

SAMHAIN (VERSION 1.1.12) USER MANUAL

Rainer Wichmann
<http://la-samhna.de>

May 20, 2001

Contents

1	Functional summary	1
1.1	Overview	1
1.2	Installation Requirements & Environment	2
1.3	How to invoke	3
1.4	Signals	3
1.5	Options & configuration file	4
1.6	Support (bug/problem reports)	4
2	Basic	5
2.1	Trusted users and trusted paths	5
2.2	Hash function	5
2.3	Logging – severities, classes, thresholds, and facilities	5
2.3.1	Severity levels	6
	Example	7
3	Configuring logging facilities	8
3.1	Thresholds – Activating logging facilities	8
	Example	9
3.2	Configuration	10
3.2.1	E-mail	10
	Complete example	11
3.2.2	Log file	11
3.2.3	Log server	12
3.2.4	External facilities	12
3.3	Details of logging facilities	12
3.3.1	Console	12
3.3.2	Syslog	12
3.3.3	E-mail	13
3.3.4	The log file	14
3.3.5	The log server	15

4	samhain – The file monitor	16
4.1	Basic usage instructions	16
4.2	File signatures	17
4.3	Defining which files/directories to monitor	17
4.3.1	Monitoring policies	17
4.3.2	File/directory specification	18
4.3.3	'All except ...'	18
4.3.4	Non-existent/disappeared/new files	19
4.3.5	Recursion depth(s)	19
4.4	Timing file checks	20
4.5	Initializing, updating, or checking	20
4.6	The database	20
4.7	Detecting Loadable Kernel Module (LKM) rootkits	20
	What is a LKM rootkit ?	20
	How can samhain detect them ?	21
	Configuration	21
4.8	Monitoring login/logout events	21
4.9	Modules	22
5	yule – The log server	23
5.1	General	23
5.2	Client registry	24
5.3	Server status information	25
5.4	Authentication protocol	26
5.4.1	Challenge-response	26
5.4.2	SRP	26
5.5	Message transfer protocol	27
5.6	File transfer protocol	27
6	External Programs	30
	Example	31
7	Signed Configuration/Database File	32

8	Stealth mode	34
8.1	Hiding the executable	34
8.2	Packing the executable	35
9	Deployment to remote host	37
9.1	Usage Notes	38
10	Security Design	40
A	Compilation options	41
A.1	General	41
A.2	OpenPGP Signatures on Configuration/Database Files	42
A.3	Client/Server Connectivity	42
A.4	Paths	43
B	Command line options	43
B.1	General	43
B.2	samhain	44
B.3	yule	44
C	The configuration file	45
C.1	General	45
	Example	45
C.1.1	Conditionals	45
	Example	46
C.2	Files to check	46
C.3	Severity of events	47
C.4	Logging thresholds	47
C.5	Watching login/logout events	47
C.6	Checking for kernel module rootkits	48
C.7	Miscellaneous	48
C.8	External	49
C.9	Clients	50
C.10	End of file	50

Abstract

samhain is a data integrity / intrusion alert system that can be used on single hosts as well as for large, UNIX-based networks.

samhain offers several features to support and facilitate centralized monitoring: **samhain** can be used as a client/server system, with monitoring clients on individual hosts and a central log server. Powerful conditionals allow to build a single configuration file for all clients on the network. Clients may download the configuration file and the database of file signatures from the log server.

This manual gives a detailed description of the **samhain** system. It is intended to be of help for anyone wishing to use, test, or modify **samhain** .

1 Functional summary

samhain is a system to monitor the integrity of files. It has a number of features that are intended to support and facilitate centralized monitoring in a network, although it can also be used on single hosts.

In particular, **samhain** can optionally be used as a client/server system with monitoring clients on individual hosts, and a central log server that collects the messages of all clients.

The configuration and database files for each client can be stored centrally and downloaded by clients from the log server. Using conditionals (based on hostname, machine type, OS, and OS release, all with regular expressions) a single configuration file for all hosts on the network can be constructed.

The client (or standalone) part is called **samhain** , while the server is referred to as **yule** . Both can run as daemon processes.

1.1 Overview

NOTE: This overview assumes that the database is already initialized (see Sect. 4.1). On startup, **samhain** /**yule** will

1. If **samhain** is used as SUID application (note that SUID usage is neither necessary nor recommended): set the effective user to some compiled-in default (e.g. **nobody**).
2. Parse the command line. Options given on the command line will override those in the configuration file.
3. Check whether the path to the configuration file is *trusted* (see Sect. 2.1), determine the checksum – or verify the signature – of the configuration file, then read in from it:
 - A list of files and directories to monitor, together with the specification of the policies that should be applied, i.e. what kind of modifications will be allowed or not. Wildcard patterns are supported.
 - Instructions regarding the logging facilities to be used.
 - Settings for the monitoring of login/logout events.
 - Miscellaneous other settings, as described in the appendix.
4. Obtain the local hostname, and information on the real and effective user. Initialize according to the specified options (e.g. disconnect from the parent process to become a daemon).
5. (**samhain** only): Determine the checksum – or verify the signature – of the file database.

6. Issue a startup message including user, time, and information on checksums – or signature keys – of configuration file and database.
7. **samhain** : Enter a loop to check the files specified in the configuration file against the database at regular intervals as defined in the configuration file.
yule : Enter a loop to wait for connections from clients.
8. **samhain** : If not running as daemon, exit after the first loop, else, exit on SIGTERM or SIGQUIT (see Sect. 1.4).
yule : Exit on SIGTERM or SIGQUIT (see Sect. 1.4).
9. Issue an exit message including time and reason for exit.

1.2 Installation Requirements & Environment

samhain requires an ANSI C compiler and a POSIX operating system. The installation procedure uses GNU autoconfigure (all configuration options are listed in the appendix):

```
./configure [options]
make
make          install
– or –
make          install-boot
```

Executables will be stripped upon installation. On Linux, the **strip** utility (copyright 1999 by Brian Raiter, under the GNU GPL) will be used to strip the executable even more, to prevent debugging with the GNU **gdb** debugger.

The following files will be installed (the last three files listed are optional, and only compiled and installed if the **--enable-network** option (**yule**, **samhain_setpwd**) or the **--with-stealth** option (**samhain_stealth**) has been selected):

Original	Installed to	Purpose	Mode
samhain.8	$\$(mandir)/man8/samhain.8$	manpage	600
samhainrc.5	$\$(mandir)/man5/samhainrc.5$	manpage	600
samhainrc	$\$(configdir)/.samhainrc$	configuration	600
samhain	$\$(bindir)/samhain$	executable	700
The log server:			
(yule)	$\$(bindir)/yule$	executable	700
Helper app (network):			
(samhain_setpwd)	$\$(bindir)/samhain_setpwd$	executable	700
Helper app (stealth):			
(samhain_stealth)	$\$(bindir)/samhain_stealth$	executable	700

The configuration file should be carefully checked before installation, especially with respect to the (e-mail, log server, time server) addresses listed therein.

Installed files should be owned by **root**. The path to the configuration file must be writeable by *trusted users* only (see Sect. 2.1).

*If the **--with-stealth** option is used, it is recommended to also use the option **--with-install-name** in order to rename all installed files, as well as files created by **samhain** , to some less suspicious name upon installation.*

1.3 How to invoke

From the command line:

```
samhain -t init [more options] to initialize the database
samhain -t check [more options] to check against the database
```

By default, **samhain** will *not* become a daemon, but stay in the foreground. Daemon mode must be set in the configuration file or on the command line.

Also by default, **samhain** will *neither* initialize its file system database *nor* check the file system against it. The desired mode must be set in the configuration file or on the command line.

A complete list of command line options is given in the appendix.

To start as daemon during the boot sequence:

For Linux, **make** will generate boot scripts for SuSE, RedHat, and Debian, and **make install-boot** will figure out which of them to install, and where (if the correct distribution cannot be determined, none of them will be installed).

For any other system, you need to figure out by yourself how to start **samhain** during the boot sequence.

1.4 Signals

On startup, all signals will be reset to their default. Then a signal handler will be installed for all signals that (i) can be trapped by a process and (ii) whose default action would be to stop, abort, or terminate the process, to allow for graceful termination,

For SIGSEGV, SIGILL, SIGBUS, and SIGFPE, a 'fast' termination will occur, with only minimal cleanup that may result in a stale lock file being left.

If the operating system supports the *siginfo_t* parameter for the signal handling routine (see **man sigaction**), the origin of the signal will be checked.

The following signals can be sent to the process to control it:

- **SIGUSR1** Switch on maximally verbose output to the console.

- **SIGUSR2** Return to previous console output mode.
- **SIGTERM** Terminate the process.
- **SIGQUIT** Terminate the server process after processing all currently pending requests from clients. Terminate the client process after finishing the current task (from the terminal, **SIGQUIT** usually is CTRL-backslash).
- **SIGHUP** Re-read the configuration file. Note that it is not possible to override command-line options given at startup.
- **SIGABRT** Unlock the log file, wait three seconds, then proceed. At the next access, the log file will be locked again and a fresh audit trail – with a fresh signature key – will be started. This allows log rotation without splitting an audit trail. See Sect. 3.3.4.

1.5 Options & configuration file

All command line options, and all settings in the configuration file, are described in the appendix.

