Player (which is free), or Virtual-box (which is open source).

2 Linux memory introspection w/ Red-Hat crash utility

Red-hat `crash`[1] utility is a tool that allows you to analyze the kernel dumps or physical memory snapshots. We have captured the memory snapshot of a Linux kernel memory and installed the crash tool in the VM. (Note that this crash tool is a modified version we modified in our research). Please first run the help command to understand how to run crash tool as shown below:

```
root@debian:~/crash# ./run-crash.sh

crash 4.1.2
Copyright (C) 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009  Red Hat, Inc.
Copyright (C) 2004, 2005, 2006  IBM Corporation
Copyright (C) 1999-2006  Hewlett-Packard Co
Copyright (C) 2005, 2006  Fujitsu Limited
Copyright (C) 2006, 2007  VA Linux Systems Japan K.K.
Copyright (C) 2005  NEC Corporation
Copyright (C) 1999, 2002, 2007  Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002  Mission Critical Linux, Inc.
This program 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.  Enter "help copying" to see the conditions.
This program has absolutely no warranty.  Enter "help warranty" for details.

GNU gdb 6.1
Copyright 2004 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 "i686-pc-linux-gnu"...

crash: cannot set context for pid: 8257
      KERNEL: ./vmlinux-2.6.18sa
```

[1] `http://people.redhat.com/anderson/crash_whitepaper/`

```
    DUMPFILE: /tmp/crash/mem
        CPUS: 1
        DATE: Wed Jan 27 14:19:01 2010
      UPTIME: 2 days, 02:47:14
LOAD AVERAGE: 0.22, 0.07, 0.02
       TASKS: 92
    NODENAME: hope
     RELEASE: 2.6.18sa
     VERSION: #1 SMP Wed Jan 6 00:41:44 EST 2010
     MACHINE: i686  (2127 Mhz)
      MEMORY: 255.9 MB
         PID: 0
     COMMAND: "swapper"
        TASK: c035dc00  [THREAD_INFO: c0426000]
         CPU: 0
       STATE: TASK_RUNNING (ACTIVE)

crash> help

*              files          mod            runq           union
alias          foreach        mount          search         vm
ascii          fuser          net            set            vtop
bt             gdb            p              sig            waitq
btop           help           ps             struct         whatis
dev            irq            pte            swap           wr
dis            kmem           ptob           sym            q
eval           list           ptov           sys
exit           log            rd             task
extend         mach           repeat         timer

crash version: 4.1.2    gdb version: 6.1
For help on any command above, enter "help <command>".
For help on input options, enter "help input".
For help on output options, enter "help output".

CRSEOF
crash>
```

