NAME

Tomb − the Crypto Undertaker

SYNOPSIS

tomb [options] command [arguments]

DESCRIPTION

Tomb is an application to manage the creation and access of encrypted storage files: it can be operated from commandline and it can integrate with a user's graphical desktop.

Tomb generates encrypted storage files to be opened and closed using their associated keys, which are also protected with a password chosen by the user. To create, open and close tombs a user will need super user rights to execute the tomb commandline utility.

A tomb is like a locked folder that can be safely transported and hidden in a filesystem; it encourages users to keep their keys separate from tombs, for instance keeping a tomb file on your computer harddisk and its key file on a USB stick.

COMMANDS

dig Generates a file that can be used as a tomb and will occupy as much space as its desired initial size, the unlocked *.tomb* file can then be locked using a *key*. It takes a mandatory *-s* option which is the size in megabytes (MiB). Tombs are digged using random data gathered from a non-blocking source (/dev/urandom). For very large tombs this may take up too much time and entropy, then it is possible to use *fallocate(1)* being aware it does not pre-fill with random data, decreasing the tomb's security.

forge Creates a new *key* and prompts the user for a *password* to protect its usage using symmetric encryption. This operation uses random data from a non-blocking source (/dev/urandom) and it may take long only in some cases; to switch using a blocking source the *--use-random* flag can be used. The *-g* option switches on the use of a GPG key instead of a password (asymmetric encryption), then the *-r* option indicates the recipient key; more recipient GPG ids can be indicated (comma separated). The default cipher to protect the key is AES256, a custom one can be specified using the *-o* option, for a list of supported ciphers use *-v*. For additional protection against dictionary attacks on keys, the *--kdf* option can be used when forging a key, making sure that the binaries in *extras/kdf* were compiled and installed on the system.

lock Initializes and locks an empty tomb (made with *dig*) using a key (made with *forge*), making it ready for usage. After this operation, the tomb can only be opened in possession of the key and knowing its password. As in any other command requiring a key, the option *-k* should be used to specify a key file; in case of encryption to GPG recipients the *-g* flag should be used followed by *-r* and the recipient's secret GPG key id. The *-o* option can be used to specify the cipher specification: default is "aes-xts-plain64", old versions of Tomb used "aes-cbc-essiv:sha256". If you are looking for something exotic, also try "serpent-xts-plain64". More options may be found in cryptsetup(8) and Linux documentation. The *--filesystem* option can be used to specify "btrfs" as an alternative filesystem used to format the tomb, in place of the default "ext4". This operation requires root privileges to loopback mount, format the tomb (using LUKS and Ext4), then set the key in its first LUKS slot.

open Opens an existing *tomb file* (first argument) using a key (*-k*) which can also be hidden inside a *jpeg image* (see *bury*/*exhume*) or a long text file (see*cloak*/*uncloak*). If a second argument is given it will indicate the *mountpoint* where the tomb should be made accessible, else the tomb is mounted

in a directory inside /media (if not available it uses /run/media/$USER). The option *-o* can be used to pass mount(8) options (default: rw,noatime,nodev). The *-g* option is needed when using GPG encryption to recipients.

list List all the tombs found open, including information about the time they were opened and the hooks that they mounted. If the first argument is present, then shows only the tomb named that way or returns an error if it's not found. If the option *--get-mountpoint* is used then print a simple list of currently open tomb mountpoint paths.

ps List all the processes found running inside the tombs that are open, printing out their PIDs and owners. This is useful to have an overview of programs that are keeping the tombs busy and would eventually be killed by the *slam* command. The lsof(8) utility is used internally to enumerate processes running in one or all tombs.

index Creates or updates the search indexes of all tombs currently open: enables use of the *search* command using simple word patterns on file names. Indexes are created using mlocate's updatedb(8) and swish-e(1) if they are found on the system. Indexes allow one to search very fast for filenames and contents inside a tomb, they are stored inside it and are not accessible if the Tomb is closed. To avoid indexing a specific tomb simply touch a *.noindex* file in it.