1.6 Support (bug/problem reports)

If you have problems getting **samhain** to run, or think that you have encountered a bug, you can visit the user forum at <http://la-samhna.de/forum> and ask there for help (recommended for questions of probably general interest), or send email to support@la-samhna.de.

Please be sure to provide relevant details, such as:

- your operating system, its release version, and the machine (**uname -srm**).
- the version of **samhain** that you are using, and the options that you have supplied to **configure**.
- in case of problems it is usually *very helpful* if you compile **samhain** with the **configure** option **--enable-debug**, and run it with the command line switches **-p debug -z 1**.

It is *even more helpful* if you have in the configuration file the line **MessageHeader="(in the [Misc] section)**, which provides information on the origin of messages (source file/line). See Sect. C.1 for details on the configuration file.

2 Basic

2.1 Trusted users and trusted paths

Trusted users are `root` and the *effective user* of the process (usually, the effective user will be root herself). Additional trusted users can be defined in the configuration file (see Sect. 3.2.2 for an example), or at compile time (see appendix for compile options).

A *trusted path* is a path with all elements writeable only by trusted users. `samhain` requires the paths to the configuration and log file to be trusted paths, as well as the path to the lock file that will be created to lock access to the log file.

Evidently, if the path to the configuration file itself is writeable by other users than `root` and the *effective user*, these *must* be defined as trusted already at compile time. This is especially the case on some systems where the root directory is owned by the user `bin`.

If a path element is group writeable, all group members must be trusted.

Please note: The list of group members in `/etc/group` may be incomplete or even empty. `samhain` will check `/etc/passwd` (where each user has a GID field) in addition to `/etc/group` to find *all* members of a group.

2.2 Hash function

A *hash function* is a one-way function $H(foo)$ such that it is easy to compute $H(foo)$ from foo , yet infeasible to compute foo from $H(foo)$.

One common usage of a hash function is the computation of *checksums* of files, such that any modification of a file can be noticed, as its checksum will change.

For computing checksums of files, and also for some other purposes, `samhain` uses the TIGER hash function developed by Ross Anderson and Eli Biham. The output of this function is 192 bits long, and the function can be implemented efficiently on 32-bit and 64-bit machines. Technical details can be found at

<http://www.cs.technion.ac.il/~biham/Reports/Tiger/>.

2.3 Logging – severities, classes, thresholds, and facilities

Events (e.g. unauthorized modifications of files monitored by `samhain`) will generate *messages* of some *severity*. These messages will be logged to all logging facilities, whose *threshold* is equal to, or lower than, the severity of the message.

Events of related type are grouped into *classes*. For each logging facility, it is possible to restrict logging to a subset of these classes (see Sect. 3.1). The available classes are:

AUD	System calls.
RUN	Normal run messages (e.g. startup, exit, ...)
STAMP	Timestamps and alike.
FIL	Messages related to file integrity checking.
TCP	Messages from the client/server subsystem.
PANIC	Fatal errors, leading to program termination.
ERR	Error messages (general).
ENET	Error messages (network).
EINPUT	Error messages (input, e.g. configuration file).

2.3.1 Severity levels

The following severity levels are defined:

none	Not logged.
debug	Debugging-level messages.
info	Informational message.
notice	Normal conditions.
warn	Warning conditions.
mark	Timestamps.
err	Error conditions.
crit	Critical conditions, including program startup/normal exit.
alert	Fatal error, causing abnormal program termination.
inet	Incoming messages from clients (server only).

Most events (e.g. timestamps, internal errors, program startup/exit) have fixed severities. The following events have configurable severities:

- policy violations (for monitored files)
- access errors for files
- access errors for directories
- obscure file names (with non-printable characters)
- login/logout events (if **samhain** is configured to monitor them)

Severity levels for events (see Sect. 2.3.1) are set in the **EventSeverity** and (for login/logout events) the **Utmp** sections of the configuration file.

Example In the configuration file, these can be set as follows:

```
[EventSeverity]
#
# these are policies (see section 4.3.1)
#
SeverityReadOnly=crit
SeverityLogFiles=crit
SeverityGrowingLogs=warn
SeverityIgnoreNone=crit
SeverityIgnoreAll=info
#
# these are access errors
#
SeverityFiles=err
SeverityDirs=err
#
# these are obscure file names
#
SeverityNames=info
#
# This is the section for login/logout monitoring
#
[Utmp]
SeverityLogin=notice
SeverityLogout=notice
# multiple logins by same user
SeverityLoginMulti=err
```

3 Configuring logging facilities

`samhain` supports the following facilities for logging:

e-mail	<code>samhain</code> uses built-in SMTP code, rather than an external mailer program. E-mails are signed to prevent forging.
syslog	The system logging utility.
console	If running as daemon, <code>/dev/console</code> is used, otherwise <code>stderr</code> . <code>/dev/console</code> can be replaced by other devices as a compile option.
log file	Entries are signed to provide tamper-resistance.
log server	<code>samhain</code> uses TCP/IP with strong authentication and signed and encrypted messages.
external	<code>samhain</code> can be configured to invoke external programs for logging.

Each of these logging facilities has to be activated by setting an appropriate threshold on the messages to be logged by this facility.

In addition, some of these facilities require proper settings in the configuration file (see next sections).

3.1 Thresholds – Activating logging facilities

Messages are only logged to a log facility if their severity is at least as high as the threshold of that facility. Thresholds can be specified individually for each facility. A threshold of *'none'* switches off the respective facility.

Thresholds are set in the **Log** section of the configuration file. For each threshold option *FacilitySeverity* there is also a corresponding option *FacilityClass* to limit that facility to messages within a given set of class. The argument must be a list of valid message classes, separated by space or comma.

System calls: certain system calls (*execve*, *utime*, *unlink*, *dup (+ dup2)*, *chdir*, *open*, *kill*, *exit (+ _exit)*, *fork*, *setuid*, *setgid*, *pipe*) can be logged (only to console and syslog). You can determine the set of system calls to log via the option `LogCalls=call1, call2,`. By default, this is off (nothing is logged). The priority is `notice` (= `LOG_NOTICE` in syslog), and the class is `AUD`.

Example

```
[Log]
#
# Threshold for E-mails (none = switched off)
#
MailSeverity=none
#
# Threshold for log file
#
LogSeverity=err
LogClass=RUN FIL STAMP
#
# Threshold for console
#
PrintSeverity=info
#
# Threshold for syslog (none = switched off)
#
SyslogSeverity=none
#
# Threshold for forwarding to the log server
#
ExportSeverity=crit
#
# Threshold for invoking an external program
#
ExternalSeverity=crit
#
# System calls to log
#
LogCalls=open, kill
```

3.2 Configuration

Configuration options should be in the [Misc] section of the configuration file, except for *external* facilities.

3.2.1 E-mail

Items that must be configured are:

Recipients address in the format

`SetMailAddress=username@hostname`

Up to eight addresses are possible, each one at most 63 characters long, each on a separate line in the configuration file

Caveat: usually not all hosts in a domain are configured to receive e-mail, but rather there is often a dedicated mail exchanger. The host given in the e-mail address *must* be willing to handle e-mail, otherwise you need the *Mail relay / Mail exchanger* option (see below).

Hint: it is recommended to use *numerical* IP addresses instead of host names (to avoid DNS lookups).

Relay host / Mail exchanger in the format

`SetMailRelay=mail.some_domain.com`

There are two cases where you need this option:

(1) Some sites don't allow outbound e-mail connections from any arbitrary host. If the recipient is offsite, and your site uses a *mail relay host* to route outbound e-mails, you need to specify the relay host.

(2) Likewise, some hosts do not accept e-mails, in which case you have to use the proper *mail exchanger* as relay. You can get the name of the mail exchanger for host.some_domain.com with the command

`nslookup -type=mx host.some_domain.com`

Maximum interval in the format

`SetMailTime=86400`

You may want to set a maximum interval between any two consecutive e-mails, to be sure that `samhain` is still 'alive'.

Maximum pending in the format

`SetMailNum=10`

Messages can be queued to send several messages in one e-mail. You may want to set the the maximum number of messages to queue. (Note: messages of highest priority (*alert*) are always sent immediately.

Multiple recipients in the format

`MailSingle=yes/no`

If there are multiple recipients, whether to send a single mail with the recipient list, or send multiple mails. If all recipients are on same domain, a single mail may suffice, otherwise it depends on whether the mail server supports forwarding (for security, most don't).

Complete example

```
[Misc]
#
# E-mail receipt (offsite in this case). Up to eight addresses,
# each one at most 63 characters long.
#
SetMailAddress=username@host.some_domain.com
#
# Need a relay host for outgoing mail.
#
SetMailRelay=relay.mydomain
#
# Number of pending mails.
#
SetMailNum=10
#
# Maximum time between e-mails.
# Want a message every day, just to be sure that the
# program still runs.
#
SetMailTime=86400
#
# To all recipients in a single mail.
#
MailSingle=yes/no
```

3.2.2 Log file

Trusted users in the format

`TrustedUser=username`

If some element in the path to the log file is writeable by someone else than `root` or the *effective user* of the process, you have to include that user in the list of *trusted users* (unless their UIDs are already compiled in).

