Stealth V.2.04.00

Frank B. Brokken
Center for Information Technology, University of Groningen

2005-2012

Contents

Chapter 1

Introduction

1.1 What's new in Stealth V.2.04.00

- The file specified at the option -s (–skip-files) may contain empty lines and lines having a # as their 1st non-blank character. These lines are ignored. Furthermore, initial and trailing blanks on its lines are ignored.
- NOTE: requires bobcat 2.17.00 or later

1.2 Stealth

Welcome to **stealth**. The program **stealth** implements a file integrity scanner. The acronym **stealth** can be expanded to

SSH-based Trust Enforcement Acquired through a Locally Trusted Host.

This expansion contains the following key terms:

- `SSH-based`: The file integrity scan is (usually) performed over an ssh-connection. Usually the computer being scanned (called the *client*) and the computer initiating the scan (called the `controller`) are different computers.
- The client should accept incoming ssh-connections from the initiating computer. The controller doesn't have to (and shouldn't, probably).
- `Trust Enforcement`: following the scan, 'trust' is enforced in the client, due to the integrity of its files.

- `Locally Trusted Host`: the client apparently trusts the controller to use an ssh-connection to perform commands on it. The client therefore *locally trusts* the controller. Hence, *locally trusted host*.

stealth is based on an idea by *Hans Gankema* and *Kees Visser*, both at the Center for Information Technology of the University of Groningen.

stealth's main task is to perform file integrity tests. However, the testing will leave no sediments on the tested computer. Therefore, **stealth** has *stealthy* characteristics. I consider this an important security improving feature of **stealth**.

The controller itself only needs two kinds of outgoing services: **ssh**(1) to reach its clients, and some mail transport agent (e.g., **sendmail**(1)) to forward its outgoing mail to some mail-hub.

Here is what happens when **stealth** is run:

- First, a *policy* file is read. This determines actions to be performed, and values of several variables used by **stealth**.
- If the command-line option `-keep-alive` or `-repeat <seconds>` is given, **stealth** will run as a backgrond process, displaying the process ID of the background process. With `-repeat <seconds>` the scan will be rerun every `<seconds>` seconds. The number of seconds until the next rerun will be at least 60. However, using the `-rerun` option a background **stealth** process may always be goated into its next scan. When `-keep-alive` is specified the scan will be performed just once, whereafter **stealth** will wait until it is reactivated by another run of **stealth**, called using the `-rerun <pid>` command-line option.
- Then, the controller opens a command shell on the client using **ssh**(1), and a command shell on the controller itself using **sh**(1).
- Next, commands defined in the policy file are executed in their order of appearance. Examples are given below. Normally, return values of the programs are tested. Non-zero return values will terminate **stealth** prematurely.
- In most cases, integrity tests can be controlled by the **find**(1) program, calling programs like **ls**(1), **sha1sum**(1) or its own `-printf` method to produce file-integrity related statistics. Most of these programs write file names at the end of generated lines. This characteristic is used by an internal routine of **stealth** to detect changes in the generated output, which could indicate some harmful intent, like an installed *root-kit*.
- When changes are detected, they are logged on a *report file*, to which information is always appended. **stealth** never reduces or rewrites the report file. When information is added to the report file the newly written information is emailed to a configurable email address for further (human) processing. Usually this will be the systems manager of the tested client. **stealth** follows the 'dark cockpit' approach in that no mail is sent when no changes were detected.

Alternatively, the command-line options `-rerun` and `-terminate` may be provided to communicate with a **stealth** process started earlier using either the `-keep-alive` or `-repeat` option. In this case,

- When started using the `-terminate <pid>` command-line option, the stealth process running at process-ID `<pid>` is terminated. Note that no check is performed as to whether the process associated with `<pid>` is truly a **stealth** process. It is the responsibility of the user to make sure that the process-ID of the intended process is specified.
- When started using the `-rerun <pid>` command-line option, the stealth process running at process-ID `<pid>` will perform another scan. Again, no check is performed as to whether the process associated with `<pid>` is truly a **stealth** process. It is the responsibility of the user to make sure that the process-ID of the intended process is specified.

The options `-suppress` and `-rerun` (see section 5.7) were implemented to allow safe rotations of **stealth**'s report file.

1.2.1 The integrity of the stealth distribution

The integrity of the archive `stealth-2.04.00.tar.gz` can be verified as follows:

- At the location where you found this archive, you should also find a file named `stealth-2.04.00.dsc`. This file contains a PGP signed **sha1sum**(1) signature of the `tar.gz` archive. The PGP sigature was provided by me using **gpg**(1) (**pgp**(1)).
- Compute the SHA1 checksum of the `stealth-2.04.00.tar.gz` archive. Its value should match the SHA1 checksum that is mentioned in the `stealth-2.04.00.dsc` file. If not, the `stealth-2.04.00.tar.gz` archive has been compromised, and should *not* be used.
- In order to verify the validity of the electronic signature, do as follows:
  - Obtain my public key from a public PGP keyserver, e.g.

    ```
    http://pgp.surfnet.nl:11371/
    ```

  - Make sure you have the right key. Its fingerprint is

    ```
    8E36 9FC4 1DAA FCDF 1A0D  B19F DAC4 BE50 38C6 6170
    ```

    and it has been electronically signed by, e.g., the University of Groningen's PGP-certificate authority. If in doubt, contact me to verify you have the right key.

– Once you're sufficiently satisfied that you indeed have obtained my public PGP key, verify the validity of the signature used for signing `stealth-2.04.00.dsc`. With **gpg**(1) this can be done by the command

```
gpg --verify stealth-2.04.00.dsc
```

This should produce output comparable to:

```
gpg: Signature made Mon Aug  1 10:57:41 2005 CEST using DSA key ID 38C66170
gpg: Good signature from "Frank B. Brokken <f.b.brokken@rug.nl>"
gpg:                 aka "Frank B. Brokken <f.b.brokken@rc.rug.nl>"
```

Chapter 2

Installation

This chapter describes **stealth**'s compilation and installation.

2.1 Compiling and installing Stealth

After downloading the **stealth** archive, it should be unpacked. The name of the archive is of the form `stealth-2.04.00.tar.gz`, where `2.04.00` is a version number. Below, `2.04.00` should be altered into the version of the archive that is actually used.

- First, determine a directory under which the archive's file should be stored. E.g., if the files in the archive should be stored under `/tmp`, and the archive itself is stored in `/tmp` as well, do:

  ```
  cd /tmp
  tar xzf stealth-2.04.00.tar.gz
  ```

  This creates a subdirectory `stealth` in which the sources are found

- Next, `chdir` to that directory:

  ```
  chdir stealth
  ```

- Check the contents of the file `make/parameters`. It should need no modifications. Among other entries, it contains the entry `GCC=g++`, indicating the compiler to use. The compiler should be the `GNU g++` compiler version 4.0.2 or above. Also note `-lbobcat` in the entry

```
LOPTS="-lbobcat -lstealth -L. -s"
```

When compiling **stealth**, the `bobcat`[1] library must be available. If you haven't installed **bobcat** yet, download it from `http://sourceforge.net/projects/bobcat/`, and follow its installation instructions. Make sure to install both the run-time (**bobcat_...**) and the development (**bobcat-dev_...**) versions.

- Execute the command

```
make/program
```

This command (note that it is **not** *make program*!) will create the program **./tmp/bin/stealth**, which may then be installed in, e.g., `/usr/bin`.

[1] http://bobcat.sourceforge.net/

Chapter 3

The 'policy' file

stealth reads a policy file defining the actions that must be performed. Each policy file is uniquely associated with a host to be tested. There may be multiple policy files for a host, though. In that case, each policy file will define a certain set of checks to be performed.

Below, the term *controller* is used for the computer where **stealth** is started, while the term *client* is used for the computer that is scanned by **stealth**. The controller and the client could be the same computer, but normally they are different.

The policy file consists of three sets of data: *define directives* (starting with the keyword **DEFINE**), *use directives* (starting with the keyword **USE**) and *commands*.

Directives are written in capitals, and should appear exactly as written below: letter casing is preserved.

Blank lines and information beyond hash-marks (#) are ignored, while lines following lines terminating in backslashes (\) will be concatenated (*en passant* removing the backslashes). Initial white space on lines of the policy file is ignored.

3.1 DEFINE directives

`DEFINE` directives can be used to define symbols for longer strings. A `DEFINE` directive is constructed as follows:

```
DEFINE name     that what is defined by 'name'
```

Here,

- the `name` following `DEFINE` is the symbol that may be used in `USE` directives (see below) and `commands` (see below).
- `DEFINE` symbols can be used in other `DEFINE` symbols. However, it is the responsibility of the author of the policy file to make sure that (indirect) circular definitions are avoided. E.g., after:

```
DEFINE A    ${B}
DEFINE B    ${A}
DEFINE C    ${C}

USE MAILARGS ${A} ${B} ${C}
```

`MAILARGS` will be expanded to

```
 ${A} ${A} ${C}
```

- The text following `DEFINE name` is then inserted literally into the `USE` directive or `command`.

