Introduction to Quick and Dirty Forensics

Leif Nixon

So, you think you may have an incident?

How do you know you might be dealing with a security incident?

- Monitoring alarm
- External alert
- Anomalous system behaviour

Is this for real? – Incident triage

Look at things like:

- system logs
- command line histories
- `ps`
- `top`
- `netstat`
- `lsof`
- …

Do *not*:

- run `rpm -Va`
- reboot the system
- kill suspect processes
- delete malicious files
- …

Observation changes the observed object

Each time you run a command, each time you read a file, you change timestamp information. Each time you write data to disk, you might overwrite previously freed data sectors.

Do the least intrusive investigation possible.

Oh, sh⋆t!

It's real! You've been hacked!

Quick, what's the first thing you do?

Oh, sh⋆t!

It's real! You've been hacked!

Quick, what's the first thing you do?

Stop!

Take a break. Go have a cup of coffee. Or tea. Or a can of soda. (Beer is probably not a good idea, though.)

Is this really your problem?

Decision point: where do we want to go with this?

Do we want *evidence* or *leads*?

The legal route

If you think you may get good enough data to go after the intruder through the legal system, or if your policy requires that you do so, you will need evidence that will stand up in court.

This probably means that the forensic investigation should not be performed by you, but by a police technician or an outside expert.

It also means the rest of this presentation is not for you – thank you for your attention and have a nice day.

The alternative route

On the other hand, your preliminary data may indicate that it is unlikely that you will find binding evidence. Or you might have a policy that forbids you to involve law enforcement.

Basically, you just want to get back into secure service as soon as possible.

So, you just reinstall the system, right?

The alternative route

On the other hand, your preliminary data may indicate that it is unlikely that you will find binding evidence. Or you might have a policy that forbids you to involve law enforcement.

Basically, you just want to get back into secure service as soon as possible.

So, you just reinstall the system, right? *Nooo!*

Our goal

To get back into *secure* service we would like to know:

- How the intruders got in
- When they did so
- What they have been doing on the system
- What we can do to stop them from returning
- Which other sites that may have been hit

We must talk about this

One extremely important task during an incident:

Talk to people!

You need to communicate with victims, your users, management, partner sites and other security teams, and keep them all appropriately updated.

Good incident response is about good communications at least as much as about technical skills. If you are working as part of a team, you should assign one person to coordinate communications.

Quick and Dirty Forensics

A careful and thorough forensic investigation is hard to perform and takes a long time.

But often, a quick and dirty investigation may be *good enough*, or sometimes even *better*.

Make sure you are not interrupted

If possible, first check open network connections, e.g. by `netstat`. Save the output, but preferrably not on the system itself; cut-and-paste it from the terminal window to a local file.

Then isolate the system.[1] Unplug the network cable, introduce a router filter or drop a firewall in place, whatever is easiest. If this is a virtual machine, snapshot it.

[1]But skip this step in the simulated exercise – no physical access.

Now we can start in earnest – data collection

There are various types of data in the system, with widely varying expected lifetime.

Table of Order of Volatility:

Registers, peripheral memory, caches, etc.	nanoseconds
Main memory	nanoseconds
Network state	milliseconds
Running processes	seconds
Disk	minutes
Backup media, etc.	years
Printouts, etc.	tens of years

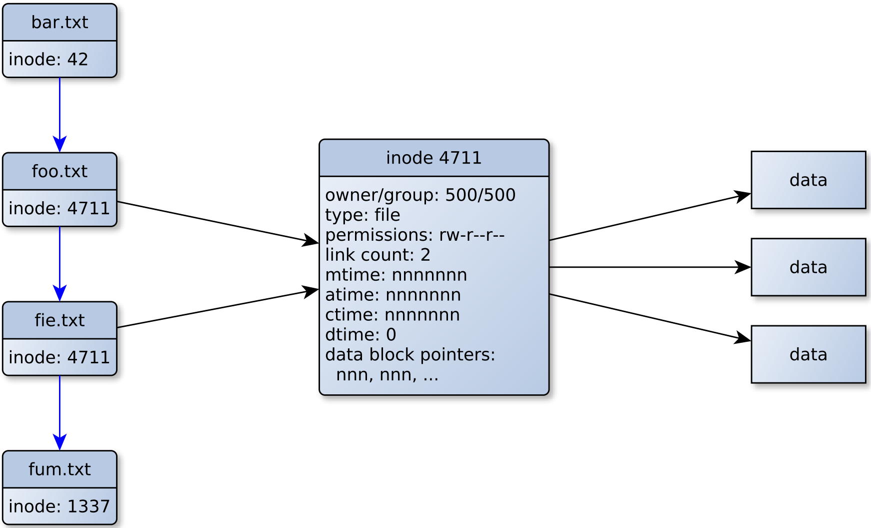
(Table borrowed from "Forensic Discovery", Farmer & Venema, Addison-Wesley 2005). You should buy this book.)

