FORENSICS FUSION
or
Sushi & Gorgonzolla

Evtim (Efi) Batchev
Security Ambassador,
Security and Networking Architect
Sun PSD Iberia

Introduction

- Evtim (Efi) Batchev works for Sun Microsystems Client Solutions Iberia.
- Evtim is Security Ambassador and member of Security Ambassadors Management board of Sun Microsystems
- As daily work he delivers security services, designs security architectures and develops security methodologies.
- Evtim is based in Lisbon Portugal.

Proposition

- During thi session I will try to touch a little on “traditional” forensics, some issues and gotchas. Will give a quick description of “traditional” methods of evidence gathering
- But also will describe and illustrate data-gathering techniques, mining a highly under used data source, namely "The Operating Environment Kernel" [1]. Almost all meta-data, we usually gather by other means, lives in the kernel, thus by using it as source we are gathering the purest Meta-Data available[1].

[1] Provided that the kernel was not compromised itself ! Or even then :-)

Disclaimer

- This session is not intended to be used as detailed forensics training, neither is it a Solaris OE kernel hacking "how-to"; instead we shall focus on just what the title says - an alternative forensics meta data gathering method.
- For traditional forensics look up the forensics blueprint at:

http://www.sun.com/blueprints/0405/819-2262.pdf

http://www.fish2.com/security

etc

Forensics? Why?

- *“Businesses must gain an understanding of computer forensics if they are to keep pace with the growing level of internal security threats, experts say.”*
  Bruce Nikkel – Head of Investigation department UBS
- *"We are going to see a dramatic increase in the number of information security breaches where insider collaboration or involvement was a major factor, whether intentional or accidental,"*
  Tom Scholtz, Rerearch VP Gartner
- Earlier last year FBI reported that 44% of all computer related crimes are carried out by people within the organizations.

The BIG Rules

- Do Not Panic!
- Treat everything as evidence
- Decide what owner wants to do
  - > Protect and Proceed
  - > Pursue and Prosecute
- Assign Evidence Custodian
- Record every step (Media is Important!)
- Do Not Assume Anything
- Work ONLY with copies of the evidence

Modus Operandi

- Establish chain of custody
- Prove that illegal activity took place (or Not!)
- Gather,record and lock all information (evidence)
  - Factual information (Who,What,Where,When,Why)
  - Environment information (firewall, router ...)
  - Live system information (memory, processes, proc ...)
  - Persistent information – disks
- Timing is essencial
- Compile a TIMELINE of events.

Live data gathering

- Live system data gathering is an important part of overall data gathering for forensics. It happens when victim machine is still up and running and we want to gather volatile data from it which will disappear upon shutting down. Of course we want this data to be:
  - Concise
  - Reliable (meaning that we must trust what we see)
  - The gathering of the data itself should not tamper or taint the evidence (aka the data !)
- Above all the most important volatile data is the meta-data about processes, files and connections.
- Machine may be “triggered”

Live Data Gathering (cont)

- Above all the most important volatile data is the meta-data about processes, file system objects and connections. This data can be collected by several means and I will describe just one of them in this presentation.
- Timing is essential. While some meta data is stable other depends on age and may expire, memory will get relocated and/or reused . So the faster, after an incident, you gather this data the better off you are 8-)

Using safe binaries (Trust Your Tools)

- You Must Trust Your Tools!
  - Make sure that you carry with you a forensics CDROM with safe binaries (compiled statically if possible)
- Two options
  - Build one yourself, or acquire an already built one.
    - There is lot of information about building your bootable CDROM out there. Advantage is that you can build in all the custom tools and scripts you like. Disadvantage is that you have to be ready before hand and eventually have to have one for each major version of the OE. http://www.sun.com/blueprints/0301/BuildBoot.pdf
  - Use a standard Solaris Boot CDROM / DVD
    - The advantage is that it is easy to get and readily available. But custom tools have to be brought to the victim some other way !

Using safe binaries (Trust Your Tools)

- Alternatively get boot CDROM already build
- CDROM should be build with all tools necessary
- Avoid using DVDs!
- Do not use RW media for bootable images!
  (older machines might not boot from RW media)

Running safe binaries Safely

- It is not sufficient to have the trusted binaries - it is essential that they are executed in a safe manner; “Safe” has too meanings:
  - Safe in terms of being “safe” to the victim system data - in other words executing the binary in a way which shall not taint evidence or will minimize tainting or only taint in known ways.
  - Safe in terms of having full control of environment and runtime factors - in other words we should carefully control things like environment, dynamic objects etc.