  Example:

```
DEFINE  SSH         /usr/bin/ssh frankbash@localhost -q
DEFINE  EXECSHA1     -xdev -perm +111  -type f -exec /usr/bin/sha1sum {} \;
```

  The symbols defined by `DEFINE` directives may consist of letters, digits and the underscore character (_). In the definition of the symbol any character can be used. The definition is, however, trimmed of initial or trailing blanks.

  To insert a definition into a `USE` directive or `command` use the

```
${name}
```

  form. E.g., `${EXECSHA1}`. Concrete examples will be given below.

3.2 USE directives

`USE` directives provide **stealth** with arguments which may be conditional to a certain installation. The following **USE** directives may be specified:

- **USE BASE** `basedirectory`

  BASE defines the directory from where **stealth** operates. All relative path specifications are interpreted relative to **BASE**. *By default* this is the directory where **stealth** was started.

  BASE and all other directories that are used below `BASE` are created by **stealth** if not yet existing.

  Example:

  ```
  USE BASE /root/client
  ```

  All information generated by **stealth** is written in or below the directory `/root/client`.

- **USE DD** `<dd>`

  The **DD** specification uses `/bin/dd` as default, and defines the location of the **dd**(1) program, both on the server and on the client. The **bin**(1) program is used to copy files between the client and the controller without opening separate ssh-connections. The program specified here is only used by stealth for the `PUT` and `GET` commands, described below.

  Example showing the default:

  ```
  USE DD /bin/dd
  ```

- **USE DIFF** `path-to-diff`

  The **DIFF** specification uses `/usr/bin/diff` as default, and defines the location of the **diff**(1) program. The **diff**(1) program is used to compare a formerly created logfile of an integrity check to a newly created logfile.

  Example showing the default:

  ```
  USE DIFF /usr/bin/diff
  ```

- **USE EMAIL** `address`

  The **EMAIL** specification defines the email-address to e-mail the client's integrity scan report to. Mail is only sent when information has changed.

  Example showing the default:

  ```
  USE EMAIL root
  ```

- **USE MAILER** `mailer`

  The **MAILER** specification defines the program that is used to send the mail to the **EMAIL**-address. By default this is /**usr**/**bin**/**mail**(1). The **MAILER** program is called as follows:

```
MAILER  MAILARGS EMAIL
```

(`MAILARGS`: see below). The information to be mailed is read from `MAILER`'s standard input stream.

Example showing the default:

```
USE MAILER /usr/bin/mail
```

- **USE MAILARGS** `arguments` The **MAILARGS** specification defines the arguments to be to be passed to the `MAILER` program. By default this is

```
USE MAILARGS -s "STEALTH scan report"
```

Note that blanks may be used in the subject specification: use double or single quotes to define elements containing blanks. Use \" to use a double quote in a string that is itself delimited by double quotes, use \' to use a single quote in a string that is itself delimited by single quotes.

Subtle note: in a construction like

```
USE MAILARGS " 't was brillig " and 't went well
```

the following arguments are passed to `MAILER`:

  - `" 't was brillig "`
  - `and`
  - `'t`
  - `went`
  - `well`

So, when single- and double-quoted strings overlap, the first string is taken as a string, and the information beyond the first string is thereupon interpreted.

- **USE REPORT** `reportfile`

REPORT defines the name of the reportfile. Information is always appended to this file. For each run of **stealth** a *time marker line* is written to the report file. Such a marker line looks like this:

```
STEALTH (1.11) started at Mon Jun 16 12:57:26 2003
```

Only when (in addition to the marker line) additional information was appended to the report file, the added contents of the report file are mailed to the mail address specified in the **USE EMAIL** specification.

Example showing the default:

```
USE REPORT report
```

- **USE ROTATE** `interval: number interval-name`[,] [`count: number`][,] [`, zip: number [zip-program-path]`]

  ROTATE defines the parameters **stealth** will use to rotate its report file. This **USE** specification supports three elements, the first of which is obligatory when **USE ROTATE** is specified. Note that the square brackets are not used in the specification, and indicate optional elements, which may or may not be specified:

  - `interval: number interval-name` defines the time interval until the report file is rotated. Rotation can be specified using an integral, positive number, followed by `hour` or `hours` for hours, `day` or `days` for days, `week` or `weeks` for weeks and `month` or `months` for months. By default no rotation takes place. If rotation is requested, the current report file is moved to the file `reportfile.1`, while existing numbered reportfiles are moved to higher ordered numbers first (so, before moving the current reportfile to `reportfile.1`, an existing `reportfile.1` is first moved to `reportfile.2`, etc.).
  - `count: number` defines the number of report files **stealth** will eventually use. By default, if **USE ROTATE** is specified, there is no practical limit to the number of report files **stealth** will create (in these cases, another program supposedly controls the number of report files that will eventually be used). External programs may freely manipulate all report files that have been rotated by **stealth**, but they should not modify the active report file (specified using the **USE REPORT** specification).
  - `zip: number zip-program-path` defines the first of the rotated files that should be compressed, using `zip-program-path` to compress the report files. By default, no compression is used, but if `zip:` is specified, the default program that will be used to compress a report file is **/bin/gzip**. If another program is used, it should expect a filename as its first argument, which will then be zipped to a new file receiving the extension **.gz**, appended to the name that was provided as its first argument. The original file is removed during the zipping-process.

    Example showing a report interval of one week, using a total of 12 report files, compressing all report files but the actual report file and its predecessor (having filename `reportfile.1`):

    ```
    USE ROTATE interval: 1 week, count: 12, zip: 2 /bin/gzip
    ```

- **USE SH** `sh-specification`

  The **SH** specification uses `/bin/sh` as default, and defines the command shell used by the controller to execute local commands.

  Example showing the default:

  ```
  USE SH /bin/sh
  ```

- **USE SSH** `ssh-specification`

  The SSH specification has no default, and *must* be specified. Assuming the client *trusts* the controller (which is, after all, what this program is all about; so this should not be a very strong assumption), preferably the public ssh-identity key of the controller should be placed in the client's root `.ssh/authorized_keys` file, granting the controller root access to the client. Root access is normally needed to gain access to all directories and files of the client's file system.

  In practice, connecting to a account using the **sh**(1) shell is preferred. When another shell is already used by that account, one should make sure that that shell doesn't setup its own redirections for standard input and standard output. One way to accomplish that is for force the execution of `/bin/sh` in the **USE SSH** specification.

  An example of an `SSH` specification to scan a localhost is:

  ```
  USE SSH root@localhost -T -q                # root's shell is /bin/sh
  ```

  The same, now explicitly using `/bin/bash`:

  ```
  USE SSH root@localhost -T -q exec /bin/bash # root uses another shell
  ```

  Alternatively, `-profile` can be specified to prevent any profile-initialization:

  ```
  USE SSH root@localhost -T -q exec /bin/bash --noprofile
  ```

Note, however, that using `stealth` to inspect `localhost` is *not* recommended, as it counters one of the main reasons for `stealth`'s existence.

As yet another alternative, applicable only to `localhost`, `ssh` could be avoided altogether. In that case `/bin/bash` or a comparable shell may be specified with `USE SSH`. For example:

```
# For stealth inspecting localhost:
    USE SSH /bin/bash --noprofile
```

3.3 Commands

Following the **USE** specifications, *commands* can be specified. The commands are executed in their order of appearance in the policy file. Processing continues until the last command has been processed or until a tested command (see below) returns a non-zero return value.

3.3.1 LABEL commands

The following **LABEL** commands are available:

- **LABEL** `text`

  This defines a text-label which is written to the **REPORT** file, just before the output generated by the next **CHECK**-command. If the next **CHECK**-command generates no output, the label is not written to the **REPORT**-file. Once a **LABEL** has been defined, it is used until it is redefined by the next **LABEL** command. Use an empty **LABEL** command to suppress the printing of labels.

  The text may contain `\n` characters (two characters) which are transformed to a newline character.

- **LABEL**

  As noted, this clears a previously defined `LABEL` command.

Examples:

```
LABEL Inspecting files in /etc\nIncluding subdirectories
LABEL
```

The second **LABEL** command clears the first label.

3.3.2 LOCAL commands

LOCAL commands can be used to specify commands that are executed on the controller itself. The following **LOCAL** commands are available:

- **LOCAL** `command`

  Execute `command` on the controller, using the **SH** command shell. The command must succeed (i.e., must return a zero exit value). Example:

  ```
  LOCAL mkdir /tmp/client
  ```

This command will create the directory `/tmp/client` on the controller.

