Stealth

Frank B. Brokken[*]
(f.b.brokken@rc.rug.nl)
Computing Center, University of Groningen

June 2003

Introduction

Hontañón (2001) starts his chapter on 'System Monitoring and Auditing' with the following sentence:

> Monitoring your system for abnormal behavior is an essential task in both system administration and information security

The normal tasks performed by computers may be interrupted, altered or suppressed by intruders gaining access to these computers. By keeping a close watch on what's happening, attempts to invade computer systems may be detected and possibly prevented. As an 'early warning' signalling system, and as a damage-reporting facility, programs that automatically check the log-files of computers are therefore very useful.

In a similar vein, Garfinkel and Spafford (1996) discuss various ways to prevent or minimize the damage that may result from people trying to attack computer systems. Among other things, they discuss the need for backups, they cover possible routes intruders may take to invade computer systems, and they discuss the need for auditing and logging as well.

When a computer becomes the target of an attack, basically two things may happen:

- First, the attacker may be interested in the mere disruption of the normal operations of the computer that is attacked. Denial of Service attacks

[*]The author would like to thank Hans Gankema and Kees Visser, both at the Computing Center of the University of Groningen, for their valuable remarks leading to the construction of *STEALTH*.

fall into this category as it is the attacker's intent to degrade facilities normally offered by the attacked computer.

- Secondly, the attacker may be interested in gaining access to a computer. Access may be pursued for various reasons: installing own facilities (ftp-servers, game servers); using resources made available by the company or organization owning the attacked computer, such as fast Internet access; using the computer as a platform to perform further attacks (thus covering the intruder's tracks, giving a false impression of the origin of attacks of yet other computers).

Although denial of service attacks are a nuisance, most systems administrators are more worried about the possibilities intruders have when they actually gain access to the computers being maintained and controlled by the administrators. Once an intruder has gained access to a computer it's not clear what the attacker may have accomplished, it's unclear which software may have been modified, and it's not evident what backdoors or possible traps the intruder may have installed. Backups, of course, don't offer any relief, as the backup may actually have *saved* the modifications inflicted upon the system by the intruder.

Therefore, software has been developed allowing the systems administrator to detect changes to the computer's software. These solutions are discussed in the context of topics like 'file integrity auditing' (Hontañón, 2001) and 'Integrity Management' (Garfunkel and Spafford, 1996).

The basic idea behind these file integrity monitoring tools is that an intruder, gaining access to a computer system, will change the system's software. Changes may involve removal of, additions to, or modifications of available software. Once the intruder has gained access to a system and starts to modify things, there's no telling what may have happened, and the sooner changes are detected, the better.

The *Sans institute* (`http://www.sans.org`) offers an overview of existing software which may be used for integrity checking. According to an overview presented by *Sans* (`http://www.sans.org/rr/audit/aide.php`) the following file integrity checking systems are currently available:

- *Tripwire* (`http://www.tripwire.com`): Tripwire is a policy driven file system integrity checking tool that allows system administrators to verify the integrity of their data. Tripwire compares properties of designated files and directories against information stored in a previously generated database.

- *Fcheck* (`http://sites.netscape.net/fcheck/`): Fcheck is a stable open-source Perl script that provides host intrusion detection and policy enforcement on servers. The script uses comparative system snapshots based on file checksums - very similar to Tripwire, but with a simpler design.

- *Filetraq* (`http://filetraq.xidus.net/`): FileTraq is a shell script designed to be run periodically from the root crontab. Each time, it compares a list of system files with the copies that it is keeping. Any changes are reported in diff or patch file style, and dated backup copies are preserved.
- *OsirisScripts* `http://www.shmoo.com/osiris/`: Simple Perl scripts for generating a catalog of MD5 hashes of executable files and then comparing the catalog to new snapshots. Compares based on missing or additional files, differing MD5 hashes, modification dates, and file attributes.
- *Sentinel* (`http://zurk.netpedia.net/zfile.html`): Sentinel is a fast file/drive scanning utility similar to the Tripwire and Viper.pl utilities.
- *Sherpa* (`http://www.nbank.net/~rick/sherpa/`): Sherpa is a Perl tool for configuring and then checking system security. It allows an admin to maintain a custom database of file and directory permissions and ownership attributes as local needs dictate.
- *Aide* (`http://www.cs.tut.fi/~rammer/aide-0.7.tar.gz`): Aide (Advanced Intrusion Detection Environment) constructs a database of the files specified in aide.conf, aide's configuration file. The Aide database stores various file attributes including: permissions, inode number, user, group, file size, mtime and ctime, atime, growing size and number of links. The binary, configuration file, policy file and database of the file integrity checker itself can be manipulated. In case of Aide the policy file and configuration file are identical. It provides a method for transferring the configuration file and database to a floppy disk for increased security.