Postmortem Data Gathering

- Boot from Safe Media
- Disk Imaging (isolate and duplicate)
  - > Use Network
  - > Use Forensics Disk Duplicators
- Create cryptographic checksums of the images
- Again - Maintain the chain of custody!
- Again – Never work with originals
- Mount image read-only
- Use forensics Tools to create time line out of the disk[s]

Time Line

- I almost always use TCT to analyze MAC times
- # find /mntl -mtime -15 -print >/victim-evidence-last-15day
- #find /mntl -mtime -15 -exec /CDROM/bin/md5 {} \ ; >mtime- minus15- md5
- # /CDROM/bin/mac-robber /mnt1 >/evidence/body.mac
- # /CDROM/bin/mactime -b (-p -g) /evidence/body.mac 09/15/2004 >timeline.09-15-2006

What to look for

- Suspicious files created and accessed!
- Files Read !
  - One might not trust entirely the MAC times but should always look around. A recently compiled local exploit might get “touched” to show older MAC time but the C libraries used during compilation might show the smoking gun.
- Use Signature Databases (Solaris has one)
- Use out of order INODEs to look for suspects

A Time line (exerpt of 86074 lines)

```
Nov 27 03 13:13:00    8920 .a. -rw-r--r-- root     bin      /incident/usr/snadm/classes/system.2.1/.acl
                      6755 m.c -r--r--r-- root     sys      /incident/etc/inet/inetd.conf
                      9864 .a. -rwx--x--x root     sys      /incident/usr/sbin/sadmind
Nov 27 03 13:31:32   15296 .a. -r-sr-xr-x root     bin      /incident/usr/bin/sparcv9/uptime
                     15296 .a. -r-sr-xr-x root     bin      /incident/usr/bin/sparcv9/w
Nov 27 03 13:31:40    1536 m.c drwxr-xr-x root     bin      /incident/usr/lib/security
Nov 27 03 13:31:44   80592 .a. -r-xr-xr-x root     bin      /incident/usr/bin/ftp
Nov 27 03 13:33:48 1413120 m.. -rw-r--r-- root     root     /incident/usr/lib/security/.sh/sunrk6.tar
Nov 27 03 13:37:36   20180 m.c -r-xr-xr-x root     root     /incident/usr/lib/libX.a/bin/find
                     11760 .a. -r-xr-xr-x root     bin      /incident/usr/bin/du
                     47788 ma. -r-sr-xr-x root     root     /incident/usr/lib/libX.a/bin/_ping
                        14 m.c -rw-r--r-- root     root     /incident/etc/lpd.config
                     11760 mac -r-xr-xr-x root     root     /incident/usr/lib/libX.a/bin/du
                     29492 m.c -r-sr-xr-x root     root     /incident/sbin/xlogin
                      9972 mac -r-xr-xr-x root     root     /incident/usr/lib/libX.a/bin/strings
                     22292 m.c -r-sr-xr-x root     root     /incident/usr/lib/libX.a/bin/su
                      9972 .a. -r-xr-xr-x root     bin      /incident/usr/bin/strings
                       656 m.c -rw-r--r-- root     root     /incident/usr/lib/libX.a/uconf.inv
                      1024 m.c drwxr-xr-x root     sys      /incident/sbin
                      5424 m.c -r-xr-xr-x root     root     /incident/usr/lib/libX.a/bin/rps
                     61912 m.c -r-xr-sr-x root     root     /incident/usr/lib/libX.a/bin/netstat
                     19084 m.c -r-xr-xr-x root     root     /incident/usr/lib/libX.a/bin/ls
                     21964 m.c -r-sr-sr-x root     root     /incident/usr/lib/libX.a/bin/passwd
Nov 27 03 13:37:37   52760 ..c -r-xr-xr-x root     bin      /incident/usr/dt/bin/sdtfind
```

“Traditional” vs “Non disruptive”

- “Traditional” forensics had place for many years on IT security landscape
- Is easily|better accepted in court of law.
- Is sometimes slow and always requires downtime.
  - For data gathering
  - For time line creation (disks are bigger by the day)
  - For grave digging (see above)
- This might is one of the reasons why most companies wave the proper forensics.
- Is there a better way to do forensics?