3.2.3 Log server

Server address in the format

`SetLogServer=my.server.address`

You have to specify the server address, unless it is already compiled in. It is possible to specify a second server that will be used as backup.

Hint: if you want to store the configuration file on the server, the server address *must* be compiled in.

3.2.4 External facilities

`samhain` can invoke external scripts/programs for logging (i.e. to implement support for pagers etc.). This is explained in detail in Sect. 6.

3.3 Details of logging facilities

This section discusses some details of the logging facilities offered by `samhain`. Configuring logging facilities (if required) is explained above. Activating logging facilities (by setting an appropriate threshold) is explained in section 3.1 .

3.3.1 Console

If running as daemon, `samhain` will use `/dev/console` for output, otherwise `stdout`. On Linux, `_PATH_CONSOLE` will be used instead of `/dev/console`, if it is defined in the file `paths.h`. You can override this as a compile option. You can also define a second console device. Console devices may be named pipes.

3.3.2 Syslog

`samhain` will translate its own severities into *syslog priorities* as follows:

debug	LOG_DEBUG
info	LOG_INFO
notice	LOG_NOTICE

warn	LOG_WARNING
mark	LOG_ERR
err	LOG_ERR
crit	LOG_CRIT
alert	LOG_ALERT

Messages will be truncated to 1023 chars. By default, **samhain** will use the *identity* 'samhain', the *syslog facility* LOG_AUTHPRIV, and will log its PID (process identification number) in addition to the message.

The syslog facility can be modified via the directive **SyslogFacility=LOG_***xxx* in the *[Misc]* section of the configuration file.

3.3.3 E-mail

The subject line contains timestamp and local hostname, which are repeated in the message body. **samhain** uses its own built-in SMTP code rather than the system mailer, because in case of temporary connection failures, the system mailer (e.g. **sendmail**) would queue the message on disk, where it may become visible to unauthorized persons.

During temporary connection failures, messages are stored in memory. The maximum number of stored messages is 128. **samhain** will re-try to mail every hour for at most 48 hours. In conformance with RFC 821, **samhain** will keep the responsibility for the message delivery until the recipient's mail server has confirmed receipt of the e-mail (except that, as noted above, after 48 hours it will assume a permanent connection failure).

The body of the mail may consist of several messages that were pending on the internal queue (see Sect. 3), followed by a signature that is computed from the message and a key. The key is initialized with a random number, and for each e-mail iterated by a *hash chain*.

The initial key is revealed in the first email sent (obviously, you have to believe that this first e-mail is authentic). This initial key is not transmitted in cleartext, but encrypted with a one-time pad. The one-time pad is generated by hashing a *base* (a compiled-in key) with a *salt* (the message itself). This way, different one-time pads can be generated from the same base.

The signature is followed by a unique identification string. This is used to identify separate audit trails (here, a *trail* is a sequence of e-mails from the same run of **samhain**), and to enumerate individual e-mails within a trail.

The mail thus looks like:

```
<--- MESSAGE ---->
first message
second message
...
<--- SIGNATURE ---->
signature
ID TRAIL_ID:hostname
<--- END ---->
```

To verify the integrity of an e-mail audit trail, a convenience function is provided:

```
samhain -M path_to_mailbox_file
```

The mailbox file may contain multiple and/or overlapping audit trails from different runs of **samhain** and/or different clients (hosts).

3.3.4 The log file

The log file is named **.samhain_log** by default, and placed into **/usr/local/var/log** by default (name and location can be configured at compile time).

The log file is created if it does not exist, and locked by creating a *lock file*. By default, the lock file is named **.samhain_lock** and placed in **/usr/local/var/log** (name and location can be configured at compile time). The lock file contains the PID of the process that created it. Upon normal program termination, the lock file is removed. Stale lock files are removed at startup if there is no process with that PID.

The directory where the log and its lock file are located must be writeable only by trusted users (see Sect. 2.1). This requirement refers to the *complete* path, i.e. all directories therein. By default, only **root** and the *effective user* of the process are trusted.

Audit trails (sequences of messages from individual runs of **samhain**) in the log file start with a **[SOF]** marker. Each message is followed by a signature, that is formed by hashing the message with a key.

The first key is generated at random, and sent by e-mail, encrypted with a one-time pad as described in the previous section on e-mail. Further keys are generated by a hash chain (i.e. the key is hashed to generate the next key). Thus, only by knowing the initial key the integrity of the log file can be assured.

The mail with the key looks like:

```
-----BEGIN MESSAGE-----  
message  
-----BEGIN LOGKEY-----  
Key(48 chars)[timestamp]  
-----BEGIN SIGNATURE-----  
signature  
ID TRAIL_ID:hostname  
<--- END --->
```

To verify the log file's integrity, a convenience function is provided:

```
samhain -L path_to_log_file
```

When encountering the start of an audit trail, you will then be asked for the key (as sent to you by e-mail). You can then:

- (i) hit **return** to skip signature verification,
- (ii) enter the key (without the appended timestamp), or
- (iii) enter the path to a file that contains the key (e.g. the mail box).

If you use option (iii), the path must be an absolute path (starting with a '/', not longer than 48 chars. For each audit trail, the file must contain a two-line block with the -----BEGIN LOGKEY----- line followed by the line (*Key(48 chars)[timestamp]*) from the mail. Additional lines before/after any such two-line block are ignored.

3.3.5 The log server

Details of the transmission protocols can be found in section 5. Configuring **samhain** for logging to the log server is explained in section 3 (setting the IP address of the server) and section 3.1 (activating the facility by setting an appropriate threshold).

During temporary connection failures, messages are stored in a FIFO queue in memory. The maximum number of stored messages is 128. After a connection failure, **samhain** will not try to connect for the next 10 minutes. A re-connection attempt is actually only made for the next message after that 10 minute deadtime – you should send timestamps (i.e. set the threshold to **mark**) to ensure re-connection attempts for failed connections.

It is possible to specify two log servers in the client configuration file. The first one will be used by default (primary), and the second one as fallback in case of a connection failure with the primary log server.

4 samhain – The file monitor

The **samhain** monitor checks the integrity of files by comparing them against a database of file signatures, and notify the user of inconsistencies. The level of logging is configurable, and several logging facilities are provided.

samhain can be used as a client that forwards messages to the server part (**yule**) of the **samhain** system, or as a standalone program (for single hosts).

samhain monitor can be run as a background process (i.e. a daemon), or it can be started at regular intervals by *cron*. It is recommended to run **samhain** as daemon and start it up immediately at system boot. Using it with *cron* opens up a security hole, because in that case the **samhain** program might be modified or replaced by a rogue program between two consecutive invocations.

4.1 Basic usage instructions

To use **samhain**, the following steps must be followed:

1. The configuration file must be prepared (see Sect. 4.3, 2.3, and 4.8 for details).
 - All *files and directories* that you want to monitor must be listed. Wildcard patterns are supported.
 - The *policies* for monitoring them (i.e. which modifications are allowed and which not) must be chosen.
 - The *severity* of a policy violation must be selected.
 - The *threshold level* of logging must be defined.
 - The *logging facilities* must be chosen.
 - Eventually, the *address* of the e-mail recipient and/or the *IP address* of the log server must be given.
2. The database must be initialized.
 - If it already exists, it should be deleted (**samhain** will not overwrite, but append), or *update* instead of *init* should be used.
 - **samhain** must be run with the command line option
`samhain -t init`
3. Now start **samhain** in *check* mode. Either select this mode in the configuration file, or use the command line option
`samhain -t check [more options]`
To run **samhain** as a background process, use the command line option
`samhain -t check -D [more options]`

4.2 File signatures

samhain works by generating a database of *file signatures*, and later comparing file against that database to recognize file modifications and/or added/deleted files.

File signatures include:

- a 192-bit *cryptographic checksum* computed using the TIGER hash algorithm,
- the inode of the file,
- the type of the file,
- owner and group,
- access permissions,
- on Linux only: flags of the ext2 file system (see `man chattr`),
- the timestamps of the file,
- the file size,
- the number of hard links,
- and the name of the linked file (if the file is a symbolic link).

Depending on the policy chosen for a particular file, only a subset of these may be checked for modifications (see sect. 4.3.1).

4.3 Defining which files/directories to monitor

This section explains how to specify in the configuration file, which files or directories should be monitored, and which monitoring policy should be used.

4.3.1 Monitoring policies

samhain offers several pre-defined monitoring policies. Each of these policies has its own section in the configuration file. Placing a file in one of these sections will select the respective policy for that file.

The available policies (section headings) are:

ReadOnly All modifications except access times will be reported for these files.

LogFiles Modifications of timestamps, file size, and signature will be ignored.

GrowingLogFiles Modifications of timestamps, and signature will be ignored. Modification of the file size will only be ignored if the file size has *increased*.

Attributes Only modifications of ownership and access permissions will be checked.

IgnoreAll No modifications will be reported. However, the *existence* of that file/directory will still be checked.

IgnoreNone All modifications, *including access time*, will be reported.

4.3.2 File/directory specification

Entries for files have the following syntax:

`file=/full/path/to/the/file`

Entries for directories have the following syntax:

`dir=[recursion depth]/full/path/to/the/directory`

The specification of a recursion depth is optional (see 4.3.5). (Note: Do not put the recursion depth in brackets – they just indicate that this is an optional argument ...).