- **LOCAL NOTEST** `command`

  Execute `command` on the controller, using the **SH** command shell. The command may or may not succeed. Example:

```
LOCAL NOTEST mkdir /tmp/subdir
```

  This command will create `/tmp/subdir` on the controller. The command will fail if the directory cannot be created, but this will not terminate **stealth**.

- **LOCAL CHECK** [**LOG** =] `logfile command`

  Execute `command` on the controller, using the **SH** command shell. The phrase '**LOG** =' is optional. If the command does not succeed a *warning* message is written to the report file. The warning message informs the reader that 'remaining results might be forged:

```
*** BE CAREFUL *** REMAINING RESULTS MAY BE FORGED
```

  This situation may occur, e.g., if an essential program (like `sha1sum`) was transferred to the controller, and it was apparently modified since the previous check. Processing continues, but remaining checks performed at the client computer should be interpreted with *extreme* caution.

  The output of this command is compared to the output of this command generated during the previous run of **stealth**. Any differences are written to **REPORT**.

  If differences were found, the existing `logfile` name is renamed to `logfile.YYYYMMDD-HHMMSS`, with `YYYYMMDD-HHMMSS` the datetime-stamp at the time **stealth** was run.

  Over time, many `logfile.YYMMDD-HHMMSS` files could be accumulated. It is up to the controller's systems manager to decide what to do with old datetime-stamped logfiles. For instance, the following script will remove all **stealth** reports below the current directory that are older than 30 days:

```
#/bin/sh
FILES=‘find ./ -path ’*[0-9]’ -mtime +30 -type f‘

if [ "$FILES" != "" ] ; then
    rm -f $FILES
fi
```

  The `logfile` specifications may use relative and absolute paths. When relative paths are used, these paths are relative to **BASE**. When the

directories implied by the `logfile` specifications do not yet exist, they are created first.

Example:

```
LOCAL CHECK LOG = local/sha1sum sha1sum /tmp/sha1sum
```

This command will check the SHA1 sum of the `/tmp/sha1sum` program. The resulting output is saved at **BASE**`/local/sha1sum`. The program must succeed (i.e., `sha1sum` must return a zero exit-value).

- **LOCAL NOTEST CHECK** [**LOG** =] `logfile command`

  Execute `command` on the controller, using the **SH** command shell. The phrase '**LOG** =' is optional. The command may or may not succeed. Otherwise, the program acts identically as the **LOCAL CHECK ...** command, discussed previously.

  Example:

```
LOCAL NOTEST CHECK LOG=local/sha1sum sha1sum /tmp/sha1sum
```

  This command will check the SHA1 sum of the `/tmp/sha1sum` program. The resulting output is saved at **BASE**`/local/sha1sum`. The program may or may not succeed (i.e., `sha1sum` may or may not return a zero exit-value).

3.3.3 REMOTE commands

Plain commands can be executed on the client computer by merely specifying them. Of course, this means that programs called `LABEL`, `LOCAL USE` or `DEFINE`, cannot be executed, since these names are interpreted otherwise by **stealth**. It's unlikely that this will cause problems. Remote commands must succeed (i.e., their return codes must be 0).

Remote commands are commands executed on the client using the **SSH** shell. These commands are executed using the standard `PATH` set for the **SSH** shell. However, it is advised to specify the full pathname to the programs to be executed, to prevent "trojan approaches" where a trojan horse is installed in an 'earlier' directory of the `PATH`-specification than the intended program.

Two special remote commands are `GET` and `PUT`, which can be used to copy files between the client and the controller. Internally, `GET` and `PUT` use the `DD` use-specification. If a non-default specification is used, one should ensure that the alternate program accepts **dd**(1)'s `if=, of=, bs=` and `count=` options. With `GET` the options `bs=, count=` and `of=` are used, with `PUT` the options `bs=, count=` and `if=` are used. Normally there should be no need to alter the default `DD` specification.

The GET command may be used as follows:

- **GET** <client-path> <local-path>
  Copy the file indicated by client-path at the client to local-path at the controller. client-path must be the full path of an existing file on the client, local-path may either be a local directory, in which case the client's file name is used, or another file name may be specified, in which case the client's file is copied to the specified local filename. If the local file already exists, it is overwritten by the copy-procedure.

  Example:
  GET /usr/bin/sha1sum /tmp
  The program /usr/bin/sha1sum, available at the client, is copied to the controller's /tmp directory. If the copying fails for some reason, any subsequent commands are skipped, and **stealth** terminates.

- **GET NOTEST** <client-path> <local-path>
  Copy the file indicated by client-path at the client to local-path at the controller. client-path must be the full path of an existing file on the client, local-path may either be a local directory, in which case the client's file name is used, or another file name may be specified, in which case the client's file is copied to the specified local filename. If the local file already exists, it is overwritten by the copy-procedure.

  Example:
  GET NOTEST /usr/bin/sha1sum /tmp
  The program /usr/bin/sha1sum, available at the client, is copied to the controller's /tmp directory. Remaining commands in the policy file are executed, even if the copying process wasn't successful.

The PUT command may be used as follows:

- **PUT** <local-path> <remote-path>
  Copy the file indicated by local-path at the controller to remote-path at the client. The argument local-path must be the full path of an existing file on the controller. The argument remote-path must be the full path to a file on the client. If the remote file already exists, it is overwritten by PUT.

  Example:
  PUT /tmp/sha1sum /usr/bin/sha1sum
  The program /tmp/sha1sum, available at the controller, is copied to the client as usr/bin/sha1sum. If the copying fails for some reason, any subsequent commands are skipped, and **stealth** terminates.

- **PUT NOTEST** <local-path> <remote-path>
  Copy the file indicated by local-path at the controller to remote-path at the client. The argument local-path must be the full path of an existing file on the controller. The argument remote-path must be the full path to a file on the client. If the remote file already exists, it is overwritten by PUT.

Example:
`PUT NOTEST /tmp/sha1sum /usr/bin/sha1sum`
Copy the file indicated by `local-path` at the controller to `remote-path` at the client. The argument `local-path` must be the full path of an existing file on the controller. The argument `remote-path` must be the full path to a file on the client. If the remote file already exists, it is overwritten by `PUT`. Remaining commands in the policy file are executed, even if the copying process wasn't successful.

Other commands to be executed on the client can be specified as follows:

- `command`

  Execute '`command`' on the client, using the **SSH** command shell. The command must succeed (i.e., must return a zero exit value). However, any output generated by the command is ignored. Example:

  ```
  /usr/bin/find /tmp -type f -exec /bin/rm {} \;
  ```

  This command will remove all ordinary files at and below the client's `/tmp` directory.

- **NOTEST** `command`

  Execute `command` on the client, using the **SSH** command shell. The command may or may not succeed.

  Example:

  ```
  NOTEST /usr/bin/find /tmp -type f -exec /bin/rm {} \;
  ```

  Same as the previous command, but this time the exit value of `/usr/bin/find` is not interpreted.

- **CHECK** [**LOG** =] `logfile command`

  Execute `command` on the client, using the **SSH** command shell. The phrase '**LOG** =' is optional. The command must succeed. The output of this command is compared to the output of this command generated during the previous run of **stealth**. Any differences are written to **REPORT**. If differences were found, the existing `logfile` name is renamed to `logfile.YYYYMMDD-HHMMSS`, with `YYYYMMDD-HHMMSS` the datetime-stamp at the time **stealth** was run.

  Note that the command is executed on the client, but the logfile is kept on the controller. This command represents the core of the method implemented by **stealth**: there will be no residues of the actions performed by **stealth** on the client computers.

  Several examples (note the use of the backslash as line continuation characters):

```
CHECK LOG = remote/ls.root  /usr/bin/find / \
        -xdev -perm +6111 -type f -exec /bin/ls -l {} \;
```

All suid/gid/executable files on the same device as the root-directory (/) on the client computer are listed with their permissions, owner and size information. The resulting listing is written on the file **BASE**`/remote/ls.root`.

This long command could be formulated shorter using a `DEFINE`:

```
DEFINE LSFIND   -xdev -perm +6111 -type f -exec /bin/ls -l {} \;
CHECK remote/ls.root  /usr/bin/find / ${LSFIND}
```

Another example:

```
DEFINE SHA1SUM -xdev -perm +6111 -type f -exec /usr/bin/sha1sum {} \;
CHECK remote/sha1.root /usr/bin/find / ${SHA1SUM}
```

The SHA1 checksums of all suid/gid/executable files on the same device as the root-directory (/) on the client computer are determined. The resulting listing is written on the file **BASE**`/remote/sha1.root`.