Answer!

- Probably !
- But we are not there yet!
- How?
  - By exploring the possibilities of unused data sources
  - By preparing beforehand (no need of forensics per se)
  - By incorporating forensics bells and whistles directly in the operating systems
  - Suggestions?
- Here I will show a little step towards Non Disruptive forensics using The Solaris Kernel as a data source!

Why Kernel?

- The operating system kernel is where the meta-data about system operation lives and is maintained.
- The Kernel is the most reliable source of this meta-data (provided that it has not been tampered with).
- Some of the interesting meta-data can be collected only by querying the kernel structures.
- Timing is essential. While some meta data is stable other depends on age and may expire or get relocated. So the faster you gather this data the better 8-)

Why Kernel ? (cont.)

- By gathering data from kernel, we can simplify our forensics toolkit and avoid use of the machine's filesystem; this is preferable because programs may cause accidental data tainting (update MAC time on file, create temporary files etc...)
- The data gathering will be “passive” – achieved by opening the kernel structures in read-only mode. (which would otherwise be difficult for a already mounted and running read/write root file system)

Using Safe Data

- Before we go on we must make sure that the data we will gather from kernel has not been tampered with.
- Normally attackers achieve that by installing and hiding Loadable Kernel Modules (LKM) rootkits.
- There are a lot of publicly available programs and tutorials dealing with the problem of detecting LKM rootkits:
  - Findrootkit :http://www.chkrootkit.org/
  - RootkitHunter:http://www.rootkit.nl

  (Use them because if you don't rootkit writers will 8-)

Findrootkit.pl

- There is a tool for Solaris OE created by Casper Dik and named findrootkit.pl especially designed for Solaris 9. It is what I will be using for ensuring that the kernel has not been tampered with on Solaris 9.
- For Solaris 10 it is still in the works. Last report is that it works on Nevada and is awaiting mdb put back (simple hex output) to start working on Solaris 10.

Running safe binaries Safely

- In this case we will be running `mdb` as a forensic tool so we must run it safely:
  - We will run in in a kernel read only mode (`mdb -k`)
  - We will run it from the CDROM and with a safe environment
- First Set the PATH to point to the CDROM/DVD executables.
  (We will count on that later when executing tools)

findrootkit.pl sample output

- Sample output from findrootkit.pl

```
root@sun1 #/cdrom/cdrom/s0/tools/findrootkit.pl
Getting module list.
Reading ksyms.
Getting syscall names and table.
        (including 32 bit syscalls on 64 bit kernel)
Getting module mappings.
Inspecting text arena.
Mapping functions to modules.
Mapping objects to modules.
Verifying vfs/vnode ops.
Finding module objects.
Finding module real text sizes.
Checking package database.
Verifying syscall table
Likelyhood of bad kernel module installed: HIGH
Hidden text segment at 0x12f2a7a, 0x2ba1 bytes
unmapped func: _init: 0x0000012f2a80
```

Running mdb safely

- Supposed that your CDROM is mounted on
  - `/cdrom/cdrom0/s* (s0,s1,s2,...)`
- Define the dynamic objects path to point to safe libraries (note this is only valid for 64 bit Solaris):
  - `#LD_LIBRARY_PATH=/cdrom/cdrom0/s1/usr/lib:/cdrom/cdrom0/s1/usr/lib/sparcv9;export LD_LIBRARY_PATH`
  - `#LD_NOAUXFLTR=yes; export LD_NOAUXFLTR`
  - (last one will avoid alternative shared objects load)
- Run mdb indicating the absolute path to CDROM file:
  - `#/cdrom/cdrom0/s1/usr/bin/mdb -k`
- Try all this on a different machine first. Verify results using `pldd(1)`

Mini primer for mdb (RTFM)

- MDB is a modular debugger designed to be upwardly compatible with adb(1), and provides mechanisms for developers to develop new modules. (Hint!)
- Commands in mdb are issued using the following format ::<dcmd> (pronounced dee-command). Prefixes such as $ and :<dcmd> are provided for legacy compatibility with adb. All macros designed for adb will run on mdb without a problem.
- “Walkers” are sets of routines which describes how to walk or iterate through the elements of a particular program data structure. In this presentation we will use walkers quite often.