You can observe that `crash` tool actually integrates with `gdb`, and then you can use many of the `gdb` command to examine the memory as you wish. Some useful tools you might want to try include `ps` that lists the running process in the snapshot, `task` that shows the current process' `task_struct`, `foreach` that can iterate certain particular type of data structure and show the field of your interest.

```
crash> task
PID: 0      TASK: c035dc00  CPU: 0   COMMAND: "swapper"
struct task_struct {
  state = 0,
  thread_info = 0xc0426000,
  usage = {
    counter = 2
  },
  flags = 8192,
  ptrace = 0,
  lock_depth = -1,
...


crash> foreach task -R pid
PID: 0      TASK: c035dc00  CPU: 0   COMMAND: "swapper"
  pid = 0,
```

```
PID: 0      TASK: c035dc00  CPU: 0   COMMAND: "swapper"
  pid = 0,

PID: 1      TASK: c12f1630  CPU: 0   COMMAND: "init"
  pid = 1,

PID: 2      TASK: c12f10b0  CPU: 0   COMMAND: "migration/0"
  pid = 2,

PID: 3      TASK: c12f0b30  CPU: 0   COMMAND: "ksoftirqd/0"
  pid = 3,

...

crash> foreach bt
PID: 0      TASK: c035dc00  CPU: 0   COMMAND: "swapper"
(active)

PID: 1      TASK: c12f1630  CPU: 0   COMMAND: "init"
 #0 [c12e1b00] schedule at c02fc6cb
 #1 [c12e1b64] schedule_timeout at c02fcefb
 #2 [c12e1b94] do_select at c01782c5
 #3 [c12e1e38] core_sys_select at c01785ff
 #4 [c12e1f78] sys_select at c0178bed
 #5 [c12e1fb8] sysenter_entry at c0103dc6
    EAX: 0000008e  EBX: 0000000b  ECX: bfe65710  EDX: 00000000
    DS:  007b      ESI: 00000000  ES:  007b      EDI: bfe65840
    SS:  007b      ESP: bfe656d0  EBP: bfe659d8
    CS:  0073      EIP: b7fd6410  ERR: 0000008e  EFLAGS: 00000246

PID: 2      TASK: c12f10b0  CPU: 0   COMMAND: "migration/0"
 #0 [c12e4f50] schedule at c02fc6cb
 #1 [c12e4fb4] migration_thread at c011dc55
 #2 [c12e4fd0] kthread at c0131f3d
 #3 [c12e4fe8] kernel_thread_helper at c0102003

...

crash> task_struct -o
struct task_struct {
     [0] volatile long int state;
     [4] struct thread_info *thread_info;
     [8] atomic_t usage;
    [12] long unsigned int flags;
    [16] long unsigned int ptrace;
    [20] int lock_depth;
    [24] int load_weight;
    [28] int prio;
    [32] int static_prio;
    [36] int normal_prio;
    [40] struct list_head run_list;
```

Exercises Please first try to be familiar with `crash` tool, and then use this tool to answer the following questions.

Q (1) (**Processes**). How many processes in total in this memory snapshot (when you run `ps` command)? How many `vi` processes (you can execute `ps|grep vi|wc` to report this)? How many of them share the same CR3 (page global directory)? Do those kernel threads (e.g., `migration/0`, `ksoftirqd/0`, `kpsmoused`) have the value in page global directory

(i.e., CR3)? (Hint: you could traverse CR3 from `task_struct -> mm -> pgd`)

Q (2) (**Files**). What are the files opened by `syslogd` process? (Hint: you could run `foreach files` to see the openning files by all the process). What are the processes that open the files in `/etc` directory (Hint: you can execute `foreach files -R /etc` to answer this question).

Q (3) (**Network Connection**).Which process has the open sockets? (`foreach net`). What are their socket types? (Please look at field `FAMILY:TYPE`).

Q (4) (**Kernel Objects**).Kernel objects are usually allocated in a pool by certain allocator (e.g., slab or slub allocator). How many `task_struct` get allocated in the slab allocator in the given memory snapshot? (Hint: `kmem -s |grep task_struct` may help you answer this question). What about `mm_struct`?

Q (5) (**Devices**). How many devices are connected this computer when taking the snapshot? (Hint: `dev` command will help you). How many of them are character device (with type CHRDEV), and how many of them are block device (with type BLKDEV)?