- **NOTEST CHECK** [**LOG** =] `logfile command`

  Execute `command` on the client, using the **SSH** command shell. The phrase '**LOG** =' is optional. The command may or may not succeed. Otherwise, the program acts identically as the **CHECK ...** command, discussed previously. Example (using the same `${SHA1SUM}`)definition:

```
NOTEST CHECK LOG = remote/sha1.root /usr/bin/find / ${SHA1SUM}
```

  The SHA1 checksums of all suid/gid/executable files on the same device as the root-directory (/) on the client computer are determined. The resulting listing is written on the file **BASE**`/remote/sha1.root`. **stealth** will not terminate if the `/usr/bin/find` program returns a non-zero exit value.

3.3.4 Preventing Controller Denial of Service (–max-size)

Either by malicious intent or by accendent (as happened to me) the controller may be a victim of a Denial of Service (DOS) attack. This DOS attack may happen when the client (apparently) sends a never ending stream of bytes in response to a `GET` or `REMOTE` command. One of my controllers once fell victim to

this attack when a client's power went down and the controller kept on trying to read bytes from that client filling up the controllers disk....

This problem was of course caused by a programming error: while reading information from a client **stealth** failed to check whether the reading had actually succeeded. This bug has now been fixed, but an intentional DOS attack could still be staged along this line by a hacker who manages to replace, e.g., the **find**(1) command by a manipulated version which would continue to write information to its standard output stream. Without further precaution **stealth** would receive a never ending stream of bytes as its 'report' thus causing its disk to fill.

To prevent this from happening **stealth** now offers the `-max-size` command line option allowing the specification of the maximum size of a stream of bytes received by **stealth** (e.g., a report or downloaded file). The maximum is used for each individual download and can be specified in bytes (using no suffix or the `B` suffix), kilo-bytes (using `K`), mega-bytes (using `M`) or giga-bytes (using `G`). The default is set at 10M, equivalent to the command line specification of `-max-size 10M`.

If a file or report received from the client exceeds its maximum allowed size then **stealth** terminates after writing the following message to the report file (which is sent to the configured mail address):

```
STEALTH - CAN'T CONTINUE: '<name of offending file>' EXCEEDS MAX.
                                                DOWNLOAD SIZE (<size shown>)
STEALTH - THIS COULD SIGNAL A SERIOUS PROBLEM WITH THE CLIENT
STEALTH - ONE OR MORE LOG FILES MAY BE INVALID AS A RESULT
STEALTH - *** INVESTIGATE ***
```

Since a `-max-size` specification may cause **stealth** to terminate while receiving the output of a (remotely run) command, an empty or partial log file will be the result. Of course this partial result is spurious as it is a direct result of **stealth** terminating due to a size violation.

After investigating (and removing) the reasons for the size violation a new **stealth** run using the previous log file as the latest baseline should show only expected changes.

For example, assume the following situation represents a (valid) state of logfiles:

```
etc                     stealth
setuid                  stealth.20080316-105756
```

Now **stealth** is run with `-max-size 20`, prematurely terminating **stealth**. This results in the following set of logfiles:

```
etc                     stealth
setuid                  stealth.20080316-105756
                        stealth.20080316-110215
```

The file `stealth` now contains incomplete data with the (latest) file `stealth.20080316-110215` containing its previous contents.

Now the reasons for the size-violation should be investigated and removed. It is suggested to move the file last saved (`stealth.20080316-110215`) to the file `stealth`, as it represents the state before the size violation was encountered. Following this **stealth** should operate normally again.

Chapter 4

Granting access

Access is granted via the `ssh` protocol.

The client must allow the controller to connect using `ssh`. Since normally no username and password can be given, the client must allow the controller to connect without specifying a password.

This is realized using *public key* technology, assuming `open-SSH` is available on both computers, with the client running an `sshd` daemon.

4.0.5 The controller's user: creating an ssh-key

The user on the controller who will call **stealth** to scan the client, now generates an `ssh-keypair`:

```
ssh-keygen -t rsh
```

This will generate a public/private ssh key pair in `.ssh` in the user's home directory. The program asks for a *passphrase* which should, for the purpose of **stealth** be **empty**: just pressing `Enter` as a response to the question

```
Enter passphrase (empty for no passphrase):
```

will do the trick (a confirmation is requested: press `Enter` again). The program returns a key fingerprint, e.g.,

```
03:96:49:63:8a:64:33:45:79:ab:ca:de:c8:c8:4f:e9 user@controller
```

which may be saved and used for future reference.

In the directory user's `.ssh` directory the files `id_rsa` and `id_rsa.pub` are now created.

This completes the actions on the controller.

4.0.6 The client's account: accepting ssh from the controller's user

Next, the account on the client where the `ssh` command connects to (using a specification in the policy file like

```
USE SSH /usr/bin/ssh -q account@client
```

must now grant access to the controller's user. In order to do so, the file `id_rsa.pub` of the user at the controller is added to the file `authorized_keys` in the `.ssh` directory of the account on the client:

```
# transfer user@controller's file id_rsa.pub to the client's /tmp
# directory. Then do:

cat /tmp/id_rsa.pub >> /home/account/.ssh/authorized_keys
```

Now user@controller may login at acount@client without specifying a password.

4.0.7 Logging into the account@client account

When user@controller now issues the command

```
ssh account@controller
```

`Ssh` responds as follows:

```
The authenticity of host 'controller (xxx.yyy.aaa.bbb)' can't be
established.
```

```
RSA key fingerprint is c4:52:d6:a3:d4:65:0d:5e:2e:66:d8:ab:de:ad:12:be.
Are you sure you want to continue connecting (yes/no)?
```

Answering `yes` results in the message:

```
Warning: Permanently added 'controller,xxx.yyy.aaa.bbb' (RSA) to the
list of known hosts.
```

The next time a login is attempted, the authenticity question isn't asked anyore. However, the proper value of the host's RSA key fingerprint (i.e., the key fingerprint of the *client* computer) should *always* be verified to prevent *man in the middle* attacks. The proper value may be obtained at the client computer by issuing the command

```
ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
```

This should result in the same value as the fingerprint shown when the first `ssh` connection was made. E.g.,

```
1024 c4:52:d6:a3:d4:65:0d:5e:2e:66:d8:ab:de:ad:12:be ssh_host_rsa_key.pub
```

4.0.8 Using the proper shell

On order to minimize the amount of clutter and possible complications when only a simple command-shell is required for executing commands, it is suggested to use a `bash` or `sh` shell when logging into the `account@client`'s account.

When another shell is already used for `account@client`, then an extra account (optionally using the same `UID` as the original account, but using **sh**(1) as the shell), could be used.

In the **passwd**(5) file this could be realized for *root* as *rootsh* as follows:

```
rootsh:x:0:0:root:/root:/bin/sh
```

If shadow passwording is used, an appropriate entry in the `/etc/shadow` file is required as well.

Chapter 5

Running 'stealth'

Now that **stealth** has been compiled, the construction of a policy file has been covered, and a service-account on the client has been defined, what must be done to run **stealth** in practice?

Here's what remains to be done:

- Install **stealth** at a proper location
- Construct one or more policy files
- Learn to interpret **stealth**'s output.
- Optionally, automate the removal of old log-files.
- Determine a schedule for running stealth automatically, e.g. using **cron**(1)

In this chapter, these topics will be discussed.

5.1 Installing 'stealth'

As **stealth** is mainly a system administrator's tool, it could be installed in `/usr/local/bin`. In that case, do (as *root*) from the directory where **stealth** was compiled/unpacked:

```
install stealth /usr/local/bin
```

options given to **install**(1) may restrict further use of **stealth**.

5.2 Construct one or more policy files

Here we assume that **stealth** is run by *root*, and that root wants to store information about the host `client` under the subdirectory `/root/stealth/client`.

Stealth reports should be sent to the user `admin@elsewhere`, who is only interested in a short notice of changes, as the full report can always be read elsewhere. So, a support-script is developed to further filter the report generated by **stealth**.

As the `sha1sum` program on the client may be hacked, it is a good idea to transfer the client's `sha1sum` program to the controller first, in order to check that program locally, before trusting it to compute the sha1sums of the client's files. The same holds true for any libraries and support programs (like `find`) that are used intensively during integrity scans

Sha1sum checks should be performed on all setuid and setgid files on the `client`, and in order to be able reach all files on `client`, `root@controller` is allowed to login to the `root@client` account using a password-less `ssh` connection.

Furthermore, sha1sum checks should be performed on all configuration files, living under `/etc` and on the file `/usr/bin/find` which is used intensively to perform the checks.

The required `policy` file is constructed as follows, per section:

5.2.1 the DEFINE directives

```
DEFINE  SSHCMD  /usr/bin/ssh root@client -T -q exec /bin/bash --noprofile
DEFINE  EXECSHA1 -xdev -perm +u+s,g+s \( -user root -or -group root \) \
                -type f -exec /usr/bin/sha1sum {} \;
```

The first `DEFINE` defines the `ssh` command to use: an ssh-connection will be made to the root account at the client.