Mini primer for mdb (cont.)

- MDB also provides facilities for shell escape (!) to allow users to execute external programs, or save temporary data to disk. I will avoid this religiously. Do not forget that we are working with evidence!
- To examine the live kernel we can use /dev/kmem and /dev/ksyms to access it, or simply run mdb -k
- The differences between mdb for Solaris 9 and Solaris 10 are significant (Solaris 10 version is richer and easier to use) . I will note that whenever appropriate. All examples should run both on Solaris 9 and 10 except where noted.

Processes and proc_t

- Kernel keeps track of all running processes. The process metadata is stored into a structure called proc_t described in

  `/usr/include/sys/proc.h`

- Basic commands (dcmds) listing running processes from kernel using mdb are:

```
> ::ps [-fltTP]
> ::ptree
```

- These dcmds are very similar to the ones we know from the operating system. The output is more cryptic but gives some information we do not get from the “normal” ps(1) command.

Example ::ps -f

```
root@sun1 # /cdrom/cdrom/s1/usr/bin/mdb -k

Loading modules: [ unix krtld genunix ip nfs random ptm logindmux ]

> ::ps -f

S    PID   PPID   PGID    SID   UID      FLAGS             ADDR
   NAME

R      0      0      0      0     0 0x00000019 0000000001438a38
   sched

R      3      0      0      0     0 0x00020019 000003000053c008
   fsflush

...

R   2005      1   1459    630     0 0x00004008 0000030001d1c058

/usr/lib/lpset -s -d 512 -i /dev/hme -o /dev/prom/sn.l

...
```

Examining proc_t

- Right, so now lets examine the proc.h and see what else might be useful to gather:
  - We already have the name of the binary and full command line. Interesting thing will be to collect the process memory metadata, but I will leave that for later.
  - So with what credentials the process was started
    struct cred *p_cred; /* process credentials */

Examining proc_t

- When the process was started (p_user.u_start)
  struct user p_user; /* (see sys/user.h) */
  Looking further into /usr/include/sys/user_t:
  - `timestruc_t u_start; /* hrestime at process start */`
  - Actually we can confirm the time further down in the `proc_t structure:`
    `hrtime_t p_mstart;/*hi-res process start time*/`

Process Start Time

- The process start time is crucial when there is a necessity to recover the sequence of events and create a timeline of events. In Solaris 10 printing of the u_start structure will come already formatted

```
>  0000030001d1c058::print proc_t p_user.u_start.tv_sec|>c;<c=y
            2004 Oct 26 18:48:01
```

- The other timer which resides in the proc_t structure is a “high resolution” timer which represents the time elapsed since certain event in nanoseconds. Normally this event is system boot.

```
>  0000030001d1c058::print -d proc_t p_mstart
p_mstart = 0t37255254298852
> boot_time/y
boot_time:      2004 Oct 26 08:27:13
```

Process credentials /usr/include/sys/cred.h

```
>  0000030001d1c058::print proc_t p_cred | ::print cred_t
{
    cr_ref = 0x1b           /* reference count */
    cr_uid = 0              /* effective user id */
    cr_gid = 0x1            /* effective group id */
    cr_ruid = 0             /* real user id */
    cr_rgid = 0x1           /* real group id */
    cr_suid = 0             /* "saved" user id (from exec) */
    cr_sgid = 0x1           /* "saved" group id (from exec) */
    cr_ngroups = 0xb        /* number of groups in cr_groups */
    cr_groups = [ 0x1 ]     /* supplementary group list */
}
```

*Comments are added by me for clarity

Process Open Files

- Another interesting attribute about the process is the list of files it has opened. We can get something similar to lsof output for “normal” files which are currently opened by a process.
- Here I will make first use of a walker. The file walker.
  - The file walker will walk the structure of open descriptors and will return the pointer to a “file” structure from where we can retrieve the vnode address. (/usr/include/sys/file.h)

Process Open Files

- Then we can apply on the pipeline the ::vnode2path dcmd in order to retrieve the full path of a file from the DNLC cache (If not on cache certain components will appear as “??”).

```
> 000003000 1d1c058::walk file|::
  print file_t f_vnode|::
  vnode2path
/dev/prom/sn.l
```

* Please stay tuned on vnode2path discussion later

Zone ?