Wildcard patterns ('*', '?', '[...]') as in shell globbing are supported for paths. The leading '/' is mandatory.

4.3.3 'All except ...'

To exclude individual files from a directory, place them under the policy **IgnoreAll**. Note that the *existence* of such files will still be checked (see next section).

To exclude subdirectories from a directory, place them under the policy **IgnoreAll** with an individual recursion depth of *-1* (see Sect. 4.3.5).

Note that any change in a directory will also modify the directory itself (i.e. the special file that holds the directory information). If you want to check all but a few files in a directory (say, */etc*), and you expect some of the excluded files to get modified, you should use a setup like:

```
[ReadOnly]
#
dir=/etc
#
[Attributes]
#
# less restrictive policy for the directory file itself
#
```

```

file=/etc
#
[IgnoreAll]
#
# exclude this file
#
file=/etc/resolv.conf.save
#

```

4.3.4 Non-existent/disappeared/new files

If files specified in the configuration file are non-existent already when the database is initialized, you will get an error message (for file access) only at initialization, while later, on file checking, only a message of severity *info* is generated.

If files disappear after initialization, you will get an error message with the severity specified for file access errors (*except* if the file is placed under the **IgnoreAll** policy, in which case a message of **SeverityIgnoreAll** – see Sect. 2.3.1 – is generated).

If new files appear in a monitored directory after initialization, you will get an error message with the severity specified for that directory's file policy (*except* if the file is placed under the **IgnoreAll** policy, in which case a message of **SeverityIgnoreAll** – see Sect. 2.3.1 – is generated).

The special treatment of files under the **IgnoreAll** policy allows to handle cases where a file might be deleted and/or recreated by the system sometimes.

4.3.5 Recursion depth(s)

Directories can be monitored up to a maximum recursion depth of 99 (i.e. 99 levels of subdirectories). The recursion depth actually used is defined in the following order of priority:

1. The recursion depth specified for that individual directory (see 4.3). As a special case, for directories with the policy **IgnoreAll**, the recursion depth should be set to 0, if you want to monitor (the existence of) the files within that directory, but to -1, if you do not want **samhain** to look *into* that directory.
2. The global default recursion depth specified in the configuration file. This is done in the configuration file section **Misc** with the entry **SetRecursionLevel=number**
3. The default recursion depth, which is zero.

4.4 Timing file checks

In the **Misc** section of the configuration file, you can set the interval (in seconds) between successive file checks:

`SetFilecheckTime=value`

4.5 Initializing, updating, or checking

In the **Misc** section of the configuration file, you can choose between initializing the database, updating it, or checking the files against the existing database:

`ChecksumTest=init—update—check—none`

If you use the mode *none*, you should specify on the command line one of *init*, *update*, or *check*, like: `samhain -t check`

4.6 The database

The database file is named `.samhain_file` by default, and placed into `/usr/local/var/log` by default (name and location can be configured at compile time).

The database is a binary file. For security reasons, it is recommended to store a backup copy of the database on read-only media, otherwise you will not be able to recognize file modifications after its deletion (by accident or by some malicious person).

`samhain` will compute the checksum of the database at startup and verify it at each access. `samhain` will first `open()` the database, compute the checksum, rewind the file, and then read it. Thus it is not possible to modify the file between checksumming and reading.

4.7 Detecting Loadable Kernel Module (LKM) rootkits

This option is currently supported *only* for Linux, kernel versions 2.2.x and 2.4.x, on i86 machines.

What is a LKM rootkit ? A *rootkit* is a set of programs installed to "keep a backdoor open" after an intruder has obtained root access to a system. Usually such rootkits are very easy to install, and provide facilities to hide the intrusion (e.g. erase all traces from audit logs, install a modified 'ps' that will not list certain programs, etc.).

While "normal" rootkits can be detected with checksums on programs, like `samhain` does (the modified 'ps' would have a different checksum than the original one), this method can be subverted by rootkits that are implemented as *loadable kernel modules*, i.e. modules that are loaded into the kernel at runtime.

A LKM can modify any kernel syscall to yield false results. Thus, it is possible to modify e.g. the `sys_getdents` call that reads directories in order to *hide* any file whose name comprises a certain "magic" string.

How can samhain detect them ? It is possible to compile into the `samhain` executable a map of all kernel syscall addresses. `samhain` will then check periodically, if any of these addresses has changed, thus indicating that the corresponding syscall has been clobbered by some other code. Note that if you use the option `--enable-khide` to use a kernel module to hide the presence of `samhain`, the `sys_getdents` syscalls will cause only a warning if modified only once (i.e. by the `samhain_hide` LKM).

To use this facility, you need to use the `configure` option:

```
--with-kcheck="/path/to/System.map"
```

`System.map` is a file (sometimes with the kernel version appended to its name) that is generated when the kernel is compiled, and is usually installed in the same directory as your kernel (e.g. `/boot`), or in the root directory. To find it, you can use:

```
locate System.map
```

Configuration This facility is configured in the `[Kernel]` section of the configuration file.

```
[Kernel]
# activate (0 for switching off)
KernelCheckActive=1
# interval between checks (in seconds, default 300)
KernelCheckInterval=600
# this is the severity (see section 2.3.1)
SeverityKernel=crit
```

4.8 Monitoring login/logout events

`samhain` can be compiled to monitor login/logout events of system users. For initialization, the system `utmp` file is searched for users currently logged in. To recognize changes (i.e. logouts or logins), the system `wtmp` file is then used.

This facility is configured in the `[Utmp]` section of the configuration file:

```
[Utmp]
#
# activate (0 for switching off)
#
```

```
LoginCheckActive=1
#
# interval between checks (in seconds)
#
LoginCheckInterval=600
#
# these are the severities (see section 2.3.1)
#
SeverityLogin=info
SeverityLogout=info
#
# multiple logins by same user
#
SeverityLoginMulti=crit
```

4.9 Modules

samhain has a programming interface that allows to add modules written in C. Basically, for each module a structure of type `struct mod_type`, as defined in `sh_modules.h`, must be added to the list in `sh_modules.c`.

This structure contains pointers to initialization, timing, checking, and cleanup functions, as well as information for parsing the configuration file.

For details, in the source code distribution check the files `sh_modules.h`, `sh_modules.c`, as well as `utmp.c`, `utmp.h`, which implement a module to monitor login/logout events.

5 yule – The log server

yule is the log server within the **samhain** file integrity monitoring system. **yule** is part of the distribution package. It is only required if you intend to use the client/server capability of the **samhain** system for centralized logging to **yule** .

5.1 General

yule is a non-forking server. Instead of forking a new process for each incoming logging request, it multiplexes connections internally.

Each potential client must be **registered** with **yule** to make a connection (see Sect. 4.1 and the example below). The client tells its host name to the server, and the server verifies it against the peer of the connecting socket. On the first connection made by a client, an authentication protocol is performed. This protocol provides *mutual authentication* of client and server, as well as a *fresh session key*.

By default, all messages are encrypted using `|i|Rijndael|i|` (selected as the Advanced Encryption Standard algorithms). The 192-bit key version of the algorithm is used. There is a compile-time option to switch off encryption, if your local lawmakers don't allow to use it (see Appendix).

yule keeps track of all clients and their session keys. As connections are dropped after successful completion of message delivery, there is *no* limit on the total number of clients. There is, however, a limit on the maximum number of *simultaneous* connections. This limit depends on the operating system, but may be of order 10^3 .

Session key expire after two hours. If its session key is expired, the client is forced to repeat the authentication protocol to set up a fresh session key.

Incoming messages are signed by the client. On receipt, **yule** will:

1. check the signature,
2. accept the message if the signature can be verified, otherwise discard it and issue an error message,
3. discard the clients signature,
4. log the message, and the client's hostname, to the console and the log file, and
5. add its own signature to the log file entry.

It is possible to set a time limit for the maximum time between two consecutive messages of a client (option `SetClientTimeLimit` in the configuration file). If the time limit is exceeded without a message from the client, the server will issue a warning. The default is 86400 seconds (one day); specifying a value of 0 will switch off this option.

By default, client messages have the severity *inet*, and are logged **only** to the console and the log file (and to external, if threshold is properly set). It is possible to override this behavior by setting the option `UseClientSeverity=yes` in the configuration file. In that case, the client message severity is used, and client messages are treated just like local messages (i.e. like those from the server itself).

5.2 Client registry

As noted above, clients must be registered with **yule** to make a connection. The respective section in the configuration file looks like:

```
[Clients]
#
# A client
#
Client=HOSTNAME_CLIENT1@salt1@verifier1
#
# another one
#
Client=HOSTNAME_CLIENT2@salt2@verifier2
#
```

The entries have to be computed in the following way:

1. Choose a *password* (16 chars hexadecimal, i.e. only 0 – 9, a – f, A – F allowed. You may use:

```
yule -gen-password
```

2. Use the program `samhain_setpwd` to reset the password in the *compiled binary* (that is, `samhain`, not `yule`) to the one you have chosen. Running `samhain_setpwd` without arguments will print out exhaustive usage information.
3. Use the server's convenience function to create a registration entry:

```
yule -P password
```

4. The output will look like:

```
Client=HOSTNAME@salt@verifier
```

You now have to replace *HOSTNAME* with the fully qualified domain name of the host on which the client should run.