The second `DEFINE` shows the arguments for **find**(1) when looking for all root setuid or setgid normal files. For all these files the **sha1sum**(1) program should be run.

5.2.2 the USE directives

```
USE BASE        /root/stealth/client
USE EMAIL       admin@elswhere
USE MAILER      /root/bin/stealthmail
USE MAILARGS    "Client STEALTH report"
```

```
USE SSH          ${SSHCMD}
```

- All output will be written under the `/root/stealth/client` directory.
- Mail will be sent to the user `admin@elsewhere`.
- The mail program will be a script (`stealthmail`), living in `/root/bin`.
- The script handles its own argument. As it can be used for other stealth-scans as well, it is given an argument which can be used as the subject when sending mail, identifying the computer that has been scanned.
- The ssh-command is defined by the `SSH-DEFINE`.
- the default values of all remaining `USE` directives can be used, and were therefore not specified. They are:

```
USE DD           /bin/dd
USE DIFF         /usr/bin/diff
USE PIDFILE      /var/run/stealth-
USE REPORT       report
USE SH           /bin/sh
```

5.2.3 the commands

First, we'll copy the client's sha1sum program to the controller. In practice, this should also include the shared object libraries that are used by sha1sum, as they might have become corrupted as well.

Obtain the client's sha1sum program

First, the `sha1sum` program is copied to a local directory

```
GET /usr/bin/sha1sum /root/tmp
```

This command must succeed.

Check the integrity of the client's sha1sum program

Next, we'll check the received `sha1sum` program, using our own:

```
LABEL \nCheck the client's sha1sum program
LOCAL CHECK LOG = local/sha1 /usr/bin/sha1sum /root/tmp/sha1sum
```

The `LABEL` command will write the label to the report file just before the output of the sha1sum program is generated.

The `LOCAL` command will check the sha1sum of the program copied from the client. The report is written on the file `/root/stealth/client/local/sha1`. If this fails, the program will not continue, but will alert `admin@elsewhere` that the check failed. This is of course rather serious, as it indicates that either the controller's `sha1sum` is behaving unexpectedly or that the client's `sha1sum` program has changed.

The `sha1sum` program *may* have changed due to a normal upgrade. If so, `admin@elsewhere` will know this, and can (probably) ignore the warning. The next time **stealth** is run, the (now updated) SHA1 value is used, and it should again match the obtained `SHA1` value from the copied `sha1sum` program.

Check the client's /usr/bin/find command

The client will use it's `find` command intensively: `find` is a great tool for producing files having almost any conceivable combination of characteristics. Of course, the client's `find` command itself must be ok, as well as the client's `sha1sum` program. Now that we know that the client's `sha1sum` program is ok, we can use it to check the client's `/usr/bin/find` program.

Note that the controller itself will not suffer any processing load here: only the client itself is taxed for checking the intergrity of its own files:

```
LABEL \nchecking the client's /usr/bin/find program
CHECK LOG = remote/binfind /usr/bin/sha1sum /usr/bin/find
```

Check the client's setuid/setgid files

Having checked the client's `sha1sum` and `find` programs, sha1 checksum checks should be performed on all setuid and setgid files on the client. For this we activate the `sha1sum` program on the client. In order to check the setuid/setgid files, the following command is added to the policy file:

```
LABEL \nsuid/sgid/executable files uid or gid root on the / partition
CHECK LOG = remote/setuidgid /usr/bin/find / ${EXECSHA1}
```

Check the configuration files in the client's /etc/ directory

Finally, the client's configuration files are checked. Some of these files change so frequently that we don't want them to be checked. E.g., `/etc/adjtime`, `/etc/mtab`. To check the configuration file, do:

```
LABEL \nconfiguration files under /etc
CHECK LOG = remote/etcfiles                          \
      /usr/bin/find /etc -type f -not -perm +6111    \
        -not -regex "/etc/\(adjtime\|mtab\)"         \
        -exec /usr/bin/sha1sum {} \;
```

5.2.4 The complete 'policy' file

Here is the complete policy file that we've constructed so far:

```
DEFINE  SSHCMD  /usr/bin/ssh root@client -T -q exec /bin/bash --noprofile
DEFINE  EXECSHA1 -xdev -perm +u+s,g+s \( -user root -or -group root \) \
                -type f -exec /usr/bin/sha1sum {} \;

USE BASE        /root/stealth/client
USE EMAIL       admin@elswhere
USE MAILER      /root/bin/stealthmail
USE MAILARGS    "Client STEALTH report"
USE SSH         ${SSHCMD}

USE DD          /bin/dd
USE DIFF        /usr/bin/diff
USE PIDFILE     /var/run/stealth-
USE REPORT      report
USE SH          /bin/sh

GET /usr/bin/sha1sum /root/tmp

LABEL \nCheck the client's sha1sum program
LOCAL CHECK LOG = local/sha1 /usr/bin/sha1sum /root/tmp/sha1sum

LABEL \nchecking the client's /usr/bin/find program
CHECK LOG = remote/binfind /usr/bin/sha1sum /usr/bin/find

LABEL \nsuid/sgid/executable files uid or gid root on the / partition
CHECK LOG = remote/setuidgid /usr/bin/find / ${EXECSHA1}

LABEL \nconfiguration files under /etc
CHECK LOG = remote/etcfiles                          \
      /usr/bin/find /etc -type f -not -perm +6111    \
```

```
        -not -regex "/etc/\(adjtime\|mtab\)"        \
        -exec /usr/bin/sha1sum {} \;
```

5.3 Running 'stealth' for the first time

When **stealth** is now run, it will create its initial report files under `root/stealth/client`.

The first time **stealth** is run, it is usually run 'by hand':

```
stealth policy
```

this will show all executed commands on the standard output, and will initialize the reports. Running **stealth** this way for the just constructed `policy` file results in the following output (lines were wrapped to improve readability):

```
GET /usr/bin/sha1sum /root/tmp
LABEL \nCheck the client's sha1sum program
LOCAL CHECK LOG = local/sha1 /usr/bin/sha1sum /root/tmp/sha1sum
LABEL \nchecking the client's /usr/bin/find program
CHECK LOG = remote/binfind /usr/bin/sha1sum /usr/bin/find
LABEL \nsuid/sgid/executable files uid or gid root on the / partition
CHECK LOG = remote/setuidgid /usr/bin/find / -xdev -perm +u+s,g+s
        \( -user root -or -group root \) -type f
        -exec /usr/bin/sha1sum {} \;
LABEL \nconfiguration files under /etc
CHECK LOG = remote/etcfiles                          /usr/bin/find /etc
        -type f -not -perm +6111   -not -regex "/etc/\(adjtime\|mtab\)"
        -exec /usr/bin/sha1sum {} \;
LOCAL /usr/bin/scp -q root@client:/usr/bin/sha1sum /root/tmp
LABEL \nCheck the client's sha1sum program
LOCAL CHECK LOG = local/sha1 /usr/bin/sha1sum /root/tmp/sha1sum
LABEL \nchecking the client's /usr/bin/find program
CHECK LOG = remote/binfind /usr/bin/sha1sum /usr/bin/find
LABEL \nsuid/sgid/executable files uid or gid root on the / partition
CHECK LOG = remote/setuidgid /usr/bin/find / -xdev -perm +u+s,g+s
        \( -user root -or -group root \) -type f
        -exec /usr/bin/sha1sum {} \;
LABEL \nconfiguration files under /etc
CHECK LOG = remote/etcfiles                          /usr/bin/find /etc
        -type f -not -perm +6111   -not -regex "/etc/\(adjtime\|mtab\)"
```

```
        -exec /usr/bin/sha1sum {} \;
```

This all produces the following output:

5.3.1 The mailed report

The `/root/bin/stealthmail` is called with the following arguments:

```
"Client STEALTH report" admin@elswhere
```

The contents of the mailed report now is (the date will of course change, the next time **stealth** is run):

```
STEALTH (1.21) started at Mon Nov 24 10:50:30 2003

Check the client's sha1sum program
Initialized report on local/sha1

checking the client's /usr/bin/find program
Initialized report on remote/binfind

suid/sgid/executable files uid or gid root on the / partition
Initialized report on remote/setuidgid

configuration files under /etc
Initialized report on remote/etcfiles
```

5.3.2 Files under /root/stealth/client

Under `/root/stealth/client` the following entries are now available:

- `local`: below this directory the reports of the locally performed checks are found. Using our demo `policy` file, only one logfile is found here: `sha1`, containing the client's SHA1 checksum of its `/usr/bin/sha1sum` program:

```
45251e259bfaf1951658a7b66c328c52  /root/tmp/sha1sum
```

- `remote`: at this directory the reports of the remotely performed checks are found. Using our demo `policy` file, three files were created:

The file `binfind`, containing the checksum of the client's `/usr/bin/find` program:

```
fc62fc774999584f1e29e0f94279a652  /usr/bin/find
```

The file `etcfiles`, containing the checksums of the client's configuration files under `/etc` (shown only partially):

```
ced739ecb2c43a20053a9f0eb308b2b0  /etc/modutils/aliases
a2322d7e2f95317b2ddf3543eb4c74c0  /etc/modutils/paths
f9e3eac60200d41dd5569eeabb4eddff  /etc/modutils/arch/i386
f07da2ebf00c6ed6649bae5501b84c4f  /etc/modutils/arch/m68k.amiga
2893201cc7f7556160fa9cd1fb5ba56a  /etc/modutils/arch/m68k.atari
    ...
bf73b4e76066381cd3caf80369ce1d0e  /etc/deluser.conf
4cd70d9aee333307a09caa4ef003501d  /etc/adduser.conf.dpkg-save
8c749353c5027d0065359562d4383b8d  /etc/gimp/1.2/gtkrc_user
3ec404ec597ef5460600cccf0192f4d6  /etc/gimp/1.2/unitrc
8c740345b891179228e3d1066291167b  /etc/gimp/1.2/gtkrc
```

The file `setuidgid`, containing the checksums of the client's setuid/setgid root files (shown only partially):

```
030f3f84ec76a8181cca087c4ba655ea  /bin/login
b6c0209547d88928f391d2bf88af34aa  /bin/ping
5d324ad212b2ff8f767637ac1a8071ec  /bin/su
344dbedc398d5114966914419ef53fcc  /usr/bin/wall
27b045bd7306001f9ea31bc18712d8b7  /usr/bin/rxvt-xpm
...
3567b18ffc39c2dc6ec0c0d0fc483f4f  /usr/lib/ssh-keysign
3383a7955ac2406311e9aa51c6ac9c2c  /usr/X11R6/bin/X
3c99ea0425c6e0278039e16478d2fb57  /usr/X11R6/bin/xterm
d590f7f5b4d6ae61680692a52235d342  /usr/local/bin/setuidcall
4c17203d7d91ec4946dea2f0ae365d5b  /sbin/unix_chkpwd
```

Of course, the checksums and the filenames shown are only for documentation purposes. At other systems this will show different files and/or checksums, no doubt.

- The file `/root/client/report` **New lines are always appended to the `/root/client/report` file. It will never shorten, unless shorten by the systems administrator at 'controller'.**

  This file contains the following:

```
STEALTH (1.21) started at Mon Nov 24 10:50:30 2003

Check the client's sha1sum program
Initialized report on local/sha1

checking the client's /usr/bin/find program
Initialized report on remote/binfind

suid/sgid/executable files uid or gid root on the / partition
Initialized report on remote/setuidgid

configuration files under /etc
Initialized report on remote/etcfiles
```

This completes the information created by **stealth** during its first run.

5.4 Running 'stealth' again

5.4.1 All files unaltered

When **stealth** is run again, it will update its report files under `root/stealth/client`. If nothing has changed, the log-files will remain unaltered. The new run will, however, produce some new info on the file `/root/client/report`:

```
STEALTH (1.21) started at Mon Nov 24 10:50:30 2003

Check the client's sha1sum program
Initialized report on local/sha1

checking the client's /usr/bin/find program
Initialized report on remote/binfind

suid/sgid/executable files uid or gid root on the / partition
Initialized report on remote/setuidgid

configuration files under /etc
Initialized report on remote/etcfiles


STEALTH (1.21) started at Mon Nov 24 10:54:35 2003
```

Note that just one extra line was added: a timestamp showing the date/time of the last run. The systems administrator may reduce/remove the report file

every once in a while to reclaim some disk space.

5.4.2 Modifications have occurred

Basically, three kinds of modifications are possible: additions, modifications, and removals. Here we'll show the effect all these changes have on **stealth**'s output.

For the example, the following changes were made to the `client`'s files:

- `/etc/motd` was changed
- the file `timezone~` was removed
- the file `/etc/motd.org` was created

Next, **stealth** was once again run, producing the following output:

- The following new info is now added to file `/root/client/report`:

```
STEALTH (1.21) started at Mon Nov 24 10:54:35 2003

configuration files under /etc
ADDED: /etc/motd.org
    < 945d0b8208e9861b8f9f2de155e619f9  /etc/motd.org
MODIFIED: /etc/motd
    < 7f96195d5f051375fe7b523d29e379c1  /etc/motd
    > 945d0b8208e9861b8f9f2de155e619f9  /etc/motd
REMOVED: /etc/timezone~
    > 6322bc8cb3ec53f5eea33201b434b74b  /etc/timezone~
```

  Note that all changes were properly detected and logged in the file `/root/client/report`.

- Furthermore, a matching report was sent by *mail*:

```
STEALTH (0.90) started at Mon Oct 28 11:28:43 2002


configuration files under /etc
ADDED: /etc/motd.org
    < 945d0b8208e9861b8f9f2de155e619f9  /etc/motd.org
MODIFIED: /etc/motd
    < 7f96195d5f051375fe7b523d29e379c1  /etc/motd
    > 945d0b8208e9861b8f9f2de155e619f9  /etc/motd
REMOVED: /etc/timezone~
    > 6322bc8cb3ec53f5eea33201b434b74b  /etc/timezone~
```

Note that the report *only* shows the info that was added to the */root/client/report* file.

The report itself could be beautified further. I myself use the following script to mail the report to the addressee:

```
#!/bin/bash

NAME=‘basename $0‘

tee /root/stealth/lastreport/$NAME | egrep -v ’^([[:space:]]|[[:space:]]*$)’ |
    sort | uniq | mail -s $1 $2
```

For the `client` computer, this little script will write the mailed report on a file `/root/stealth/lastreport/client`, overwriting its previous contents, will remove all lines beginning with blanks (thus trimming away the `diff`-generated lines), and will mail the `sort`ed and `uniq`ed lines using `mail`. The addressee (`admin@elsewhere`) will receive the following information:

```
ADDED: /etc/motd.org
MODIFIED: /etc/motd
REMOVED: /etc/timezone~
STEALTH (0.90) started at Mon Oct 28 11:28:43 2002
configuration files under /etc
```

In practice this suffices to have me take action if something out of the ordinary has happened.

- Finally, the file

  ```
  /root/stealth/client/remote/etcfiles
  ```

  was recreated, saving the old file as

  ```
  /root/stealth/client/remote/etcfiles.20021028-112851
  ```

  As remarked earlier (see section 3.3), many `logfile.YYMMDD-HHMMSS` files could eventually accumulate. As discussed in section 3.3, it might be considered to remove old log files every now and then.

5.5 Failing LOCAL commands

If the client's `sha1sum` program itself is altered, a serious situation has developed. In that case, further actions by **stealth** would be suspect, as their results might easily be currupted. Checks *will* proceed, but a warning is generated on the `report` file (and in the mail sent to `admin@elsewhere`:

```
STEALTH (1.21) started at Mon Nov 24 10:54:35 2003

Check the client's sha1sum program
MODIFIED: /root/tmp/sha1sum
    < fc62fc774999584f1e29e0f94279a652  /root/tmp/sha1sum
    > 45251e259bfaf1951658a7b66c328c52  /root/tmp/sha1sum

*** BE CAREFUL *** REMAINING RESULTS MAY BE FORGED

configuration files under /etc
REMOVED: /etc/motd.org
    > 945d0b8208e9861b8f9f2de155e619f9  /etc/motd.org
MODIFIED: /etc/motd
    < 945d0b8208e9861b8f9f2de155e619f9  /etc/motd
    > 7f96195d5f051375fe7b523d29e379c1  /etc/motd
```

(The report shows the removal of the previously added file `motd.org`, and the modification of `motd`. These are real, as the original `motd` file, modified earlier, was restored at this point).

5.5.1 Skipping (some) integrity checks

Some files may not require integrity checks. Automated processes may modify files which are not threatening the proper functioning of running programs or processes. In those cases a file can be prepared holding the absolute paths of files that can be skipped. Each file should appear on a line of its own without any additional information.

Stealth can be informed about this file using the `-s skippath` or `-skip-files skippath` option. The file holding the paths of the files to be skipped should be specified using absolute paths, one file per line. Initial and trailing blanks, empty lines and lines having a `#` as their first non blank character are ignored.

Here is an example:

```
stealth -e --skip-files /root/stealth/remote/skipping remote.pol
```

If a file `/etc/skipme` appears in the current logs which is thereafter added to the `skippath` file then the mail generated by **stealth** will once contain a line like the following:

```
SKIPPING: /etc/skipme
    > a7695bb2d019e60988e757a4b692acfe  /etc/skipme
```

The shown hash-value is the hash-value at the time of the stealth-run reporting the `SKIPPING` message.