The Order of Volatility

Basically, you should follow the order of volatility when collecting data.

With one exception: filesystem timestamp data. This is often the most important data, and you want to capture it early in the investigation.

Something about filesystems



Types of timestamps

mtime – modification time; the last time the *contents* (data blocks) of a file changed

atime – access time; the last time the file was read[2]

ctime – change time; the last time one of the attributes in the inode changed

dtime – deletion time; recorded in deleted inodes (ext*n*fs)

crtime – creation time; ext4fs only

[2] Death to the `noatime` mount option!

Trustworthiness of timestamps

The mtime and atime can easily be set to arbitrary values (using `touch`, but *not* the ctime. This is sometimes very important.

(It *is* possible to change the ctime by changing the system time or directly modifying the on-disk file system with `fsdebug`, but this is a bit tricky, especially if the file system is mounted.)

Generating timelines from timestamps

By collecting and sorting timestamp data from the entire filesystem, you can sometimes gain surprising insights into past activities.

There are two basic ways to collect the data, each with their own (dis)advantages:

1. `stat` every file in the mounted filesystem
2. bypass the kernel filesystem code and dig out the data directly from the device or an image using specialized tools

Generating timelines the quick and dirty way

Collecting data from the mounted filesystem is a simple one-liner. Generating the timeline is almost as easy.

```
find / -xdev -print0 | xargs -0 stat -c "%Y %X %Z %A %U %G %n" \
    >> timestamps.dat

timeline-decorator.py < timestamps.dat | sort -n > timeline.txt
```

Generating timelines the quick and dirty way

`timeline-decorator.py`:

```
#! /usr/bin/python

import sys, time

def print_line(flags, t, mode, user, group, name):
    print t, time.ctime(float(t)), flags, mode, user, group, name

for line in sys.stdin:
    line = line[:-1]
    (m, a, c, mode, user, group, name) = line.split(" ", 6)
    if m == a:
        if m == c:
            print_line("mac", m, mode, user, group, name)
        else:
            print_line("ma-", m, mode, user, group, name)
            print_line("--c", c, mode, user, group, name)
    else:
        if m == c:
            print_line("m-c", m, mode, user, group, name)
            print_line("-a-", a, mode, user, group, name)
        else:
            print_line("m--", m, mode, user, group, name)
            print_line("-a-", a, mode, user, group, name)
            print_line("--c", c, mode, user, group, name)
```

Generating timelines the quick and dirty way

Doing it this way is very easy, which is good, especially if you are coaching an inexperienced admin.

However, you will be messing up the atimes on every directory, and you will miss information about deleted files.

If you are not careful about where you store the output data, it may overwrite important deleted data blocks on the system.

Also, if the system is rootkitted, you will miss any hidden files.

Slightly slower and cleaner timelines

Alternatively, you can use The Sleuth Kit[3], TSK, to generate timelines.

TSK is an open source toolkit that, among other things, can generate timelines by reading the raw disk device (or a disk image).

TSK finds deleted inodes and directory entries.

[3] `http://www.sleuthkit.org/`

TSK timelines

```
fls -r -m / /dev/sda1 > rootfs.body
```

```
mactime -b rootfs.body > rootfs.timeline
```

TSK timelines

With TSK you bypass the kernel filesystem code and any rootkits, revealing any hidden files. You also see deleted directory entries.

However, you have to somehow either make the TSK binaries available on the system (compile them in place, transfer them to from another system or mount some filesystem (NFS, USB stick…)), or make an image of the disk and transfer it somewhere else.