When studying the characteristics of these file integrity checkers, several things spring to mind:

- First, the data summarizing the state of the information that must be protected are commonly stored on the actual systems that undergo the integrity check. This potentially poses a serious problem, as intruders may be able to tamper with this information in such a way that it won't reflect the changes made by the intruder.
- Secondly, as far as management is concerned, each system needs its own integrity checker. This places an extra burden on the often already oversized task load of the systems administrators.
- Thirdly, most integrity checkers allow a certain set of snapshots to be taken. For example, Aide allows many attributes and many cryptographical summary statistics to be computed, but supports no other features. This is unfortunate, as it puts a serious burden on the ingenuity of the developers of these integrity checkers (did I choose the right set and a

complete set of gadgets?) and these integrity checkers cannot adapt easily to newer tools, producing newer statistics, that may become available in the future.

- Fourthly, most integrity checkers themselves are stored on the systems to be protected. Usually, at some point or location, they leave information or traces behind that they are there, such as crontab entries, binaries, logfiles, etc. This information may be used by intruders to actually prevent the integrity check from happening: crontab entries may be modified, data bases may be restored, binaries could be modified, etc.

In order to remedy these flaws which are, at least to some extent, present in current file integrity checkers, the *STEALTH* program was developed. Specifically, *STEALTH* provides a file integrity checking facility that:

- Leaves no trace on the computer being checked;
- Is very difficult to detect or evade by intruders;
- Cannot be modified by intruders;
- Can be used to perform integrity checks on many computers, without the associated maintenance burden.
- Can be used to check many different types of computers and operating systems without the need to install *STEALTH* on these computers. Porting *STEALTH* to these different computers and operating systems systems should be no problem, but *STEALTH* can be used on such systems without actually completing one single port.

STEALTH - Concepts

The philosophy behind *STEALTH* is that a file integrity checker can be installed on a single (central) computer, which computer may then use *secure connections* to communicate with other computers. Controlled by *STEALTH*, these other computers will then perform their own integrity checks.

Considering the stealthy nature of this process, in which the controlling computer (running *STEALTH*) is almost always invisible to the computers that are being controlled (and when visible, it's only indirectly visible), *STEALTH* is an apt name for such a program.

While finding the name *STEALTH* was rather easy, it turned out to be kind of hard to find an appropriate expansion for its name. But here it is. *STEALTH* can be expanded to:

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

The following key terms are used in this expansion:

- *Ssh-based*: the computer on which *STEALTH* has been installed communicates with other computers using an encrypted (ssh) connection. Usually the computers being scanned (called *clients*) and the computer initiating the scan (called the *controller*) are different computers. Clients should accept incoming ssh-connections from the controller. The controller doesn't have to accept incoming ssh connections at all (and it *shoudn't*, probably).
- *Trust Enforcement*: The whole purpose of a file integrity scan is to be able to *trust* the integrity of the inspected computer's software. So, by performing file integrity checks, we *enforce trust* on the clients, due to the observed integrity of their files.
- *Locally Trusted Host*: clients apparently trust their controllers, as they allow the controller to open ssh-connections to them. Clients therefore *locally trust* their controllers. Hence, *Locally Trusted Host.*

As the controller is able to access clients using an ssh-connection, all communication between the controller and the clients is secure, and all commands performed on the clients, on behalf of the controller, 'leave no trace' on the clients. As there are no sediments on the clients subsequent to integrity scans, intruders will have no clue about any integrity check that has ever been performed on the clients. *STEALTH* truly has stealthy characterstics.