5.6 Automating 'stealth' runs using 'cron'

In order to automate the execution of **stealth**, a file `/etc/cron.d/stealth` could be created, containing a line like (assuming **stealth** lives in `/usr/bin`):

```
2,17,32,47 * * * *  root    test -x /usr/bin/stealth && \
                             /usr/bin/stealth -q /root/stealth/client.pol
```

This will start **stealth** 2 minutes after every hour. Alternate schemes are left to the reader to design.

In general, randomizing events makes it harder to notice them. **stealth** may start its tasks at a random point in time if its `-i` flag (for *random interval*) is used. This flag expects an argument in seconds (or in minutes, if at least an `m` is appended to the interval specification). Somewhere between the time **stealth** starts and the specified interval the scan will commence. For example, the following two commands have identical effects: the scan is started somewhere between the moment **stealth** was started and 5 minutes:

```
stealth -i 5min -q /root/stealth/client.pol
stealth -i 300  -q /root/stealth/client.pol
```

When the `-d` flag is given, the `-i` flag has no effect.

As another alternative, **stealth** my be started specifying the `-keep-alive pidfile` option. Here, `pidfile` is the name of a file that will contain the process id of the stealth process running in the background. For example:

```
stealth --keep-alive /var/run/stealth -i 300  -q /root/stealth/client.pol
```

Now, **cron**(1) may be used to restart this process at indicated times:

```
    2,17,32,47 * * * *  root    test -x /usr/bin/stealth && \
                                   /usr/bin/stealth --rerun /var/run/stealth
```

As yet another alternative, the cron-job may activate a script performing **stealth**'s rerun, starting another **stealth** run if necessary. The advantage of such an approach is that **stealth** is automatically started after, e.g., a reboot. The following script expects two arguments (both of which must be absolute paths). The first argument is the path to the *pidfile* to use, the second argument is the path to the policy file to use. The script is found in the distribution package as `/usr/share/doc/stealth/usr/bin/stealthcron`:

```
#!/bin/bash

PROG=‘basename $0‘
STEALTH=/usr/bin/stealth

testAbsolute()
{
    echo $1 | grep "^/" > /dev/null 2>&1 && return

    echo "\‘$1’ must be absolute path"
    exit 1
}

case $# in
    (2|3)
        testAbsolute $1
        testAbsolute $2
        if [ "$3" != "" ]
        then
            testAbsolute $3
            SKIP="-s $3"
        fi

        if [ -x ${STEALTH} ] ; then
            ${STEALTH} --rerun $1
            [ $? -eq 0 ] || ${STEALTH} --keep-alive $1 ${SKIP} -q $2
        fi
    ;;

    (*)
    echo "
$PROG by Frank B. Brokken (f.b.brokken@rug.nl)
Usage: $PROG pidfile configfile [skipfile]
where:
```

```
    pidfile:    absolute path to pidfile to be used by ${STEALTH}
    configfile: absolute path to configuration file to be used by ${STEALTH}
    skipfile:   absolute path to file holding files to skip (optional)

    calls $STEALTH} --rerun pidfile.
    If that fails,
        ${STEALTH} --keep-alive pidfile -q configfile
    or (if skipfile was specified)
        ${STEALTH} --keep-alive pidfile -s skipfile -q configfile
    is started.
"
    exit 1
    ;;
esac
```

The script could be called from `/etc/cron.d/stealth` using a line like

```
22  8  * * * root test -x /usr/bin/stealthcron && /usr/bin/stealthcron
        /var/run/stealth.target /usr/share/stealth/target.pol
```

Note that the command should be on a single line. It was spread out here over two lines to enhance readability. Also note that the directory `/var/run` is usually not writable to non-root users. If `stealth` is used by a non-root user another directory should be used to store `stealth.target` in.

5.7 Report File Rotation

When **stealth** performs integrity scans it will append information to the report file. This file will therefore eventually grow to a large size, and the systems manager controlling **stealth** might want to *rotate* the report file every once in a while (e.g., using a program like **logrotate**(1), also see the upcoming section 5.7.2). In order to ensure that no log-rotation takes place while **stealth** is busy performing integrity scans (thus modifying the report file) the options –**suppress** and `-resume` were implemented. Both options require the process-ID file of currently active **stealth** process as their argument.

For example, if a **stealth** process was once started using the command

```
    stealth -q --keep-alive /var/run/stealth.small --repeat 900 \
                    /var/stealth/policies/small.pol
```

then the `-suppress` and `-resume` commands for this process should be formulated as:

```
stealth --suppress /var/run/stealth.small
stealth --resume /var/run/stealth.small
```

The **stealth** process identified in the files provided as arguments to the `-suppress` and `-resume` options is called the *targeted stealth process* below.

The `-suppress` option has the following effect:

- If the targeted **stealth** process is currently processing its policy file, performing a (new) integrity scan, then the currently executing policy file command is completed, whereafter further commands are ignored, except for `-resume` (see below) and `-terminate`.
- Any scheduled integrity scans following the `-suppress` command are ignored for the targeted **stealth** process;
- The targeted **stealth** process will write a message that it is being suppressed to the report file and will then process the report file as usual;
- The targeted **stealth** process relinquishes its control over the report file;
- The command '**stealth** `-suppress pidfile`' terminates.

Now that the report file will no longer be affected by the targeted **stealth** process, log-rotation may take place. E.g., a program like **logrotate**(1) allows its users to specify a command or script just before log-rotation takes place, and '**stealth** `-suppress pidfile`' could be specified nicely in such a pre-rotation section.

The `-resume` option has the following effect:

- The targeted **stealth** process resumes its activities by performing another integrity scan. Thus, `-resume` implies `-rerun`.
- Any scheduled integrity scans following the `-resume` command are again honored by the targeted **stealth** process, following the completion of the `-resume` command.
- The command '**stealth** `-resume pidfile`' terminates.

Note that, once `-suppress` has been issued, all commands except `-resume` and `-terminate` are ignored by the targeted **stealth** process. While suppressed, the `-terminate` command is acknowledged as a 'emergency exit' which may or may not interfere with, e.g., an ongoing log-rotation process. The targeted **stealth** process should not normally be terminated while it is in its suppressed mode. The normal way to terminate a stealth process running in the background is:

- Wait for the targeted **stealth** process to complete a series of integrity scans;
- Issue the '**stealth** `-terminate pidfile`' command.

5.7.1 Status file cleanup

Whenever **stealth** is run and it encounters a modified situation the already existing status file that is used to summarize that particular situation is saved and a new status file is created. Eventually, this will result in many status files. While report files can be rotated, it is pointless to rotate old status files, since they never are modified. Instead status files exceeding a certain age could be removed and more recent files might be zipped to conserve space. In **stealth**'s binary distribution the file `/usr/share/doc/stealth/usr/bin/stealthcleanup` is provided which can be used to perform this cleanup. The script expects one argument: a resource file defining the following shell variables:

- `directories`: the directories below which the status files are found;
- `gzdays`: the number of days a status file must exist before it is compressed using **gzip**(1);
- `rmdays`: the maximum age (in days) of compressed status files. Files exceeding this age are removed using **rm**(1).

Here is the `stealthcleanup` script as it is found in the binary distribution's `/usr/share/doc/stealth/usr/bin` directory:

```
#!/bin/bash

usage()
{
    echo "
Usage: $0 [-v] rc-file
Where:
    -v: Show the actions that are performed
    rc-file: resource file defining:
            \‘directories’ - one or more directories containing status files
            \‘gzdays’      - number of days status files may exist before they
                             are compressed
            \‘rmdays’      - number of days gzipped status files may exist
                             before they are removed.
"
    exit 1
}


error()
{
    echo "$*" >&2
    exit 1
}

if [ "$1" == "-v" ]
```

```
then
    verbose=1
    shift 1
else
    verbose=0
fi

[ $# == 1 ] || usage

# now source the configuration file
. $1

for x in $directories
do
    cd $x || error "\‘$x’ must be a directory"
    if [ $verbose -eq 1 ]
    then
        echo "
cd $x"
    fi

    if [ $verbose -eq 1 ]
    then
        echo \
    /usr/bin/find ./ -mtime +$rmdays -type f -regex '.*[0-9]+-[0-9]+\.gz' \
        -exec /bin/rm {} \;
    fi
    /usr/bin/find ./ -mtime +$rmdays -type f -regex '.*[0-9]+-[0-9]+\.gz' \
        -exec /bin/rm {} \;

    if [ $verbose -eq 1 ]
    then
        echo \
    /usr/bin/find ./ -mtime +$gzdays -type f -regex '.*[0-9]+-[0-9]+' \
        -exec /bin/gzip {} \;
    fi
    /usr/bin/find ./ -mtime +$gzdays -type f -regex '.*[0-9]+-[0-9]+' \
        -exec /bin/gzip {} \;
done

exit 0
```

Assuming that the status files are written in `/var/stealth/target/local` and `/var/stealth/target/remote`; that status file should be compressed when older than 2 days and removed after 30 days, the resource file is:

```
directories="
    /var/stealth/target/local
    /var/stealth/target/remote
    "

rmdays=30
gzdays=3
```

Furthermore assuming that the resourcefile is installed in `/etc/stealth/cleanup.rc` and the `stealthcleanup` script itself in `/usr/bin/stealthcleanup`, the `stealthcleanup` script could be called as follows:

```
    /usr/bin/stealthcleanup /etc/stealth/cleanup.rc
```

Note that `stealthcleanup` may be called whether or not there are active **stealth** processes, as **stealth** does not use status files anymore once they have been written.