5. Put the registration entry into the servers's configuration file, under the section heading **Clients** (see Sect. 5.2). You need to send SIGHUP to the server for the new entry to take effect.
6. Repeat steps (a) – (e) for any number of clients you need (actually, you need a registration entry for each client's host, but you don't necessarily need different passwords for each client. I.e. you may skip steps (a) – (c)).

5.3 Server status information

yule writes the current status to a HTML file. The default name of this file is *.samhain.html*, and by default it is placed in */usr/local/var/log*.

The file contains a header with the current status of the server (starting time, current time, open connections, total connections since start), and a table that lists the status of all registered clients.

There are a number of pre-defined events that may occur for a client:

Inactive	The client has not connected since server startup.
Started	The client has started.
	This message may be missing if the client was already running at server startup.
Exited	The client has exited.
Message	The client has sent a message.
File transfer	The client has fetched a file from the server.
ILLEGAL	Startup without prior exit.
	May indicate a preceding abnormal termination.
PANIC	The client has encountered a fatal error condition.
FAILED	An unsuccessful attempt to set up a session key or transfer a message.
POLICY	The client has discovered a policy violation.

For each client, the latest event of each given type is listed. Events are sorted by time. Events that have not occurred (yet) are not listed.

It is possible to specify templates for (i) the file header, (ii) a single table entry, and (iii) the file end. Templates must be named *head.html*, *entry.html*, and *foot.html*, respectively, and must be located in the *\$dataroot* directory (see Sect. A.4). The distribution package includes two sample files *head.html* and *foot.html*.

The following replacements will be made in the *head* template:

%T	Current time.
%S	Startup time.
%L	Time of last connection.
%O	Open connections.
%A	Total connections since startup.
%M	Maximum simultaneous connections.

The following replacements will be made in the *entry* template:

%H	Host name.
%S	Event.
%T	Time of event.

NOTE: A literal '%' in the HTML output must be represented by a '% ' ('%' followed by space) in the template.

5.4 Authentication protocol

Depending in the option selected at compile time, either a challenge-response protocol or the *Secure Remote Password (SRP)* protocol will be used for mutual authentication and exchange of a session key.

5.4.1 Challenge-response

1. The client requests a random nonce from the server.
2. The server generates a random nonce v and sends $H(v:password)v$ to the client. (H is a one-way *hash* function.)
3. The client generates a random nonce u and sends $H(H(u:v)password)u$.
4. The session key is $H(v:password:u)$

5.4.2 SRP

The protocol is described in detail in the following paper (available at <http://srp.stanford.edu/srp>):

T. Wu, The Secure Remote Password Protocol, in Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, San Diego, CA, Mar 1998,

pp. 97-111.

Some of the advantages of SRP are:

1. No useful information about the password is revealed.
2. No useful information about the session key is revealed to an eavesdropper.
3. A compromise of a session key does not help to determine the password.
4. A compromise of the password does not allow to determine the session key for past sessions.
5. A man-in-the-middle may at worst cause the authentication to fail.

5.5 Message transfer protocol

To submit a message to `yule`, the following protocol is used:

1. The client request a random nonce from the server.
2. The server generates a random nonce u and sends it to the client.
3. The client send the message, followed by a signature. The signature is computed as $H(message:u:session\ key)$. (H is a one-way *hash* function.)
4. On receipt of the message, the server verifies the signature, and discards *message* on failure.
5. The server confirms successful receipt by sending $H(message:session\ key:u)$ (i.e. reverse order of u and *session key* in the hash).
6. The client verifies the server's confirmation.

Message transfer is *reliable* in the sense that the client assumes responsibility for the message until it has verified the server's confirmation of the receipt.

5.6 File transfer protocol

Caveat: Obviously, retrieving the configuration file from the log server requires that the IP address of the log server is *compiled in*.

If the compiled-in path to the configuration file begins the special value “REQ_FROM_SERVER”, the client will request to download the configuration file from `yule`. If “REQ_FROM_SERVER”

is followed by a path, the server will use that path as the path to its configuration file (basically, this feature allows to use the same configuration options for client and server). If the client is initializing the database (rather than checking), and “REQ_FROM_SERVER” is followed by a path, the client will use that path as the path to a local configuration file.

Likewise, if the compiled-in path to the database file begins with the special value “REQ_FROM_SERVER”, the client will request to download the database file from yule for reading. “REQ_FROM_SERVER” must be followed by a path that will be used for writing the database file when *initializing* (the client cannot *upload* the database file to the server, as this would open a security hole).

For file transmission, the following protocol is used:

1. The client announces that it requests a file from the server.
2. The server generates and sends a random nonce u .
3. The client generates and sends a random nonce v , together with a request for either the configuration or database file.
4. The server sends the file in chunks of 65280 bytes, each preceded by a checksum computed as $H(H(u:v:session\ key)H(data))$.
5. The client verifies the checksum, and discards $data$ on failure.
6. The server ends the file transmission with an EOF marker signed by $H(H(u:v:session\ key)H(client_hostname))$.
7. The client verifies the EOF marker, and discards the file on failure.

The server will search for the configuration file to send in the following order of priority (`$dataroot` is the data directory, see Sect. A.4; *clientname* is the hostname of the client’s host):

1. `$dataroot/rc.clientname`
2. `$dataroot/rc`
3. The server’s own configuration file

The server will search for the database file to send in the following order of priority:

1. `$dataroot/file.clientname`
2. `$dataroot/file`

On the client side, transferred data are written to a *temporary file* that is created in the home directory of the *effective* user. The filename is chosen at random, the file is opened for writing after checking that it does not exist already, and immediately thereafter *unlinked*.

Thus the *name* of the file will be deleted from the filesystem, but the file itself will remain in existence until the file descriptor referring it is closed (see `man unlink`), or the process exits (on exit, *all* open file descriptors belonging to the process are closed).

6 External Programs

samhain may invoke external programs or scripts in order to implement logging capabilities that are not supported by **samhain** itself (e.g. pager support). This section provides an overview of this capability.

External programs/scripts invoked for logging will receive the formatted log message on **stdin**. The program should expect that **stdout** and **stderr** are closed, and that the working directory is the root directory.

Each external program must be defined in the configuration file, in a section starting with the header **[External]**.

In addition, **ExternalSeverity** must be set to an appropriate threshold in the section **[Log]**.

Each program definition starts with the line

OpenCommand=*/full/path*

Options for the program may follow. The definition of an external program is ended when the section ends, or when another **OpenCommand=***/full/path* line for the next command is encountered.

- There are several places in **samhain** where external programs may be called. Each such place is identified by a **type**. Currently, valid types are:

log An external logging facility, which is handled like other logging facilities. The program will receive the logged message on **stdin**, followed by a newline, followed by the string **[EOF]** and another newline.

srv Executed by the server, whenever the status of a client, as displayed in the HTML status table, has changed. The program will receive the client host-name, the timestamp, and the new status, followed by a newline, followed by **[EOF]** and another newline.

- Any number of external programs may be defined in the configuration file. Each external program has a **type**, which is *log* by default. Whenever external programs are called, *all* programs of the appropriate **type** are executed. The **type** can be set with

SetType=*type*

- External programs must be on a trusted path (see Sect. 2.1), i.e. must not be writeable by untrusted users.
- For enhanced security, the (192-bit TIGER) checksum of the external program/script may be specified in the configuration file:

SetChecksum=*checksum (one string, no blanks in checksum)*

- Command line arguments and environment variables for each external program are configurable (the default is no command line arguments, and only the timezone in the environment):
`SetCommandline=full command line (starting with the name of the program)`
`Setenviron=KEY=value`
- The user whose credentials shall be used, can be specified:
`SetCredentials=username`
- Some filters are available to make the execution of an external program dependent on the message content:
`SetFilterNot=list` If any word in *list* matches a word in the message, the program is not executed, else
`SetFilterAnd=list` if any word in *list* is missing in the message, the program is not executed, else
`SetFilterOr=list` if none of the words in *list* is in the message, the program is not executed.
Any filter not defined is not evaluated.
- It is possible to set a 'deadtime'. Within that 'deadtime', the respective external program will be executed only once (if triggered).

Example

```
[External]
# start definition of first external program
OpenCommand=/usr/local/bin/warn_me
SetType=log
# arguments
SetCommandline=warn_me -v
# environment
SetEnviron=HOME=/home/moses
SetEnviron=PATH=/bin:/usr/bin:/usr/local/bin
# checksum
SetChecksum=4CA372D66F9C909B8A974E27A43EAC51D68F11FE0B30E08A
# credentials
SetCredentials=moses
# filter
SetFilterOr=POLICY
```

7 Signed Configuration/Database File

Both the configuration file (Sect. C.1) and the database of file signatures (Sect. 4.6) may be *cleartext* signed by GnuPG (**gpg**) or PGP (**pgp**). If compiled *without* support for signatures, **samhain** will *ignore* them (the signatures then may still be useful for manual verification.)

If compiled *with* support, **samhain** will invoke **gpg** or **pgp** to verify the signature. To compile with **gpg/pgp** support, use the **configure** option:

```
./configure --with-gpg=/path/to/gpg (GnuPG)
```

or

```
./configure --with-pgp=/path/to/pgp (PGP)
```

Note that you must compile in the absolute path to the **gpg/pgp** executable.