Q (6) (**Virtual Memory**). For `init` process (with pid 1), how many virtual memory area are not actually mapped in the memory? (Hint: look at the result from `foreach vm -R` will help you answer this question). By looking at the virtual memory mapping of the `vi` processes, what you learn? (e.g., how the memory is layed out, and where is the library space) Below is the virtual memory mapping for `syslogd` process. There are some gaps between the library code, what they are? (e.g., Are they data sections of these binary code)?

```
PID: 1231   TASK: cf6ef810  CPU: 0   COMMAND: "syslogd"
   MM       PGD      RSS    TOTAL_VM
cf5e2c60  c1379000   704k    1788k
  VMA       START      END    FLAGS  FILE
c1307374  45388000  453a1000    875  /lib/ld-2.5.so
cf0bd95c  453a1000  453a2000 100871  /lib/ld-2.5.so
ce92b8b4  453a2000  453a3000 100873  /lib/ld-2.5.so
cf0bd128  80000000  80008000   1875  /sbin/syslogd
cf64a614  80008000  80009000 101873  /sbin/syslogd
cf7e1c50  80009000  8002a000 100073
c1307a04  b7dd3000  b7de2000     75  /lib/libresolv-2.5.so
cf0bd668  b7de2000  b7de3000 100071  /lib/libresolv-2.5.so
cf7b62cc  b7de3000  b7de4000 100073  /lib/libresolv-2.5.so
c1307da0  b7de4000  b7de6000 100073
cf80b668  b7de6000  b7dea000     75  /lib/libnss_dns-2.5.so
cf0bd2cc  b7dea000  b7deb000 100071  /lib/libnss_dns-2.5.so
c1330f98  b7deb000  b7dec000 100073  /lib/libnss_dns-2.5.so
cf7b6224  b7dfa000  b7dfb000 100073
cf64a95c  b7dfb000  b7f32000     75  /lib/libc-2.5.so
c13ac614  b7f32000  b7f34000 100071  /lib/libc-2.5.so
cf0bda58  b7f34000  b7f35000 100073  /lib/libc-2.5.so
cf80bb00  b7f35000  b7f38000 100073
cf0bd9b0  b7f3a000  b7f43000     75  /lib/libnss_files-2.5.so
c139ab00  b7f43000  b7f44000 100071  /lib/libnss_files-2.5.so
c131c4c4  b7f44000  b7f45000 100073  /lib/libnss_files-2.5.so
ced43278  b7f45000  b7f47000 100073
cf7e15c0  b7f47000  b7f48000     75
ce92b710  bf914000  bf929000 100173
```

3 Windows memory forensics w/ Volatility

Often times, one of the first steps for diagnosing a potential intrusion incident is backing up a RAM image (since RAM often contains important traces, such as information on running processes or active network connections), and then analyze the volatile memory. Volatility[2] is such an analysis framework.

In this lab, you will be asked to use volatility to analyze a memory dump that contains hidden malicious process. Note that volatility is open source, and it has been set up in the analysis VM (`http://www.utdallas.edu/~zhiqiang.lin/file/mem-analysis-vm.tar.gz`), as shown below. The to be analyzed memory dump (i.e., `hidden_process.img`) is also installed in the VM.

```
root@debian:~/volatility-2.4# vol.py -h
Volatility Foundation Volatility Framework 2.4
Usage: Volatility - A memory forensics analysis platform.

Options:
  -h, --help            list all available options and their default values.
                        Default values may be set in the configuration file
                        (/etc/volatilityrc)
  --conf-file=/root/.volatilityrc
                        User based configuration file
  -d, --debug           Debug volatility
  --plugins=PLUGINS     Additional plugin directories to use (colon separated)
  --info                Print information about all registered objects
  --cache-directory=/root/.cache/volatility
                        Directory where cache files are stored
  --cache               Use caching
  --tz=TZ               Sets the timezone for displaying timestamps
  -f FILENAME, --filename=FILENAME
                        Filename to use when opening an image
  --profile=WinXPSP2x86
                        Name of the profile to load
  -l LOCATION, --location=LOCATION
                        A URN location from which to load an address space
  -w, --write           Enable write support
  --dtb=DTB             DTB Address
  --shift=SHIFT         Mac KASLR shift address
  --output=text         Output in this format (format support is module
                        specific)
  --output-file=OUTPUT_FILE
                        write output in this file
  -v, --verbose         Verbose information
  -g KDBG, --kdbg=KDBG  Specify a specific KDBG virtual address
  -k KPCR, --kpcr=KPCR  Specify a specific KPCR address

        Supported Plugin Commands:

                apihooks        Detect API hooks in process and kernel memory
                atoms           Print session and window station atom tables
                atomscan        Pool scanner for atom tables
...

root@debian:~/windows# ll
total 262408
-rw-r--r-- 1 root root 268435456 Jul  6  2010 hidden_process.img
-rw-r--r-- 1 root root       153 Aug 31 13:07 README
```