Example

```
Aug 16 14:03:15 .a. r-xr-xr-x root     root     /usr/bin/w
Aug 16 14:03:28 .a. rwxr-xr-x root     root     /usr/bin/curl
Aug 16 14:03:36 .a. rwxr-xr-x root     root     /usr/bin/bzip2
Aug 16 14:04:41 .a. rwxr-xr-x root     root     /usr/bin/shred
Aug 16 14:06:26 .a. rw-r--r-- root     root     /usr/include/crypt.h
Aug 16 14:07:25 m.. rwxrwxr-x x_lenix x_lenix /var/tmp/...
Aug 16 14:08:01 m.c rw-r--r-- root     root     /var/tmp/.../openssh-5.2p1.tar.bz2
                                                        (deleted-realloc)
Aug 16 14:08:01 m.c rw-r--r-- root     root     /var/tmp/.../openssh-5.2p1
                                                        (deleted-realloc)
```

What does the timeline tell us?

ctimes often tells us when files were created
atimes can tell us when files were read and binaries executed
mtimes can be useful *because* they can be modified

mtime preservation

Many commands preserve mtimes (and atimes) when they copy files.

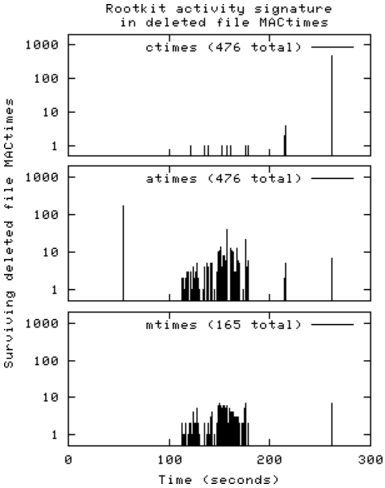
```
[nixon@host1]$ stat fie
  File: ‘fie’
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: fd01h/64769d Inode: 314704      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  500/   nixon)   Gid: (  500/   nixon)
Access: 2012-04-19 13:39:29.311321819 +0200
Modify: 2012-04-19 13:39:29.311321819 +0200
Change: 2012-04-19 13:39:29.311321819 +0200
 Birth: -

[nixon@host1]$ scp -p fie host2:

[nixon@host2]$ stat fie
  File: ‘fie’
  Size: 0           Blocks: 0          IO Block: 4096   regular empty file
Device: ca10h/51728d Inode: 2013283013  Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 7090/   nixon)   Gid: ( 7090/   nixon)
Access: 2012-04-19 13:39:29.000000000 +0200
Modify: 2012-04-19 13:39:29.000000000 +0200
Change: 2012-04-19 14:15:50.905321991 +0200
```

Traps and complicating factors

- You only see the last timestamp – surprisingly easy to forget!
- Prelinking
- `updatedb`
- `tmpwatch`
- `makewhatis`

Signs of modifications

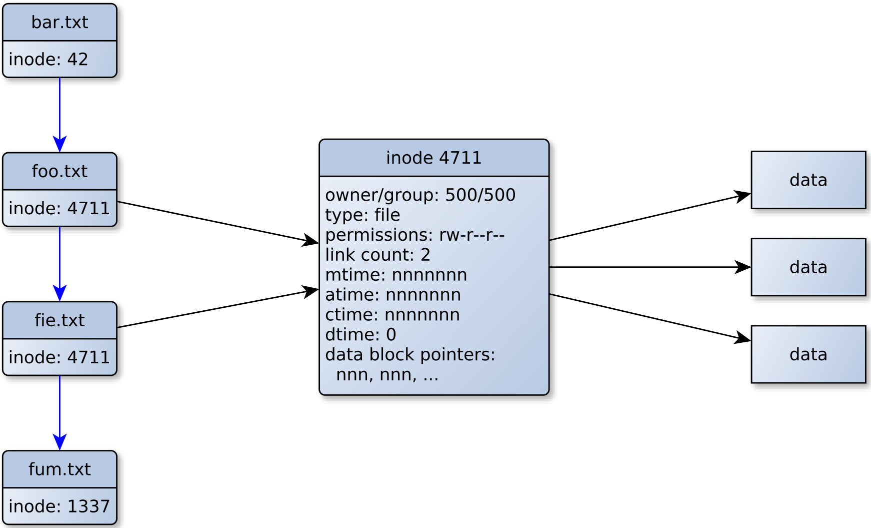
Has something interesting happened to this filesystem?




```
ls -i
12982  fileA
34919  fileB
12984  fileC
12985  fileD
```