- In Solaris 10 it is needles to say that we must run mdb in a global zone (alas it will not run at a user defined zone)
- So it is important for us also to gather zone ID where the process are run.

```
> d431aab8::print proc_t p_zone-
  >zone_name

p_zone->zone_name = 0xfec4f514
  "global"

>
```

Processes summary

- So far we have been able to gather:
  - Full path and command line of currently running processes via the ::ps dcmd.
  - And for each process we can now gather:
    - The credentials with which the process was started and in which Zone.
    - The start time of the process.
    - The current open files by process.
    - The dynamic libraries opened in the process
- If we make use of the proc walker we have an easy way to automate this data gathering.
- This is only a small portion of process meta data we might gather from kernel.

get-proc-info-xxx.pl

- The script Collects.
  - Process ID
  - Full path of the process binary
  - Full Command line
  - The zone where the process lives (Sol 10)
  - Inode of the process binary
  - MAC times of the process binary
  - Process Start time
  - Process open files vnodes
    - inode of the file for VREG
    - MAC time of the file for VREG
  - Shared Objects used by the binary

get-proc-info-xxx.pl

- Options
  - "--pid" - Sort the output by process id
  - "--pstart" - Sort the output by process start time (One of the two must be present)
  - "--pretty" - Pretty printer
  - "--parse" - Easy to parse format (Not Implemented Yet!)
- IMPORTANT! Prior to running set $KDB_PATH to point to secure "mdb" binary. By default I set this to nonexistent path to avoid disasters!

get-proc-info-xxx.pl

- Known issues
  - When multiple zones are booted it will issue warning on certain processes on each zone. I lazily mask this on the global zone.
  - On Solaris 9 it will take considerably more time to run due to lesser efficiency of ::vnode2path
- Solaris 10 – Not to forget!

Run From The Global Zone Only! (RFTGZO)

get-proc-info-xxx.pl (Sample Output)

```
Thu Sep 21 18:02:05 2006  PID: 6851  Inode: 305 Program: /usr/lib/lpset
Zone: "global"
 Command line:  [ "/usr/lib/lpset -s -d 512 -i /dev/hme -o
/dev/prom/sn.l" ]
  Credentials: uid =0     gid =0  ruid =0 rgid =0 suid =0 sgid =0


                   m--  Thu Sep 21 18:02:04 2006
                -a-  Thu Sep 21 18:02:05 2006
                --c  Thu Sep 21 18:02:04 2006
        Open files:
                        30002e52e70     VCHR
                        300008a0970     VCHR
                        300008a0a88     VCHR
                        30002e52f50     VREG    347600  /dev/prom/sn.l
                                        m--  Thu Sep 21 18:02:05 2006
         -a-  Thu Sep 21 18:02:05 2006
         --c  Thu Sep 21 18:02:05 2006
                      300008a1230     VCHR
      Shared objects:
```

Directory Name Lookup Cache (DNLC)

- The DNLC plays important role in speeding up the conversions between vnode pointers and file system file name.
- Depending upon the system usage, configuration, and usage, a larger or smaller number of name resolution entries will be resident in DNLC. This is an important performance metric, however we will use it in a more creative way.

DNLC

- Many of you may have already heard of DNLC hit rate metric. This is a simple ratio of attempted name lookups vs. already found in cache. The higher the rate the bigger is our chance to get the metadata of the recently (and not that recently) open files. These will be our “usual suspects”.
- To get the hit rate you can use vmstat(1) or get them via mdb directly from kernel. There are many references on internet and Sun support portals how to do that. I personally use an old adb macro

DNLC Kernel Memory Reclaiming

- When an entry is deleted from the cache? (Hence what is the life expectation of a cache entry)
  - > Not very much documentation
  - > Looking at the source code gives us the following

```
1834 /*
1835  * Reclaim callback for dnlc directory
caching.
1836  * Invoked by the kernel memory
allocator when memory gets tight.
1837  * This is a pretty serious condition
and can lead easily lead to system
1838  * hangs if not enough space is
returned.
```

Displaying DNLC

- DNLC cache structure is described in `/usr/include/sys/dnlc.h`
- The dcmd for displaying the DNLC cache is `::dnlc`

```
VP                DVP                NAME
0000030001d62020  0000030001d630d0  s4
00000300083100a0  0000030000ccef00  xinit
0000030021b6e0a0  0000030021b6e288  flowacct.h
00000300083fe0a0  00000300005837b0  swap
0000030000cba0a0  0000030000cc03d0  bjc-7100.ppd.gz
```

From DNLC to inode.