On the other hand, the controller's security can be *very* strict: it doesn't have to accept *any* inbound connections, and normally it needs to accept only two outbound connections: it should allow outbound *ssh* connections to its clients, and (normally) it should allow outgoing e-mail connections. The e-mail connections are used to inform the involved clients' systems administrators about any changes that were observed.

For each client whose file integrity is checked by *stealth* at least one *policy* file is constructed. The policy file determines which actions should be taken during the integrity check. Multiple policy files may be associated with a single client, defining various levels of integrity scans: e.g., frequently performed superficial scans and incidentally performed thorough scans.

The policy file contains commands which are executed on either the controller or on the clients. Usually, only a few or lightweight commands are run on the controller. However, most commands are run on the clients. This approach minimizes the controller's load, and affects clients only in relation to their own file integrity. Any computer, even an old or cheap one, can still function perfectly well as a *STEALTH* controller, controlling the file integrity of many other, more up-to-date or expensive computers.

This is what happens when *STEALTH* is run to perform a client's integrity check:

- First, the client's *policy file* is read. The policy file defines the actions to be performed, and the values of several variables that are used by *STEALTH*.
- Secondly, the controller opens a command shell at the client using *ssh*, and it opens a command shell on the controller itself using *sh*.
- Thirdly, commands defined in the policy file are executed in their order of appearance. Examples will follow. Normally, return values of these commands are inspected. Non-zero return values will terminate *STEALTH* prematurely.
- Finally, the commands that are requested in the policy files normally produce output. Differences between the output generated during subsequent runs of *STEALTH* are logged on a *report file*, to which information is always appended. When this happens, the differences can be e-mailed to a particular systems administrator for further handling. *STEALTH* follows the 'dark cockpit' approach in that no e-mail is ever sent when no changes are detected.

Many (if not all), integrity tests can actually be performed using the `unix` (`Linux`, `Cygwin` on MS-Windows systems) `find` program, where `find` itself may start programs like `ls`, `md5sum` or even its own `-printf` method to produce file-integrity related statistics. As these programs are practically universally installed, their existence, nor their use will cause any suspicion. Of course, intruders may modify these tools. Fortunately, it is rather simple to check for this unwelcome situation.

STEALTH - Practical Use

Currently, *STEALTH* may be obtained from the University of Groningen's ftp server. The *STEALTH* tarball is located at:

```
ftp://ftp.rug.nl/contrib/frank/software/linux/stealth
```

The tarball contains *STEALTH*'s sources, as well as its *user guide*.

After installing *STEALTH*, at least one *policy file* must be constructed for each client to be checked by *STEALTH*. The construction of policy files is briefly summarized in the next chapter. It is covered more extensively in *STEALTH*'s user manual, which is part of *STEALTH*'s distribution.

In this chapter *STEALTH*'s practical use is illustrated, assuming the existence of a computer acting as a controller (here given the generic name `control.domain`) and a client (using the generic name `client.domain`). The steps to take to actually start using *STEALTH* are now discussed. This will offer further insight into the way *STEALTH* operates.

The following steps must be taken:

- At `client.domain` an account must be defined accepting incoming *ssh* connections from `control.domain`. Since using *STEALTH* is all about trust, it's probably safe to grant *root-access rights* to `client.domain` from `control.domain`. This allows `control.domain` to perform every required integrity check, without running against restricted access barriers.
- *STEALTH* can now run. It will produce output, which will inform the systems administrator about the integrity status of `client.domain`'s files. The kind of output that is produced is discussed in some detail in this chapter.
- `Control.domain` may perform integrity checks of `client.domain` at certain moments in time (e.g., using a periodic command scheduler, like the `unix` program `cron`). To reduce the predictability of these checks, *STEALTH* offers the possibility to postpone the actual onset of an integrity scan by a randomly chosen delay, thus making it harder to detect at `client.domain` that its files are being integrity-checked.

The abovementioned steps are now be discussed in more detail.