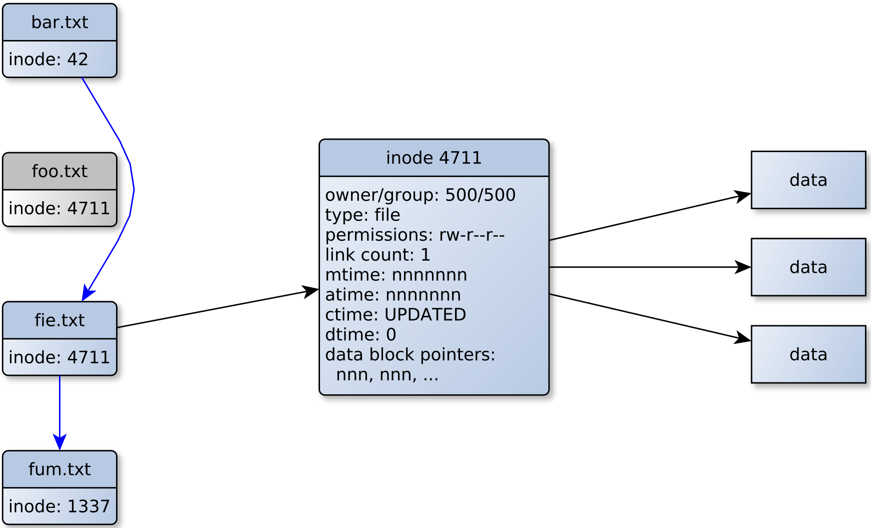
Looking at deleted data

When a file is deleted, it is of course not actually removed from the disk.



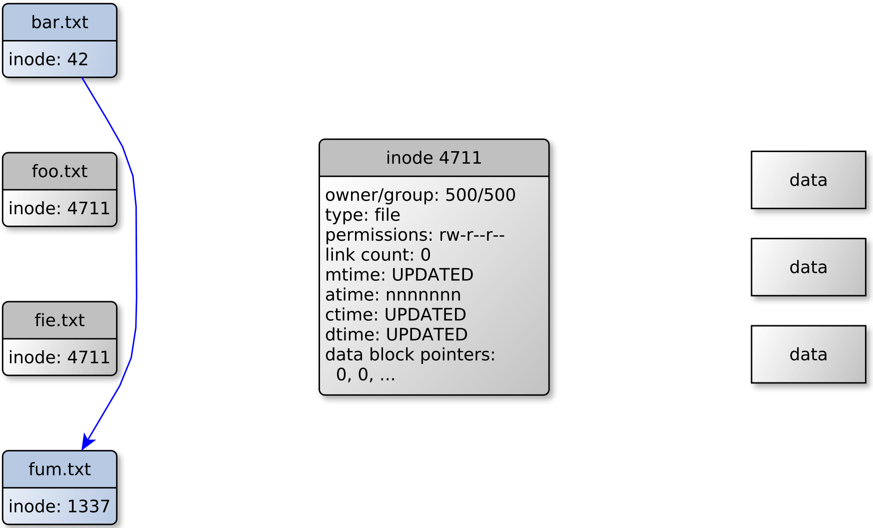
Looking at deleted data

When a file is deleted, it is of course not actually removed from the disk.



Looking at deleted data

When a file is deleted, it is of course not actually removed from the disk.



Looking at deleted data

TSK can display deleted directory entries and inodes.

Looking at deleted data

TSK can display deleted directory entries and inodes.

Retrieving the contents of deleted files is harder. If a file was deleted sufficiently recently that the inode contents remain in the file system journal, it can be recovered using `extundelete` or `ext3grep`.

Looking at deleted data

TSK can display deleted directory entries and inodes.

Retrieving the contents of deleted files is harder. If a file was deleted sufficiently recently that the inode contents remain in the file system journal, it can be recovered using `extundelete` or `ext3grep`.

Even if all metadata is lost and the file can't be undeleted, the contents can often be recovered using a tool like `photorec`.

Looking at deleted data

TSK can display deleted directory entries and inodes.

Retrieving the contents of deleted files is harder. If a file was deleted sufficiently recently that the inode contents remain in the file system journal, it can be recovered using `extundelete` or `ext3grep`.

Even if all metadata is lost and the file can't be undeleted, the contents can often be recovered using a tool like `photorec`.

My favourite quick and dirty method: Simply grep through the whole image!

```
strings sda.img | grep "sshd.*Accepted "
```

Looking at deleted data

Since disk space is allocated in data blocks of (typically) 4096 bytes there will be some unused space in the last data block if the file size is not a multiple of 4096. This unused space is called *slack space*.

Slack space is mainly interesting for two reasons; it can sometimes contain data from old deleted files, and it can be used by an intruder to hide data on the disk.