[2] http://www.volatilityfoundation.org/#!24/c12wa

There are many plugins for either Windows or Linux memory forensics inside the volatility. In this task, you are asked to use some of them to find the hidden process. In particular, there are three plugins: pslist, psscan, and psxview, that would be of your special interest. `pslist` walks the operating system's list of processes, `psscan` does a brute force scan for process objects, and `psxview` finds the hidden processes. Any process found by the scan which isn't found by the walk is unusual, most likely hidden processes. Let's give a try on these plugins.

```
root@debian:~/windows# vol.py pslist -f hidden_process.img
Volatility Foundation Volatility Framework 2.4
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------
0x819cc830 System                    4      0     51      254 ------      0
0x817e4670 smss.exe                360      4      3       19 ------      0 2008-11-26 07:38:11
0x8181bd78 csrss.exe               596    360     10      322      0      0 2008-11-26 07:38:13
0x8182b100 winlogon.exe            620    360     16      503      0      0 2008-11-26 07:38:14
0x8183ba78 services.exe            672    620     15      245      0      0 2008-11-26 07:38:15
...

root@debian:~/windows# vol.py psscan -f hidden_process.img
Volatility Foundation Volatility Framework 2.4
Offset(P)          Name                PID   PPID PDB        Time created
------------------ ---------------- ------ ------ ---------- ------------------------------
0x000000000181b748 alg.exe             992    660 0x08140260 2008-11-15 23:43:25
0x0000000001843b28 wuauclt.exe        1372   1064 0x08140180 2008-11-26 07:39:38
0x000000000184e3a8 wscntfy.exe         560   1064 0x081402a0 2008-11-26 07:44:57
...
root@debian:~/windows# vol.py psxview -f hidden_process.img
Volatility Foundation Volatility Framework 2.4
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- --------
0x01a2b100 winlogon.exe            620 True   True   True     True   True  True    True
0x01a3d360 svchost.exe             932 True   True   True     True   True  True    True
...
```

Q (1) (**pslist**). How many processes in total in this memory snapshot (when you run `pslist` command)?

Q (2) (**psscan**). How many processes in total in this memory snapshot (when you run `psscan` command)?

Q (3) (**psxview**). How many of the process give the false result to `pslist`, and how many of them for `psscan`? Please list these processes in greater details.

Q (4) (Robust signature). You may wonder why `psxview` can even detect the hidden process that is not show to both `pslist` and `psscan`. The reason is `psxview` uses a robust kernel object data structure signatures to find out the hidden process. Please identify which process it is. Some info about this robuster scanner can be found at `http://moyix.blogspot.com/2010/07/plugin-post-robust-process-scanner.html`.

Q (5) (Binary extraction). Volatility also provides a `procdump` plugin to extract the binary code of a given process, as shown below. How many bytes you observed of the extracted binary file?

```
root@debian:~/windows# vol.py -f hidden_process.img procdump -o 0x01a4bc20 --dump-dir=.
Volatility Foundation Volatility Framework 2.4
Process(V) ImageBase  Name                 Result
---------- ---------- -------------------- ------
```

```
0x8184bc20 0x00400000 network_listene      OK: executable.1696.exe
root@debian:~/windows# ls
executable.1696.exe  hidden_process.img  README
root@debian:~/windows# file executable.1696.exe
executable.1696.exe: PE32 executable for MS Windows (console) Intel 80386 32-bit
```

Q (6) (**Other Plugins**). There are many other plugins that might be of your interest. For instance, `pstree`, and `deskscan`. Please describe which processes have the parent PID 672? Also, what you found when you execute `deskscan` plugin?