Before calling the program, **samhain** will check that the path to the executable is writeable **only by trusted users** (see Sect. 2.1).

The program will be called without using the shell, with its full path (as compiled in), and with an environment that is limited to the **\$HOME** variable, which is set to the home directory of the *effective* user (as determined from **/etc/passwd**).

Usually, the *effective* user would be **root**, and thus **\$HOME** would be **/root** or whatever **root**'s home directory is.

The **\$HOME** environment variable determines where **gpg/pgp** will look for the public key to verify the signatures (subdirectories **\$HOME/.gnupg/\$HOME/.pgp**).

As signatures on files are only useful as long as you can trust the **gpg/pgp** executable and the file holding the public key, you may consider using the following options:

- it is possible to compile in the TIGER checksum of the **gpg/pgp** executable, which then will be verified before calling the program. The appropriate **configure** option is:

```
--with-checksum="CHECKSUM"
```

CHECKSUM should be the checksum as printed by

```
gpg --load-extension tiger --print-md TIGER /path/to/gpg or  
samhain -H /path/to/gpg
```

(the full line of output, with spaces).

Example:

```
--with-checksum="/usr/bin/gpg: 1C739B6A F768C949 FABEF313 5F0B37F5 22ED4A27  
60D59664"
```

- it is possible to compile in the key fingerprint of the signature key, which then will be verified after checking the signature itself:

`--with-fp=FINGERPRINT`

FINGERPRINT should be the key fingerprint *without* spaces.

Example:

`--with-fp=EF6CEF54701A0AFDB86AF4C31AAD26C80F571F6C`

samhain will report the signature key owner and the key fingerprint as obtained from **gpg/pgp**. If both files are present and checked (i.e. when checking files against the database), both must be signed with the same key. If the verification is successful, **samhain** will only report the signature on the configuration file. If the verification fails, or the key for the configuration file is different from that of the database file, an error message will result.

8 Stealth mode

If an intruder does not know that `samhain` is running, s/he will make no attempt to subvert it. Hence, you may consider to run `samhain` in stealth mode, using some of the options discussed in this section.

8.1 Hiding the executable

`samhain` may be compiled with support for a stealth mode of operation, meaning that the program can be run without any obvious trace of its presence on disk. The following options are provided:

`--with-stealth=xor_val` provides the following measures:

1. All embedded strings are obfuscated by XORing them with some value `xor_val` chosen at compile time. The allowed range for `xor_val` is 128 to 255.
2. The messages in the log file are obfuscated by XORing them with `xor_val`. The built-in routine for validating the log file will handle this transparently.
3. Paths in the database file are obfuscated by XORing them with `xor_val`.
4. The configuration file must be steganographically hidden in a postscript image file (the image data must be *uncompressed*). To create such a file from an existing image, you may use e.g. the program `convert`, which is part of the `ImageMagick` package, such as:

```
convert +compress ima.jpg ima.ps.
```

To hide/extract the configuration data within/from the postscript file, a utility program `samhain_stealth` is provided. Use it without options to get help.

`--with-micro-stealth=xor_val` is like `--with-stealth`, but uses a 'normal' configuration file (not hidden steganographically).

`--with-nocl[=ARG]` will disable command line parsing. The optional argument is a 'magic' word that will enable *reading* command-line arguments *from stdin*. If the first command-line argument is not the 'magic' word, all command line arguments will be ignored. This allows to start the program with completely arbitrary command-line arguments.

`--with-install-name=NAME` will rename every installed file from `*samhain*` to `*NAME*`. Also, the boot scripts (`samhain.startSuSE`, `samhain.startDebian`, `samhain.startRedHat`)

will be updated accordingly. Files created by **samhain** (e.g. the database) will also have 'samhain' replaced by 'NAME' in their filenames.

Hint: the man pages have far too much specific information enabling an intruder to infer the presence of samhain. There is no point in changing 'samhain' to 'NAME' there - this would rather help an intruder to find out what 'NAME' is. You probably want to avoid installing man8/samhain.8 and man5/samhainrc.5.

--enable-khide (*Linux only*) will compile/install two loadable kernel modules (**samhain_hide.o/samhain_erase.o**). **samhain_hide.o** will hide every file/directory/process with the string NAME (from **--with-install-name**). If **--with-install-name** is not used, NAME is set to **samhain**.

To hide the module itself, the second module **samhain_erase.o** is provided. Loading and immediately thereafter unloading this module will hide any module with the string NAME in its name.

make install will install the kernel modules to the appropriate place.

Note: hidden files can still be accessed if their names are known, thus using the **--with-install-name** to rename installed files is recommended for security.

Note: using the modules at system boot may cause problems with the GNOME (1.2) **gdm** display manager (no problems observed with **kdm**). In case of problems, you may need to reboot into single-user mode and edit the boot init script ...

8.2 Packing the executable

For even more stealthiness, it is possible to pack and encrypt the **samhain** executable. The packer is just moderately effective, but portable. Note that the encryption key of course must be present in the packed executable, thus this is no secure encryption, but rather is intended for obfuscation of the executable. There is a make target for packing the **samhain** executable:

```
make samhain.pk
```

On execution, **samhain.pk** will unpack into a temporary file and execute this, passing along all command line arguments. The temporary file is created in **/tmp**, if the sticky bit is set on this directory, and in **/usr/bin** otherwise. The filename is chosen at random, and the file is only opened if it does not exist already (otherwise a new random filename will be tried). The file permission is set to 700.

The directory entry for the unpacked executable will be deleted after executing it, but on systems with a **/proc** filesystem, the deleted entry may show up there. In particular, this is the case for Linux. You should be aware that this may raise suspicion.

On Linux, the **/proc** filesystem is used to call the unpacked executable without a race condition, by executing **/proc/self/fd/NN**, where NN is the file descriptor to which the unpacked executable has been written. On other systems, the filename of the unpacked executable must be used, which creates a race condition (the file may be modified between

creation and execution).

The packed executable will not honour the SUID bit.

9 Deployment to remote host

samhain includes a system to facilitate deployment of the client to remote hosts. There are two major parts of this system:

- A library of *profiles* depending on the remote system *type* (the subdirectories **profiles/type/** in the source tree) that includes three files for each system type:
 1. **configopts** holds the build configure options, i.e. the options given to **configure** when building the **samhain** executable on the remote host,
 2. **samhainrc** holds the configuration file for the **samhain** executable, and
 3. **bootscript** is a script that modifies the remote host configuration to make **samhain** start when booting.
- A script **deploy.sh** (created by **configure** from **deploy.sh.in**) that, on execution, will:
 1. create a mini-distribution *samhain-deploy.tar.gz*,
 2. copy it to the remote host,
 3. compile (if needed) and install the **samhain** client,
 4. initialize and retrieve the database (and the compiled binary), delete the database on the remote host, and
 5. store the client's credentials in a file **yulerc**. If this file does not exist already, it is copied from **yulerc.template**.

The compiled client is retrieved and saved in the **profiles/type/** directory. For deployment to another host of the same type, this compiled client will be used, instead of recompiling it.

deploy.sh takes the following arguments (*order is relevant*):

[-v --verbose]	verbose output
[-f --force]	force recompilation, even if compiled binary available
[-p --pack]	pack the executable
<i>host</i>	remote hostname
<i>type</i>	system type of remote host (profiles/type/)
<i>[password]</i>	client/server password (autogenerated by default)

In addition, the following environment variables are recognized:

SH_PREFIX	Install directory prefix on the remote host (set by <code>configure</code>).
SH_NAME	The name of the executable (default=samhain).
SH_SRCDIR	The top source directory (default=.).
SH_BUILDDIR	The build directory on the remote host (default=/).
SH_REMOTE_USER	The remote host username for compiling (default=root).
SH_REMOTE_ROOT	The remote host superuser name (default=root).
SH_LOCALHOST	The local hostname (FQDN). Will use 'hostname', if available.
SH_NOCL_CODE	The 'magic' value to enable CL parsing (default=quark).
SH_XOR_CODE	The XOR value to obfuscate strings (set by <code>configure</code>).
SH_BASE1	The 'B1' in <code>-with-base=B1,B2</code> (set by <code>configure</code>).
SH_BASE2	The 'B2' in <code>-with-base=B1,B2</code> (set by <code>configure</code>).

9.1 Usage Notes

- You must run `configure` first, *and* compile the server (`yule`), before using `deploy.sh`.
- `yule` must be in your path, if `deploy.sh` is not used from the top source directory. It is not necessary to have the server running, though.
- `deploy.sh` uses `ssh/scp`. You need to have the `sshd` daemon running on the remote host. It is helpful if RSA-based authentication is possible for `root`, otherwise you have to type in your password quite a few times.
Note: if you use RSA-based authentication, it is recommended:
 - **not** to store an unencrypted private RSA key (in `.ssh/identity`) on a remote host that may be accessible to an intruder (*very dangerous* – the private RSA key can be used to login as root on other machines).
 - **not** to use `ssh`'s own scheme of encrypting the private key with a passphrase (*very inconvenient* – you would need to type in the passphrase for every `ssh/scp` command).
 - **but instead** to use GnuPG or PGP to encrypt the private RSA key (in `.ssh/identity`), and store it on a trusted machine or removeable media only. Only store the public RSA key (in `.ssh/authorized_keys`) on remote hosts. Only decrypt the private RSA key if you need to login to (a) remote host(s), and delete the *decrypted* key if not needed anymore.
- SH_XOR_CODE, SH_BASE1, SH_BASE2 are needed for consistency across multiple runs of `configure`. This is *not* important for client/server interaction, but for verification of e-mails/log files written by the client (if you make use of these additional logging facilities).
- The deployed client is compiled to retrieve the database and the configuration file from the server. It will not work (*except for initialization of the database*) with

database/configuration files stored on the client side. When invoked for file system checking, the deployed client will expect the server running on the host specified in the environment variable `SH_LOCALHOST`, which by default is set to the local host on which `deploy.sh` is executed (surprise, surprise ...).