Similarily to deleted data, slack space data can be found by grepping through the disk image.

Working with disk images

Grabbing a disk image is easy enough. To get the whole disk:

```
dd if=/dev/sda of=sda.img bs=512
```

Just a specific partition:

```
dd if=/dev/sda1 of=sda1.img bs=512
```

Caution: if disk is mounted at the time, the resulting image will be inconsistent and probably not mountable. Still, TSK will be able to work with it.

Working with disk images

Listing and extracting partitions with TSK:

```
$ mmls -a sda.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
02:  00:00   0000002048   0000022527   0000020480   Linux (0x83)
06:  01:00   0000024576   0000126975   0000102400   Linux (0x83)
10:  02:00   0000129024   0000169983   0000040960   Linux Swap / Solaris x86 (0x82)
14:  03:00   0000172032   0000262143   0000090112   Linux (0x83)

$ mmcat sda.img 6 > sda2.img

$ ls -lh
total 51M
-rw-rw-r-- 1 nixon nixon  50M Apr 22 12:36 sda2.img
-rw-rw-r-- 1 nixon nixon 129M Apr 22 12:32 sda.img
```

Working with disk images

Images can be loopback-mounted for easy access:

```
mount -o loop,ro sda2-copy.img mnt
```

Sometimes this will fail; this can be because the image is corrupted, or simply because you have to replay the journal by briefly mounting the image read-write[4]:

```
mount -o loop sda2-copy.img mnt
umount mnt
mount -o loop,ro sda2-copy.img mnt
```

[4] or by using the `norecovery` mount option, but then the filesystem may be inconsistent

Working with disk images

To avoid time- and space-consuming copy operations, you can work with partitions in-place:

`fls -o 24576`[5] `-r -m / sda.img > rootfs.body`

`mount -o ro,loop,offset=12582912`[6] `sda-copy.img mnt`

[5] Offset in sectors, as reported by `mmls`

[6] Offset in bytes, i.e. 24576×512

Looking at other data

Of course, we must also look at other data sources on the running system. However, if the system is root compromised, it might be lying to us.

We might gain some confidence in the system by verifying system binaries by running e.g. `rpm -Va`[7].

If we find that e.g. the `ps` binary has been replaced, perhaps we can copy a fresh binary from another system, or simply use `pstree` or `top` instead.

[7] *Don't* do this before you have gathered timestamps, since it will zap all atimes!

Looking at processes

Once we think that we might be getting reliable data, look at the running processes. Remember that malicious processes can change their name to masquerade as, say, an extra `init` process.

Look for anomalies like duplicate system processes or strange inheritances (`ping` should not have a `bash` child process, for example).

Also look at pid numbers; system processes usually have pids in a narrow range. Something that looks like a system process but has a much higher pid might be suspicious.

Looking at open files and sockets

Use `netstat` and `lsof` to check open files and sockets. This can help identifying evil processes.

Looking at memory

If you find a malicious process, its memory may contain important information. You can use e.g. `gcore`[8] to generate a core dump for the process. Running `strings` on this can often reveal stuff like IP addresses and passwords.

It may also be interesting to dump the entire RAM of the system. Unfortunately, doing this can be less than trivial in modern kernels – see e.g. `https://code.google.com/p/lime-forensics/` for one method, and to actually analyze the dump, check out Volatility (`https://code.google.com/p/volatility/wiki/LinuxMemoryForensics`).

[8] part of the gdb package

Quick and dirty malware analysis

If you find a malicious binary, running `strings -a`[9] on it can yield interesting results.

You can also try to execute the binary under `strace` and `ltrace` to see in greater detail what it is doing. *This must be done very carefully*, preferably on an isolated host (like e.g. a VM without network access).

[9] Little known fact: `strings` will try to be smart when it's being run on an ELF binary and just look through certain ELF sectors. Use `-a` to read the whole file.