5.7.2 Using 'logrotate' to control report- and status files

A program like **logrotate**(1) allows its users to specify a command or script immediately following log-rotation, and '**stealth** `-resume pidfile`' could be specified nicely in such a post-rotation section.

Here is an example of a specification that can be used with **logrotate**(1). Logrotate (on Debian systems) keeps its configuration files in `/etc/logrotate.d`, and assuming there is a host `target`, whose report file is `/var/stealth/target/report`, the required **logrotate**(1) specification file (e.g., `/etc/logrotate.d/target` could be:

```
/var/stealth/target/report {
    weekly
    rotate 12
    compress
    missingok
    prerotate
        /usr/bin/stealth --suppress /var/run/stealth.target
    endscript
    postrotate
        /usr/bin/stealth --resume /var/run/stealth.target
    endscript
}
```

Using this specification file, **logrotate**(1) will

- Perform weekly rotations of the report file;

- Keep up to 12 rotated files, compressing them using **gzip**(1);
- Before rotating the report file, **stealth**'s actions are suppressed;
- Following the rotation, **stealth**'s actions are resumed

Note thet `stealth -resume xxx` will always start with another file integrity scan.

Chapter 6

Kick starting 'stealth'

Here are the steps to take to kick-start **stealth**

- Install the stealth Debian package `stealth_2.04.00_i386.deb` and thus accept the provided binary program (skipping the next three steps) or do not accept the provided binary, and compile **stealth** yourself, as per the following steps:
- Unpack `stealth_2.04.00.tar.gz`: `tar xzvf tealth_2.04.00.tar.gz`
- `cd stealth`
- Inspect the values of the variables in the file `INSTALL.cf` Modify these values when necessary.
- Make sure the bobcat library has been installed. (`http://bobcat.sourceforge.net`)
- Run '`./make/program`' to compile **stealth**. Note: it's *not* '`make program`'
- Run (probably as root) '`./make/install`' to install. Note: it's *not* '`make install`'

Following the installation nothing in the `stealth` directory tree is required for **stealth**'s proper functioning, so consider removing it.

Compiling **stealth** assumes that `g++` version 3.3 (or higher) is available. If not: install it first.

Next, do:

- `cp stealthmail /usr/local/bin`
- `mkdir /root/stealth`
- `cp local.pol /root/stealth`

`ssh` and `sh` should be available. `root@localhost` should be able to login at `localhost` using `ssh root@localhost`, using the `/bin/bash` or `/bin/sh` shell. Check (as 'root') at least

```
ssh root@localhost
```

as this might ask you for a confirmation that you've got the correct host. Now, run

```
stealth /root/stealth/localhost.pol
```

to initialize the stealth-report files for `localhost`. This will initialize the report for:

- all root setuid/setgid executable files on `localhost`,
- and for all files under `/etc/` on `localhost`.

The mail-report is written on `/tmp/stealth-2.04.00.mail`

Now change or add or remove one of these files, and rerun **stealth**. The file `/tmp/stealth-2.04.00.mail` should reflect these changes.

Chapter 7

Usage info

When **stealth** is started without arguments, it provides some help about how to start it. A message like the following is produced:

```
stealth by Frank B. Brokken (f.b.brokken@rug.nl)

stealth V2.04.00
SSH-based Trust Enhancement Acquired through a Locally Trusted Host
Copyright (c) GPL 2005-2012

Usage 1:
   stealth options policy
Where:
   options: (long options between parentheses) select from:
       -c: (--parse-config-file) process the config file,
           no further action, report the results to std output.
       -d: (--debug) write debug messages to std error
       -e: (--echo-commands) echo commands to std error when they
           are processed (implied by -d)
       -h (--help): show this usage info
       -i <interval>[m]: (--random-interval) start the scan between now and
           a random interval of interval seconds, or minutes if an 'm' is
            appended to the specified interval.
            Requires --repeat and --keep-alive.
       -n: (--no-child-processes) no child processes are
            executed: child actions are faked to be OK.
       -o: (--only-stdout) scan report is written to stdout. No mail is sent.
       -q: (--quiet) suppress progress messages to stderr.
       -r <nr>: (--run-command) only run command <nr> (natural  number).
       -s <skipfiles>: (--skip-files) skip the integrity checks of
            files having their absolute path names listed in the file
            'skipfiles'
       -v: (--version): display version information (and exit).
```

```
   --keep-alive pidfile: keep running as a daemon, wake up at interrupts.
   --max-size <size>[BKMG]: files retrieved by GET may at most
           have <size> bytes (B), Kbytes (K), Mbytes (M), Gbytes (G)
           default: 10M, default spec. unit: B
   --repeat <seconds>: keep running as a daemon, wake up at
           interrupts. or after <seconds> seconds.
           Requires --keep-alive.
   --usage: provide this help (and exit)
   --help: provide this help (and exit)
   policy: path to the policyfile

Usage 2:
   stealth [--rerun|--resume|--suppress|--terminate] pidfile
Where:
   --rerun: restart a stealth integrity scan
   --resume: resume stealth following --suppress
   --suppress: suppress stealth activities
   --terminate: terminate stealth
   pidfile: file containing the pid of the stealth process to rerun or
            terminate.
```

Note that with the second type of usage the policy file is not required: here only the `pidfile` must be specified.

Chapter 8

Errormessages

`/bin/sh: no output from /usr/bin/diff ...`

the actual program names appearing here could change due to your local configuration. The defaults are shown. This indicates that the `/usr/bin/diff` program could not be activated on the controller. Check the correctness of both the shell program (`/bin/sh`) and the diff program (`/usr/bin/diff`): Do they exist? Have their paths been specfied well? Note that filenames passed to `diff` might not exist anymore when the program terminates. This should not be the cause of the error.

`Can't chdir to 'path'`

the directory `path` could not be created/used. This may be a permission problem. Check the permissions of `path` if `path` does actually exist. The problem may be in a path component, not necessarily in the last element of the path.

`Can't open /dev/null`

This message may be generated by a child-process: `sh`, `ssh` or `diff`. It is generated when the child process could not redirect its standard error messages to the standard error stream. If it appears then there is probably something incorrect in your `/dev/` directory: check the availability of `/dev/null`, check if you can copy a file to `/dev/null`.

`Can't open ... to write`

This message may be generated when the mentioned log-file could not be written to. Check the permissions of the file, check if the path to the file exists. The problem may be in a path component, not necessarily in the last element of the path or in the file itself.

`Can't read ...`

the mentioned file could not be read. Check if the file exists, and if you have read permissions for it.

`Config line '...' invalid`

The mentioned line of the specified policy file was ill-formed. Check the line's contents against the description of the policy file.

`ConfigSorter file processed`

In this case, the `-c` option has been given. When `-c` was provided, **stealth** stops after having processed the configuration file.

`Corrupt line in policy file: ...`

The apparently corrupted line is shown. The line is corrupted if the line could not be split into an initial word and its remainder. Normally this should not happen. As the line is mentioned, the message itself should assist you in your repairs.

`Inserting command '...' failed.`

the mentioned command could not be sent to a child-process (`sh` or `ssh`). Check the availability of the `ssh` connection to the client. The command itself might also be unacceptable.

`Invalid interval for -i.`

The -i flag was given an invalid (too large or negative) argument.

`Non-zero exit value for '...'`

A local command (not using the `CHECK` keyword), returned with a non-zero exit. This will terminate further processing of the policy file. Inspect and/or rerun the command 'by hand' to find indications about what went wrong. The report file or the standard error stream may also contain additional information about the reason of the failure.

`Unable to create the logfile '...'`

the mentioned log file could not be created. Check the permissions of the file, check if the path to the file exists. The problem may be in a path component, not necessarily in the last element of the path or in the file itself.

`USE SSH ... entry missing in the configuration file`

there is no default for the `USE SSH` specification in the policy file.
The specification could not be found. Provide a specification like:

```
USE SSH     ssh -q root@localhost
```