Granting access

Access to `client.domain` by `control.domain` is granted via the *ssh* protocol. `Client.domain` will normally allow `control.domain` to establish a connection without the need to identify itself using a user name and password.

This is realized using *public key* technology. The SSH-documentation can be consulted for details. Also, the *STEALTH* user manual contains information about how to set up such a trusted access route from `control.domain` to `client.domain`.

It is stressed here that only a trusted access route *from* `control.domain` *to* `client.domain` is required. As far as *STEALTH* is concerned, there is *absolutely no need* for allowing *any* incoming connection to `control.domain`. It is strongly advised *not* to allow *any* incoming connection to `control.domain`. By denying incoming connections, the security of `control.domain` (and by implication: of `client.domain`) is enhanced.

Once an ssh-connection, not requiring the specification of a password, can be established from `control.domain` to `client.domain`, `client.domain`'s policy file living at `control.domain` (e.g., the file `client.pol`) may now be given a line specifying how to establish this connection. E.g.,

```
    USE SSH /usr/bin/ssh root@client.domain -q
```

Initially the *ssh* connection between `client.domain` and `control.domain` should be established 'by hand' (i.e., not using *STEALTH*). After this initial connection, `cient.domain`'s ssh-key fingerprint will be available in `control.domain`'s list of *known hosts*.

Now that an ssh-connection can be established, the next step can be performed: actually running *STEALTH*.

Running STEALTH

When *STEALTH* is run for the first time, it will create an initial report file at a configurable location on `control.domain`.

Assuming `control.domain`'s policy file is `/root/stealth/client.domain.pol`, *STEALTH* is simply started using the command:

```
    stealth /root/stealth/client.domain.pol
```

This will show, to the standard output, all executed commands, initializing various log files during the process. Usually commands will target particular sections of `client.domain`'s file system, like all setuid / setgid files that live on `client.domain`.

The results of the actions performed by *STEALTH* are now e-mailed to a configurable e-mail address. The e-mail may be sent directly, or via a script, which may be used to, e.g., encrypt the report before sending it out.

The contents of the mailed report, reporting the initialization of the log files, is simply a date/time stamp and a list of files which are initialized. For example:

```
  STEALTH (1.11) started at Mon Apr  7 20:42:31 2003

  Check the client's md5sum program
```

```
Initialized log on local/md5

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

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

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

The initialized report files

During *STEALTH*'s first run, initial log files are created in the (configurable) directory `/root/stealth/client.domain`. The initialization report mentions the files that were created. Note that these log files are *not* stored on `client.domain` itself, but only on `control.domain`. Therefore, they are inaccessible for intruders of `client.domain`. Also, these intruders will find no traces on `client.domain` which may suggest that every now and then an integrity scan is performed.

The way these logs are generated is highly configurable. Usually they will contain combinations of checksums, filenames, and other statistics that are vital to the system's integrity. For example, checking the setuid/setgid files on `client.domain` might result in an overview containing lines like:

```
030f3f84ec76a8181cca087c4ba655ea  /bin/login
b6c0209547d88928f391d2bf88af34aa  /bin/ping
3c99ea0425c6e0278039e16478d2fb57  /usr/X11R6/bin/xterm
4c17203d7d91ec4946dea2f0ae365d5b  /sbin/unix_chkpwd
```

Depending on the way the policy file was constructed, several of these log files may have been generated. Their initial creation (usually at *STEALTH*'s first run) establish a baseline integrity status of `client.domain`. Having created this baseline status, *STEALTH* is now ready for its operational task: monitoring the file integrity of `client.domain`.

Re-running STEALTH: no modifications observed

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