Quick and dirty malware analysis

`ltrace` and `strace` of a suspect `sshd` binary when logging in as `myuser:mypassword`:

```
:
:
3348 strcmp("mypassword", ".ssh/authorized_keys2  ") = 1
3348 memset(0x7fff24742210, '\000', 2048)        = 0x7fff24742210
3348 memset(0x7fff24742c10, '\000', 512)         = 0x7fff24742c10
3348 memset(0x7fff24742a10, '\000', 512)         = 0x7fff24742a10
3348 strcat("SR: '", "myuser")                   = "SR: 'myuser"
3348 strcat("SR: 'myuser' '", "mypassword")          = "SR: 'myuser' 'mypassword"
:
:


:
:
3318  <... read resumed> "\n\0\0\0\6mypassword", 11) = 11
3321  read(4,  <unfinished ...>
3318  open("/usr/share/kbd/keymaps/i386/azerty/c1", O_RDWR|O_CREAT|O_APPEND, 0666) =
3318  getuid()                          = 0
:
:
```

Obfuscated data

Often, trojans will obfuscate strings (e.g. filenames) in the binary and data in log files. This is almost, almost always done by xor:ing the data with a single byte.

So, if a file contains binary junk, try xor:ing it with different values until you find something interesting.

Obfuscated data

```
$ file azerty/c1
azerty/c1: data

$ xor.py azerty/c1

$ ls
0x01.out  0x21.out  0x41.out  0x61.out  0x81.out  0xa1.out  0xc1.out  0xe1.out
0x02.out  0x22.out  0x42.out  0x62.out  0x82.out  0xa2.out  0xc2.out  0xe2.out
:
:

$ grep SR: *.out

0xff.out: SR: 'myuser' 'mypassword'
```

Obfuscated data

```
#! /usr/bin/python

import sys, argparse

def xor(buf, n):
    f = open("0x%02x.out" % n, "w")
    for c in buf:
        f.write(chr(ord(c) ^ n))
    f.close()

parser = argparse.ArgumentParser(description="xor a file with one or several integer values, output to one or s
parser.add_argument("-n", help="Integer to xor with (default: loop over 1-255)")
parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
                    default=sys.stdin, help="Input file (default: stdin)")
args = parser.parse_args()
if args.n:
    if args.n.startswith("0x"):
        n = int(args.n, 16)
    else:
        n = int(args.n)
else:
    n = None

data = args.infile.read()

if n:
    xor(data, n)
else:
    for i in range(1,256):
        xor(data, i)
```

Looking at logs

In a root intrusion, local system logs may be wiped or sanitized. Of course, this shouldn't be a problem, since you are also logging remotely to a secure central log server, *right*?

However, in the unlikely event that you don't have remote logging, remember that e.g. ssh logins will leave traces in many different places, including (on a standard RHEL5-type system):

- `/var/log/secure` – ssh logs
- `/var/log/wtmp` – binary db of terminal sessions
- `/var/log/btmp` – binary db of failed logins
- `/var/log/lastlog` – binary (sparse file) db of latest logins per user
- `/var/log/audit/audit.log` – events from the audit subsystem

Even if the intruder has tried to remove his traces, he might have missed one of these places – check them all!

A brief note on rootkits

The main problem with the quick and dirty approach to forensics is that we are placing a lot of trust in the tools on the system. If the intruder has deployed a rootkit, we may be in trouble.

User level rootkits

User level rootkits basically work by replacing key system binaries. These can often be discovered by running e.g. `rpm -Va` (this of course presumes that `rpm` itself is trustworthy – you may want to use several different methods to verify binaries).

Kernel-based rootkits

Kernel-based rootkits instead subverts the running kernel into lying about the state of the system. Kernel rootkits can typically hide the existence of certain files and processes. These rootkits can be hard to detect, but tools like `chkrootkit` and `rkhunter` can find some common kinds of rootkits.

It is also noteworthy that TSK usually can detect files hidden by kernel rootkits, since it bypasses most of the kernel filesystem stack.

Putting it all together

Once you have all your data, remember to compare and cross-check your data sources; Do command line histories match atimes on binaries? Do filesystem timestamps match login/out times? Etc, etc.

This not only increases the confidence level of your data, it can also help you gain new insights.

For example, if filesystem timestamps indicate attacker activities taking place at times when there were no active logins, perhaps there is a backdoor on the system that you have missed.

Going back into service

After your forensic investigation, you hopefully have a pretty good idea of the intruder's actions. This allows you to clean your system from malware and to plug any hole the intruder might have exploited.

However, a *root compromised* system will almost always need to be installed. Still, your investigation will help you stop the same thing from happening in the future.

A quick and dirty conclusion

We have looked at some simple methods to collect forensic data. These methods are somewhat fragile and can be fooled by a clever attacker.

However, most attackers aren't very clever, and the quick and dirty approach surprisingly often can give a surprisingly detailed picture of the intruder's actions.

All sysadmins should know some basic quick and dirty forensics methods.

Hopefully, you do so now. Let's try it!