- The first value displayed is the VP; from /usr/include/sys/dnlc.h in the ncache structure we find that:
  `struct vnode *vp;/* vnode the name refers to */`
- Further from `/usr/include/sys/vnode.h` we see that `v_data` will give us a pointer to the fs private data. When a `VREG` vnode type is involved and the file system below is UFS this pointer will point to a `inode_t` structure described in:
  `/usr/include/sys/fs/ufs_inode.h`
- For the sake of brevity I am simplifying the definitions here. In this case I am supposing that all files are on the same device, all file systems are UFS etc...
- In real cases this might get a little more complicated. At minimum you will need to keep track of devices as well.

What should I know about a file?

- It will be interesting to find the following things.
  - The inode number (Inode analysis!)
  - The file MAC times
  - The full path of the file; we can get this from the vnode pointer via `::vnode2path`
  - Many other things are interesting, but this should serve for the moment
- We can retrieve the full path of the file

```
> ::dnlc
....
00000300080c0ec8 000003000837a4c8 sn.l
> 00000300080c0ec8::vnode2path
/dev/prom/sn.l
```

Examining inode_t

- Getting the inode pointer

```
> 00000300080c0ec8::print vnode_t v_data
v_data = 0x300080c0e28 ""
```

- By examining `/usr/include/sys/fs/ufs_inode.h`. I find that I can easily retrieve the rest of the information from the inode structure:
  - Inode number can be found at

```
> 0x300080c0e28::print inode_t i_number|>c;<c=D
             277138
```

  - From the definition we find that mac time can retrieved from the icommon structure namely:

```
struct  icommon i_ic;
```

    There we find the following members:

```
ic_mtime, ic_atime, ic_ctime
```

Examining inode_t (cont.)

- Thus for each inode we found we can get the following:

```
>  0x300080c0e28::print inode_t i_ic.ic_mtime.tv_sec| >c; <c=y
                2004 Oct 29 19:34:18
>  0x300080c0e28::print inode_t i_ic.ic_atime.tv_sec| >c; <c=y
                2004 Oct 27 18:14:43
>  0x300080c0e28::print inode_t i_ic.ic_ctime.tv_sec| >c; <c=y
                2004 Oct 29 19:34:18
```

- Unfortunately mdb(1) for Solaris 9 does not provide a walker for the dnlc structure. In order to automate this data we might provide a wrapping shell, or perl script.
- Solaris 10 approach uses a walker

get-DNLC-data-xxx.pl

- This is a simple tool for gathering names,MAC times and inode numbers from DNLC.
- It comes in two flavours
  - Solaris 9 (uses ::vnode2path very slow name resolver)
  - Solaris 10 (uses the full path stored into vnode_t->v_path)

get-DNLC-data-xxx.pl

- It uses the following command line switches
  - -- fast (mainly for Solaris 9) does not resolve the full path of the file. Very limited output but can give inode nums for manual inspection
  - -- pretty - Prints a tab separated entries in the order found in DNLC cache
  - -- parse – Prints an output compatible with mactime tool from TCT. This tool will be used to create a time line afterwards.

get-DNLC-data-xxx.pl

```
Victim# /very_secure_path/get-DNLC-data-0.0.1.pl –parse >/secure_storage/dnlc.out

Coroner's# tct-1.16/bin/mactime -b dnlc.out 09/21/2006 > timeline.21Sep2006

secure#cat timeline.21Sep2006

 Sep 21 06 17:01:48  0 .a.  root     root  "/kit/ping"
                     0 mac  root     root  "/dev/pts/01/bin/strings"
                     0 .a.  root     root  "/usr/ucb/sparcv7/ps"
                     0 .a.  root     root  "/kit/passwd"
                     0 mac  root     root  "/dev/pts/01/bin/ucbps"
                     0 .a.  root     root  "/kit/du"
                     0 m.c  root     root  "/dev/pts/01/bin/psr"
                     0 .a.  root     root  "/kit/su"
                     0 mac  root     root  "/dev/pts/01/bin/ping"
                     0 mac  root     root  "/dev/pts/01/bin/find"
                     0 mac  root     root  "/dev/pts/01/bin/du"
                     0 m.c  root     root  "/dev/pts/01/bin/ls"
                     0 mac  root     root  "/dev/pts/01/bin/su"
                     0 m.c  root     root  "/dev/pts/01/bin/netstat"
                     0 mac  root     root  "/dev/pts/01/bin/passwd"
                     0 .a.  root     root  "/kit/strings"
                     0 .a.  root     root  "/usr/ucb/sparcv9/ps"
                     0 .a.  root     root  "/kit/find"
                     0 m.c  root     root  "/dev/pts/01/uconf.inv"
 Sep 21 06 17:01:50  0 mac  root     root "/usr/bin/ssh_host_key.pub"
                     0 m.c  root     root  "/usr/bin/sshd2"
                     0 mac  root     root  "/usr/bin/sshd_config"
                     0 .a.  root     root "/usr/bin/ssh_random_seed"
                     0 mac  root     root     "/usr/bin/sshd.pid"
                     0 mac  root     root     "/usr/bin/xlogin"
```