- To add support for another system type, just create a subdirectory named `profiles/type/` in the source tree, and figure out appropriate files `configopts`, `samhainrc`, and `bootscript`.
- To add credentials of new clients at runtime to `yule` , copy the file `yulerc` (or the new client credentials therein) to the server's configuration file, and send a `SIGUSR1` signal to `yule` to reconfigure.

10 Security Design

Obviously, a security application should not open up security holes by itself. Therefore, an important aspect in the development of **samhain** has been the security of the program itself. While **samhain** comes with no warranty (see the license), much effort has been invested to identify security problems and avoid them.

To avoid buffer overflows, only secure string handling functions are used to limit the amount of data copied into a buffer to the size of the respective buffer (unless it is known in advance that the data will fit into the buffer).

On startup, the timezone is saved, and all environment variables are set to zero thereafter. Signal handlers, timers, and file creation mask are reset, and the core dump size is set to zero. If started as daemon, all file descriptors are closed, and the first three streams are opened to `/dev/null`.

If external programs are used (in the entropy gatherer, if `/dev/random` is not available), they are invoked directly (without using the shell), with the full path, and with a limited environment (by default only the timezone). Privileged credentials are dropped before calling the external program.

With respect to its own files (configuration, database, the log file, and its lock), on access **samhain** checks the complete path for write access by untrusted users. Some care has been taken to avoid race conditions on file access as far as possible.

samhain requires root privileges to monitor files with privileged access. If set **SUID root**, **samhain** will run with the credentials of a compiled-in user, which by default is **nobody**. In that case, root privileges will only be used if necessary.

Critical information, including session keys and data read from files for computing checksums, is kept in memory for which paging is disabled (if the operating system supports this). This way it is avoided that such information is transferred to a persistent swap store medium, where it might be accessible to unauthorized users.

Random numbers are generated from a pseudo-random number generator (PRNG) with a period of 2^{88} (actually by mixing the output from three instances of the PRNG). The internal state of the PRNG is seeded from a strong entropy source (if available, `/dev/random` is used, else lots of system statistics is pooled and mixed with a hash function). The PRNG is re-seeded from the entropy source at regular intervals (one hour).

Numbers generated from a PRNG can be predicted, if the internal state of the PRNG can be inferred. To avoid this, the internal state of the PRNG is hidden by hashing the output with a hash function.

A Compilation options

A.1 General

- enable-login-watch** Compile in the module to watch for login/logout events.
- with-identity=USER** The username to use when dropping root privileges (default `nobody`).
- with-sender=SENDER** The username of the sender for e-mail (default is `daemon`).
- with-recipient=ADDR** The recipient(s) for e-mail, separated by whitespace (max. 8). You can add recipients in the configuration file as well.
- with-trusted=UID** Trusted users (must be a comma-separated list of numerical UIDs). Only required if the configuration file must be on a path writeable by others than `root` and the *effective* user.
- with-timeserver=HOST** Set host address for time server (default is literal “NULL” - use own clock). You can set this in the configuration file as well. An address in the configuration file will take precedence.
- with-alttimeserver=HOST** Set host address for an alternative (backup) time server.
- with-kcheck=SYSTEM_MAP** (Linux only) Check for clobbered kernel syscalls (to detect kernel module rootkits). `SYSTEM_MAP` must be the path to the `System.map` file corresponding to the kernel.
- with-stealth=XOR_VAL** Enable stealth mode, and set `XOR_VAL`. `XOR_VAL` must be decimal, in the range 127 – 255, and will be used to obfuscate literal strings.
- with-micro-stealth=XOR_VAL** As `--with-stealth`, but without steganographic hidden configuration file.
- with-nocl=PW** Command line parsing is enabled only if the first command line argument is `PW`. `PW=""` (empty string) will disable command line parsing completely. This may be used as addition to `–with(-micro)-stealth` to prevent interactive enforcement of telltale output.
- with-install-name=NAME** Upon installation, rename every file from `*samhain*` to `*NAME*`. To be used in conjunction with `–with(-micro)stealth`.
- with-base=B1,B2** Set base key for one-time pads. Must be ONE string (no space) made of TWO comma-separated integers in the range 0 – 2147483647.
Caveat: If this option is *not* used, a random value will be chosen at compile time (by the configuration script). Binaries compiled with different values cannot verify the audit trail(s) of each other.

- enable-debug** Enable debugging. Will slow down things, increase resource usage, and *may* leak information that should be kept secure.
- enable-pttrace** Periodically check whether a debugger is attached, and abort if yes. Only takes effect if **--enable-debug** is not used.

A.2 OpenPGP Signatures on Configuration/Database Files

- with-gpg=PATH** Use GnuPG to verify database/configuration file. The public key of the *effective* user (in `/.gnupg/pubring.gpg`) will be used.
- with-pgp=PATH** Use PPG to verify database/configuration file. The public key of the *effective* user (in `/.pgp/pubring.pgp`) will be used.
- with-checksum=CHECKSUM** Compile in TIGER checksum of the `gpg/pgp` binary. CHECKSUM must be the full line output by `samhain` or `gpg` when computing the checksum (`pgp` has no support for the TIGER algorithm).
- with-fp=FINGERPRINT** Compile in the fingerprint of the key used to sign the configuration/database files. FINGERPRINT must be without spaces. If used, `samhain` will verify the fingerprint, but still report on the used public key.

A.3 Client/Server Connectivity

- enable-network** Compile with client/server support.
- disable-encrypt** Disable encryption for client/server communication.
- disable-srp** Disable the use of the zero-knowledge SRP protocol to authenticate to log server, and use a (faster, but less secure) challenge-response protocol.
- with-port=PORT** The port on which the server will listen (default is 49777). Only needed if this port is already used by some other application. Port numbers below 1024 require `root` privileges for the server.
- with-logserver=HOST** The host address of the log server. This can be set in the configuration file. A compiled-in address is only required if you want to fetch the configuration file from the log server. An address in the configuration file will take precedence.
- with-altlogserver=HOST** The host address of an alternative (backup) log server.

A.4 Paths

Compiled-in paths may be as long as 255 chars. If the `--with-stealth` option is used, the limit is 127 chars.

- `-prefix=PREFIX` The root install directory (default is */usr/local*).
- `-with-config-file=FILE` The full path of the configuration file (default is *\$PREFIX/etc/.samhainrc*).
- `-with-dataroot-prefix=PFX` The dataroot directory (default is *\$PREFIX/var/log*).
- `-with-log-file=FILE` The path of the log file (default is *\$PFX/.samhain.log*).
- `-with-lock-file=FILE` The path of the lock file (default is *\$PFX/.samhain.lock*).
- `-with-data-file=FILE` The path of the database file written by `samhain` (default is *\$PFX/.samhain_file*).
- `-with-html-file=FILE` The path of the html report file written by `yule` (default is *\$PFX/.samhain.html*).
- `-with-console=PATH` The path of the console (default is */dev/console*). This may be a FIFO.
- `-with-altconsole=PATH` The path of a second console (default is *none*). This may be a FIFO. If defined, console output will always go to *both* console devices (but note that console devices are only used when running as daemon).

B Command line options

B.1 General

- `-D, -daemon` Run as daemon.
- `-s <arg>, -set-syslog-severity=<arg>` Set the severity threshold for syslog. *arg* may be one of `none`, `debug`, `info`, `notice`, `warn`, `mark`, `err`, `crit`, `alert`.
- `-l <arg>, -set-log-severity=<arg>` Set the severity threshold for logfile. *arg* may be one of `none`, `debug`, `info`, `notice`, `warn`, `mark`, `err`, `crit`, `alert`.
- `-m <arg>, -set-mail-severity=<arg>` Set the severity threshold for e-mail. *arg* may be one of `none`, `debug`, `info`, `notice`, `warn`, `mark`, `err`, `crit`, `alert`.
- `-p <arg>, -set-print-severity=<arg>` Set the severity threshold for terminal/console. *arg* may be one of `none`, `debug`, `info`, `notice`, `warn`, `mark`, `err`, `crit`, `alert`.