```
STEALTH (1.11) started at Mon Apr  7 20:42:31 2003

Check the client's md5sum program
Initialized log on local/md5

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

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

configuration files under /etc
Initialized log on remote/etcfiles

STEALTH (1.11) started at Mon Apr  7 21:12:16 2003
```

Note that just one extra line was added: a *timestamp* showing the date and time of the last run. The systems administrator may reduce/remove the report file every once in a while to prevent it from growing unlimitedly.

Re-running STEALTH: modifications were observed

Basically, three kinds of modifications are possible: additions, modifications, and removals. The effects of these changes on *STEALTH*'s output are illustrated below.

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

- a new `/bin/login` program was installed
- the program `/sbin/unix_chkpwd` was removed
- a program `/usr/bin/setuidcall` was installed

Next, *STEALTH* was run again. This produced the following:

- New lines were added to the report file `/root/client.domain/report`:

```
STEALTH (1.11) started at Mon Apr  7 21:37:44 2003

suid/sgid/executable files on the / partition
```

```
ADDED: /usr/bin/setuidcall
 < 945d0b8208e9861b8f9f2de155e619f9  /usr/bin/setuidcall
MODIFIED:
 < 7f96195d5f051375fe7b523d29e379c1  /bin/login
 > 030f3f84ec76a8181cca087c4ba655ea  /bin/login
REMOVED:
 > 4c17203d7d91ec4946dea2f0ae365d5b  /sbin/unix_chkpwd
```

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

- This report was sent by e-mail to the proper system's administrator.
- The file `/root/stealth/client.domain/remote/setuidgid` was recreated, containing the overview of all actual setuid/setgid files. However, the old file is still available as the file

  `/root/stealth/client.domain/remote/setuidgid.20030407-213744`

Of course, over time these old files will be considered obsolete. Whenever appropriate, time-stamped log files may be removed. In any case, they are not used by *STEALTH*, as *STEALTH* only uses the most recent log files.

Calling STEALTH automatically and unpredictably

In order to automate the execution of *STEALTH*, it can be started by a periodic command scheduler, like the `unix` command `cron`. Assuming the availability of the `cron` program, automatically running *STEALTH* can be realized by creating, again on `control.domain`, the file `/etc/cron.d/stealth`. This file may be given a line like:

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

This will cause *STEALTH* to start at two minutes after every quarter of an hour. Alternate schemes are left to the reader to design.

In general, however, randomly occurring events are harder te detect. Therefore, *STEALTH* may start its job at a randomly chosen point in time. For this, *STEALTH*'s `-i` flag (or `--random-interval`) can be 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 period the integrity 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 later:

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

Once again, note that *no* information about or related to *STEALTH* is available on `client.domain`. Only when the integrity scan itself takes place something will be visible: an `ssh`-connection from `control.domain`, and a command (like `find`) is momentarily executed. After a short while, the `ssh` connection has disappeared and nothing at `client.domain` provides any clue about any integrity check that was ever performed. No log files, no report file, no programs, no cron-job specifications.

STEALTH - Configuration

As noted earlier, *STEALTH* reads policy files to determine the actions it should perform. Each policy file is uniquely associated with a particular host. However, for each host multiple policy files may be defined. In that case, each policy file will define a certain set of checks to be performed. This way, it is easy to perform a set of less tasking tests frequently, and more thorough tests occasionally.

In this section the main topics related to constructing policy files are covered. For a full discussion, the reader should consult the *STEALTH* user guide.

A policy file consists of three types of commands. It may also contain *comment*, improving readability for human readers. The three types of commands are: *define directives* (starting with the keyword **DEFINE**), *use directives* (starting with the keyword **USE**) and *commands*. These three types of commands are discussed in some detail next.

DEFINE directives

`DEFINE` directives are used to reduce typing. Basically, a `DEFINE` directive is an abbreviation for a longer piece of text, albeit that they were given *some* intelligence. The generic form of the `DEFINE` directive looks like this:

```
DEFINE symbol     that what is defined by ‘name’
```

The symbols that are defined by `DEFINE` directives may consist of letters, digits and the underscore character (_). There is no restriction on the characters that are used in the definition of the symbol. The definition is, however, trimmed of initial or trailing blanks.

Having defined `symbol` in a `DEFINE` directive,

- `symbol` may be used in `USE` directives and `commands` (see below).
- The text following `DEFINE symbol` is then inserted literally into the `USE` directive or `command`.
- To apply a symbol that's defined by a `DEFINE` directive, the form `${symbol}` must be used. E.g., `${EXECMD5}`.
- `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.

Examples:

```
DEFINE  SSH         /usr/bin/ssh root@client.domain -q
DEFINE  EXECMD5     -xdev -perm +111  -type f \
                            -exec /usr/bin/md5sum {} \;
```

USE directives

`USE` directives are used to provide *STEALTH* with arguments which may be conditional to certain installations. The following **USE** directives are supported:

- **USE BASE** `basedirectory`

  The BASE specification has no default. **BASE** defines the directory from where *STEALTH* will operate. As this directive has no default it *must* be specified. Example:

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

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

  The **DIFF** specification defines the location of the program that is used to compare the log file that's created during a *STEALTH* run to the log file created by the previous *STEALTH* run. The example shows its default:

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

- **USE EMAIL** `address`

  The **EMAIL** specification provides one or more (space delimited) email-addresses receiving the reports of the integrity scans of `client.domain`. No mail is sent when no changes were detected with respect to the previous activation of *STEALTH*. The example shows its default:

  ```
  USE EMAIL root
  ```

- **USE MAILER** `mailer`

  The **MAILER** specification defines the program that is used to send the mail to the **EMAIL**-address. The **MAILER** program is called as follows:

  ```
  MAILER MAILARGS EMAIL
  ```

  (`MAILARGS`: discussed below). The `MAILER` program must be able to read its information from its standard input stream. The example shows its default:

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

- **USE MAILARGS** `arguments`

  The **MAILARGS** specification defines the arguments to pass to the `MAILER` program. The example shows its default:

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

  Note that blanks may be used in the subject specification, they should, however, be surrounded by double or single quotes when significant (see the example).

- **USE REPORT** `reportfile`

  REPORT defines the name of the report file. 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 as follows:

  ```
  STEALTH (1.11) started at Mon Apr 07 21:57:26 2003
  ```

  Only when (in addition to the marker line) information was appended to the report file, the *additional* contents of the report file are mailed to the mail address specified in the **USE EMAIL** specification. The following example shows its default:

  ```
  USE REPORT report
  ```

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

  The **SH** specification defines the command shell used by the controller to execute commands on `control.domain` itself (i.e., *local* commands). The example shows its default:

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

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

  The SSH specification has no default. Assuming `client.domain` *trusts* `controller.domain` (which is, after all, what *STEALTH* is all about), this should not be a very strong assumption. Example:

  ```
  USE SSH rootsh@client.domain -q
  ```

  When `client.domain` and `control.domain` happen to be the same computer, the `SSH` specification is also required (although this will undo all benefits offered by *STEALTH*, of course). An example of an `SSH` specification to scan `control.domain` itself is:

  ```
  USE SSH /usr/bin/ssh root@localhost -q
  ```

Command specifications

Following the **USE** specifications, *commands* are 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.

LABEL commands

- **LABEL** `text`

  This defines a text-label which is written to the **REPORT** file, just before the output generated by the next **CHECK**-command.

- **LABEL**

  A `LABEL` command without text clears a hitherto applied label.

LOCAL commands

Commands to execute on `control.domain` itself can be specified using **LOCAL** commands. All `LOCAL` commands are executed through the command shell specified by the `USE SH` specification. The following `LOCAL` commands are available:

- **LOCAL** `command`

  This command *must* succeed (i.e., must return a zero exit value), or *STEALTH* will terminate its run. Example:

```
LOCAL scp rootsh@client:/usr/bin/md5sum /tmp
```

This command will copy the `md5sum` program from `client.domain` to the `/tmp` directory at `control.domain`.

- **LOCAL NOTEST** `command`

  This command acts similarly as the previous command, but its return value is not interpreted. 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** `logfile command`

  If this command does not succeed (i.e., does not return a zero return value) the following *warning* message is written to the report file, but processing continues:

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

  This situation could occur, e.g., if an essential program (like `md5sum`) was transferred to `control.domain` to be integrity checked locally, and it was found to have been modified since the previous check. Processing continues, but any remaining checks should be interpreted with extreme caution. Example:

  ```
  LOCAL CHECK local/md5sum md5sum /tmp/md5sum
  ```

  This command will check the MD5 sum of the `/tmp/md5sum` program. The resulting output is saved at `${BASE}/local/md5sum`. The program must succeed (i.e., `md5sum` must return a zero exit-value).

Commands executed at client.domain

Plain commands can be executed on `client.domain` by merely specifying them. Such commands are executed on `client.domain` using the ssh connection specified by the **USE SSH** directive.

Commands run on `client.domain` can be specified as follows:

- `command`

  Here, `command` is executed on `client.domain`. The command must succeed (i.e., must return a zero exit value). Any output generated by the this command is ignored. Example:

```
/etc/init.d/inetd stop
```

This command will stop the `inetd` super-server.

- **NOTEST** `command`

  This command is executed on `client.domain`, without interpreting its return value. Example:

  ```
  NOTEST /etc/init.d/inetd stop
  ```

  Same as the previous command, but this time the exit value of the command is not interpreted.

- **CHECK** `logfile command`

  This command is executed on `client.domain`. Again, the command must return a zero return value. The output of this command, however, is compared to the output of this command generated during the previous run of *STEALTH*. Any differences are written to the report file.

  Please note that the command is *executed* on `client.domain`, while *the logfile is kept* on `control.domain`. This command represents the core of the philosophy implemented in *STEALTH*: once the command has been completed, there will be no residues on `client.domain`. Here are some examples of this type of command:

  ```
  CHECK 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 `client.domain` are produced, showing their permissions, owner and size information. The log file `${BASE}/remote/ls.root` will contain the resulting listing.

  Another example:

  ```
  DEFINE MD5SUM -xdev -perm +6111 -type f \
                          -exec /usr/bin/md5sum {} \;
  CHECK remote/md5.root /usr/bin/find / ${MD5SUM}
  ```

  The MD5 checksums of all suid/gid/executable files on the same device as the root-directory (/) on `client.domain` are determined. The resulting listing is written to the file `${BASE}/remote/md5.root`.

- **NOTEST CHECK** `logfile command`

  This command is executed on `client.domain`. The return value of this command is not interpreted. Apart from this, the specification is handled identically as the **CHECK** `logfile command` type of command, discussed before. Example:

  ```
  NOTEST CHECK remote/md5.root /usr/bin/find / ${MD5SUM}
  ```

  The MD5 checksums of all suid/gid/executable files on the same device as the root-directory (/) on `client.domain` are determined. The resulting listing is written on the file `${BASE}/remote/md5.root`. Using this command, *STEALTH* will not terminate if `/usr/bin/find` program returns a non-zero exit value.

References

Aide: Advanced intrusion detection environment.
(`http://www.cs.tut.fi/~rammer/aide-0.7.tar.gz`).

Fcheck: a script providing host intrusion detection and policy enforcement on servers.
(`http://sites.netscape.net/fcheck/`).

Filetraq: a script comparing a list of system files with copies kept.
(`http://filetraq.xidus.net/`).

Hontañón (2001): *Linux Security*, ISBN 0-7821-2741-X, Sybex, London.

Garfinkel and Spafford (1996): *Practical Unix & Internet Security*, ISBN 1-56592-148-8, O'Reilly, Cambridge.

OsirisScripts: scripts for generating a catalog of MD5 hashes and comparing the catalog to new snapshots.
(`http://www.shmoo.com/osiris/`).

Sans: The SANS (SysAdmin, Audit, Network, Security) Institute.
(`http://www.sans.org`).

Sentinel: a scanning utility similar to Tripwire.
(`http://zurk.netpedia.net/zfile.html`).

Sherpa: a script for configuring and then checking system security.
(`http://www.nbank.net/~rick/sherpa/`).

SSH: establishing a secure shell connection between computers.
(`http://www.openssh.com`).

STEALTH: SSH-based Trust Enhancement Acquired through a Locally Trusted Host.
(`ftp://ftp.rug.nl/contrib/frank/software/linux/stealth/`).

Tripwire: a policy driven file system integrity checking tool.
(`http://www.tripwire.com`).