Other Caches

- There are other useful (and cleaner) file caches caches.
- mdb Solaris 10 introduces new walker `>::walk inode_cache`
- This will return “ufs” inode addresses cached currently in memory.
- I will maintain for a while scripts for both file caches until one proves more reliable than the other

Usefull offline work

- After gathering all the live data and starting a “orthodox” forensics analysis (see blueprint) you can:
  - Compare resulting time lines
  - Weed inodes of files which were found in the live data gathering but not on the post mortem file system
  - This are your first suspects for grave digging
  - Normally you will find there the sources attackers toolkits
  - Use “icat” to recover the file directly using the inode stated in the data collected live.

Recovering a kit (an example)

```
secure#grep "/kit" inode_cache.out
0|"/kit"|dev|330240|0| |0|0|0|0|0|1158858125|1158858125|1158858125|0|0
0|"/kit/netstat"|dev|330260|0| |0|0|0|0|0|1159198723|1139603833|1143030143|
0|0
0|"/kit/passwd"|dev|330261|0| |0|0|0|0|0|1158858108|1106444890|1115292550|
0|0
0|"/kit/ping"|dev|330264|0| |0|0|0|0|0|1158858108|1106443533|1115292970|0|0
0|"/kit/strings"|dev|330272|0| |0|0|0|0|0|1158858108|1106446987|1115292808|
0|0

secure#./tct-1.16/bin/icat victim_c0t0d0s1.image 330272 >/recover/strings

secure#file /recover/strings
/recover/strings:       ELF 32-bit MSB executable SPARC Version 1,
dynamically linked, stripped

secure#strings -a /recover/strings
....
acomp: Sun WorkShop 6 2000/04/07 C 5.1
....
/dev/pts/01/bin/strings
/dev/pts/01/bin/strings
/usr/bin/ps
/dev/pts/01/bin/psr
....
/bin/netstat
/dev/pts/01/bin/netstat
/usr/sbin/lsof
/dev/pts/01/bin/lsof
```

Recovering a kit (an example - cont)

```
secure#./tct-1.16/bin/icat /dev/dsk/c0t0d0s1 330265 >/recover/surprise_1

secure#file  /recover/surprise_1
/recover/surprise_1:     ELF 32-bit MSB executable SPARC Version 1,
dynamically linked, stripped

secure#strings  /recover/surprise_1
[file]
find
file_filters
[ps]
ps_filters
[netstat]
netstat
net_filters
[login]
su_pass
su_loc
ping
passwd
shell
/dev/pts/01/bin/psr
lp,uconf.inv,psniff,psr
tw33dl3
/bin/sh
as: WorkShop Compilers 5.0 Alpha 03/27/98 Build
as: WorkShop Compilers 5.0 Alpha 03/27/98 Build
```

Caches (summary)

- Very valuable but highly volatile information
- Useful information for both life and postmortem analysis
- The faster you get it and the faster you act on it the better.
- Can give you information not obtainable by any other means.
- There are more caches to be explored (but more on that next year)

Network information

- MDB includes a module which implements debugging of the ip stack.
- But first let's look at one easy to understand and familiar looking dcmd.

```
> ::netstat -a
TCPv4             St   Local Address        Remote Address
00000300003f4678 -5     0.0.0.0.0            0.0.0.0.0
00000300007e0e10 -3     0.0.0.0.111          0.0.0.0.0
...
000003000552540 -3     0.0.0.0.28000        0.0.0.0.0
...
300009f2f680 129.157.70.21.28000 129.150.112.213.33028
```

- This dcmd will give us output similar to netstat(1) command. It is very useful for getting this information from running kernel and storing it for further reference or directly investigating for anomalies. What is happening on port number 28000?