- x** <arg>, **-set-extern-severity=<arg>** Set the severity threshold for external program(s). *arg* may be one of **none**, **debug**, **info**, **notice**, **warn**, **mark**, **err**, **crit**, **alert**.
- L** <arg>, **-verify-log=<arg>** Verify the integrity of the log file (*arg* is the path of the log file).
- M** <arg>, **-verify-mail=<arg>** Verify the integrity of e-mailed messages (*arg* is the path of the mail box).
- H** <arg>, **-hash-string=<arg>** Print the hash of a string / the checksum of a file, and exit. If *arg* starts with a '/', it is assumed to be a file, otherwise a string. This function is useful to test the hash algorithm.
- z** <arg>, **-tracelevel=<arg>** If compiled with **-enable-debug**: *arg* > 0 to switch on debug output.
If compiled with **-enable-trace**: *arg* > 0 max. level for call tracing.
- i** <arg>, **-milestone=<arg>** If compiled with **-enable-trace**: trace from milestone *arg* to *arg*+1. If *arg* = -1, trace all.
- O**, **-pre1.1.9** Enable compatibility mode.
- c**, **-copyright** Print copyright information and exit.
- h**, **-help** Print a short help on command line options and exit.

B.2 samhain

- t** <arg>, **-set-checksum-test=<arg>** Set file checking to *init*, *update*, or *check*. Use *init* to create the database, *update* to update it, and *check* to check files against the database.
- e** <arg>, **-set-export-severity=<arg>** Set the severity threshold for forwarding messages to the log server. *arg* may be one of **none**, **debug**, **info**, **notice**, **warn**, **mark**, **err**, **crit**, **alert**.
- r** <arg>, **-recursion=<arg>** Set the default recursion level for directories (0 – 99).

B.3 yule

- S**, **-server** Run as server. Only required if the binary is dual-purpose.
- q**, **-qualified** Log received messages with the fully qualified name of client host.

- G** <arg>, **-gen-password** Generate a random password suitable for use in the following option (16 hexadecimal digits).
- P** <arg>, **-password=<arg>** Compute a client registry entry. *arg* is the chosen password (16 hexadecimal digits).

C The configuration file

C.1 General

The configuration file for **samhain** is named **.samhainrc** by default. Also by default, it is placed in **/usr/local/etc**. (Name and location is configurable at compile time). The distribution package comes with a commented sample configuration file.

This section introduces the general structure of the configuration file. Details on individual entries in the configuration files are discussed in Sect. 4.3 (which files to monitor), Sect. 2.3 (what should be logged, which logging facilities should be used, and how these facilities are properly configured), and Sect. 4.8 (monitoring login/logout events).

The configuration file contains several *sections*, indicated by *headings* in *square brackets*. Each section may hold zero or more **key=value** pairs. Keys are not case sensitive, and space around the '=' is allowed. Blank lines and lines starting with '#' are comments. Everything before the first section and after an **[EOF]** is ignored. The **[EOF]** end-of-file marker is optional. Keys are not case sensitive, and space around the '=' is allowed. The file thus looks like:

Example

```
# this is a comment
[Section heading]
key1=value
key2=value

[Another section]
key3=value
key4=value
```

C.1.1 Conditionals

Conditional inclusion of entries for some host(s) is supported via any number of **@hostname/@end** directives. **@hostname** and **@end** must each be on separate lines. Lines in

between will only be read if *hostname* (which may be a *regular expression*) matches the local host.

Likewise, conditional inclusion of entries based on system type is supported via any number of *\$sysname:release:machine/\$end* directives.

sysname:release:machine for the local host can be determined using the command **uname -srm** and may be a *regular expression*.

A *'!'* in front of the *'@'/'\$'* will *invert* its meaning. Conditionals may be *nested* up to 15 levels.

Example

```
@hostname
only read if hostname matches local host
@end
!@hostname
not read if hostname matches local host
@end
#
$sysname:release:machine
only read if sysname:release:machine matches local host
$end
!$sysname:release:machine
not read if sysname:release:machine matches local host
$end
```

C.2 Files to check

Allowed section headings (see Sect. 4.3.1 for more details) are:

```
[Attributes]
[LogFiles]
[GrowingLogFiles]
[IgnoreAll]
[IgnoreNone]
[ReadOnly]
```

Placing an entry under one of these headings will select the respective policy for that entry (see Sect. 4.3.1). Entries under the above section headings must be of the form:

```
dir=[optional numerical recursion depth]path
file=path
```

C.3 Severity of events

Section heading (see Sect. 2.3.1 for more details):

[EventSeverity]

Entries:

SeverityReadOnly=*severity*
SeverityLogFiles=*severity*
SeverityGrowingLogs=*severity*
SeverityIgnoreNone=*severity*
SeverityIgnoreAll=*severity*
SeverityAttributes=*severity*

SeverityFiles=*severity*
SeverityDirs=*severity*
SeverityNames=*severity*

severity may be one of none, debug, info, notice, warn, mark, err, crit, alert.

C.4 Logging thresholds

Section heading (see Sect. 3.1 for more details):

[Log]

Entries:

MailSeverity=[optional specifier]*threshold*
PrintSeverity=[optional specifier]*threshold*
LogSeverity=[optional specifier]*threshold*
SyslogSeverity=[optional specifier]*threshold*
ExportSeverity=[optional specifier]*threshold*
ExternalSeverity=[optional specifier]*threshold*

threshold may be one of none, debug, info, notice, warn, mark, err, crit, alert.

The optional specifier may be one of '!', '*', or '=', which are interpreted as 'all', 'all but', and 'only', respectively.

C.5 Watching login/logout events

Section heading:

[Utmp]

Entries:

LoginCheckActive=*1/0* '1' to switch on, '0' to switch off.
LoginCheckInterval=*seconds* Interval between checks.
SeverityLogin=*severity* Severity for login events.
SeverityLoginMulti=*severity* Severity for logout events.
SeverityLogout=*severity* Severity for multiple logins by same user.

C.6 Checking for kernel module rootkits

Section heading:

[Kernel]

Entries:

KernelCheckActive=*1/0* '1' to switch on, '0' to switch off.
KernelCheckInterval=*seconds* Interval between checks.
SeverityKernel=*severity* Severity for events.

C.7 Miscellaneous

Section heading:

[Misc]

Entries:

Daemon= <i>yes/no</i>	Whether to become a daemon (default: no)
SetLoopTime= <i>seconds</i>	Interval between timestamp messages.
SetFilecheckTime= <i>seconds</i>	Interval between file checks.
ReportOnlyOnce= <i>yes/no</i>	Report only once on a modified file.
ReportFullDetail= <i>yes/no</i>	Report in full detail on modified files.
ChecksumTest= <i>none/init/update/check</i>	The default action.
SetMailTime= <i>seconds</i>	Maximum time interval between mail messages.
SetMailNum= <i>0 - 127</i>	Maximum number of pending mails on internal queue.
SetMailAddress= <i>recipient</i>	Add a recipient e-mail address (max. 8).
SetMailRelay= <i>IP address</i>	The mail relay (for offsite mail).
SamhainPath= <i>path</i>	The path of the process image.

<code>SetLogServer=IP address</code>	The log server.
<code>SetTimeServer=IP address</code>	The time server.
<code>TrustedUser=username(,username,..)</code>	List of additional trusted users.
<code>UseClientSeverity=yes/no</code>	Use severity of client message.
<code>SeverityLookup=severity</code>	Severity for socket peer not equal client address.
<code>SetClientTimeLimit=seconds</code>	Time limit until next client message (server-only).
<code>MessageHeader="%S %T %F %L %C"</code>	Specify custom format for message header.
<code>HideSetup=yes/no</code>	Don't log names of config/database files on startup.
<code>SyslogFacility=LOG_xxx</code>	Set syslog facility (default is LOG_AUTHPRIV).
<code>MACType=HASH-TIGER/HMAC-TIGER</code>	Set type of message auth. code (HMAC).
<code>pre1_1_9=yes/no</code>	Enable compatibility mode.

Remarks: (i) `root` and the effective user are always trusted.

(ii) If no time server is given, the local host clock is used.

(iii) If the path of the process image is given, the process image will be checksummed at startup and exit, and both checksums compared.

C.8 External

Definition of an arbitrary number of external programs/scripts (see Sect. 6). Section heading:

[External]

Entries:

<code>OpenCommand=/full/path/to/program</code>	Starts new command definition.
<code>SetType=log/srv</code>	Type/purpose of the program.
<code>SetCommandline=list</code>	The command line.
<code>SetEnviron=KEY=value</code>	Environment variable (can be repeated).
<code>SetChecksum=TIGER checksum</code>	Checksum of the program.
<code>SetCredentials=username</code>	User whose credentials shall be used.
<code>SetFilterNot=list</code>	Words not allowed in message.
<code>SetFilterAnd=list</code>	Words required (ALL) in message.
<code>SetFilterOr=list</code>	Words required (at least one) in message.
<code>SetDeadtime=seconds</code>	Deadtime between consecutive calls.

C.9 Clients

This section is relevant for `yule` only. Section heading:

`[Clients]`

Entries must be of the form:

`Client=hostname@salt@verifier`

See Sect. 5.2 on how to compute a valid entry.

The hostname must be the same name that the client retrieves from the host on which it runs. Usually, this will be a fully qualified hostname, no numerical address. However, there is no method that guarantees to yield the fully qualified hostname (it is not even guaranteed that a host has one ...).

The only way to know for sure is to set up the client, and check whether the connection is refused by the server with a message like

`Connection attempt from unregistered host hostname`

In that case, *hostname* is what you should use.

C.10 End of file

`[EOF]` Not required, unless there is junk beyond.