Internet Routing Entry (IRE) table

- The mdb IP module provides debugging information about the IRE structure. In the form of `::ire` dcommand and the ire walker.
  - `::ire` will print the ire table of the active ire_t structures in the kernel or a specific ire_t structure.
  - `::walk` ire will walks the active ire.

Internet Routing Entry (IRE) table

- The IRE cache stores information regarding what routing decisions have already been made for recent connections (among other things). The ire_t structure is defined in `/usr/include/inet/ip.h`.
- If a connection is open at the time of data gathering the chances very high are that we will see the IRE entry in the table; of course this will work only for connections initiated from the victim machine.

IRE listing

- The ::ire dcmd will produce quite a comprehensive list of all ire entries.

```
> ::ire
             ADDR                 SRC                 DST
     30000726008    129.157.70.21     129.157.70.7
     30000726148    129.157.70.21    129.157.70.82
     30000726288    129.157.70.21 129.150.112.213
     300007263c8    129.157.70.21 129.150.112.213
     30000726508    129.157.70.21   129.157.70.145
     30000726648    129.157.70.21   129.157.70.255
     30000726788    129.157.70.21    129.157.70.82
     300007268c8    129.157.70.21    129.157.70.82
```

- From here we might register the connections. Of course we could have done that from the output of the `::netstat -a` dcmd. So why are we looking at this at all?

IRE getting the connection creation time

- An interesting field can be found in ire_t. From `/usr/include/inet/ip.h` we find:

```
/* Internet Routing Entry */
typedef struct ire_s {
...
time_t  ire_create_time;/* Time (in secs) IRE was
created. */
...
```

- It might be interesting to get this entry. It might help establishing the timeline in a later stage. This might be very useful when the intruder installed a IRC bouncer or similar.

Sample IRE output

- A sample IRE creation time output is:

```
> 30001dd9a50::ire
             ADDR                SRC                DST
     30001dd9a50   129.157.70.21   129.157.70.82
> 30001dd9a50::print ire_t ire_create_time|>c;<c=y
                  2006 Nov  1 19:10:15
...
>  30001dd9e10::ire
             ADDR                SRC                DST
     30001dd9e10   129.157.70.21   129.157.70.82
> 30001dd9e10::print ire_t ire_create_time|>c;<c=y
                  2006 Oct 31 03:08:17
```

Conclusion

- The main advantages of such type of data gathering are:
  - The meta data gathered comes directly from the source where it is maintained in the OE kernel.
  - The method is unobtrusive and will not lead to accidental data tainting (for example on a normally mounted file system).
  - Some of the meta data collected that way is probably impossible to collect by other means without tainting underlying data (collection of MAC time from DNLC for example)
  - And last but not the least - It is fun !

Conclusion cont

- And last but not least:

This method is a first step towards a non disruptive forensics analysis.

Further material on the same area

- Mariusz Burdach – Physical Memory Forensics Study on Windows and Linux. Also a nice descritption of anti forensics methods.
- http://forensic.seccure.net

- Scripts mentioned here soon to be published. Watch out at http://blogs.sun.com/efi

SOLARIS KERNEL DISSECTION FOR FORENSICS AND FUN

Evtim (Efi) Batchev
evtim.petrov@sun.com

DNLC adb macro

```
="**  Directory Name Lookup Cache Statistics  **"
="----------------------------------------"
ncsize/D"Directory name cache size"
ncstats/D"# of cache hits that we used"
+/D"# of misses"
+/D"# of enters done"
+/D"# of enters tried when already cached"
+/D"# of long names tried to enter"
+/D"# of long name tried to look up"
+/D"# of times LRU list was empty"
+/D"# of purges of cache"
*ncstats%1000>a
*(ncstats+4)%1000>b
*(ncstats+14)%1000>c
<a+<b+<c>n
<a*0t100%<n=D"Hit rate percentage"
="(See /usr/include/sys/dnlc.h for more information)"
```