search Takes any string as argument and searches for them through all tombs currently open and previously indexed using the *index* command. The search matches filenames if mlocate is installed and then also file contents if swish++ is present on the system, results are listed on the console.

close Closes a currently open tomb. If more tombs are open, the first argument should be used to specify the name of the tomb to be closed, or *all* to close all currently open tombs. This command fails if the tomb is in use by running processes (to force close, see *slam* below).

slam Closes a tomb like the command *close* does, but it doesn't fail even if the tomb is in use by other application processes: it looks for and closes each of them (in order: TERM, HUP, KILL). This command may provoke unsaved data loss, but assists users to face surprise situations. It requires *lsof* else it falls back to *close*.

passwd

Changes the password protecting a key file specified using *-k*. With keys encrypted for GPG recipients use *-g* followed by *-r* to indicate the new recipient key, or a comma separated list.. The user will need to know the key's current password, or possess at least one of the current recipients GPG secret keys, because the key contents will be decoded and reencoded using the new passwords or keys. If the key file is broken (missing headers) this function also attempts its recovery.

setkey Changes the key file that locks a tomb, substituting the old one with a new one. Both the old and the new key files are needed for this operation and their passwords or GPG recipient(s) secret keys must be available. The new key must be specified using the *-k* option, the first argument should be the old key and the second and last argument the tomb file. Use the *-g* option to unlock the tomb with a GPG key, the *-r* to indicate the recipient or a comma separated list for more than one recipient.

resize Increase the size of a tomb file to the amount specified by the *-s* option, which is the new size in megabytes (MiB). Full access to the tomb using a key (*-k*) and its password is required. Tombs can only grow and can never be made smaller. This command makes use of the cryptsetup(8) resize

feature and the resize2fs command: its much more practical than creating a new tomb and moving everything into it. There is no data-loss if a failure occurs during resize: the command can be re-launched and the resize operation will complete.

engrave
This command transforms a tomb key into an image that can be printed on paper and physically stored as backup, i.e. hidden in a book. It Renders a QRCode of the tomb key, still protected by its password: a PNG image (extension *.qr.png*) will be created in the current directory and can be later printed (fits an A4 or Letter format). To recover an engraved key one can use any QRCode reader on a smartphone: save it into a file and then use that file as a key (*-k*).

bury Hides a tomb key (*-k*) inside a *jpeg image* (first argument) using *steganography*: the image will change in a way that cannot be noticed by human eye and hardly detected by data analysis. This option is useful to backup tomb keys in unsuspected places; it depends from the availability of *steghide*. Use the *-g* flag and *-r* option followed by recipient id to use GPG asymmetric encryption.

exhume
This command recovers from jpeg images the keys that were previously hidden into them using *bury*. Exhume requires a key filename (*-k*) and a *jpeg image* file (first argument) known to be containing a key. If the right key password is given, the key will be exhumed. If the password is not known, it is very hard to verify if a key is buried in any image or not.

cloak Hides a tomb key (*-k*) inside a *long plain-text file* (first argument) using *steganography*: the text will change in a way that can hardly be noticed by human eye and hardly detected by data analysis. This option is useful to backup tomb keys in unsuspected places; it depends from the availability of *cloakify* and consequently *python2*. This function does not support asymmetric encryption using the *-g* flag.

uncloak
This command recovers from long plain-text files the keys that were previously hidden into them using *cloak*. Cloak requires a key filename (*-k*) and a *plain-text* file (first argument) known to be containing a key. If the right key password is given, the key will be exhumed. If the password is not known, it is quite hard to verify if a key is buried in a text or not.

OPTIONS

-k <*keyfile*>
For all operations requiring a key, this option specifies the location of the key file to use. Arguments can also be *jpeg image* files where keys have been hidden using the *bury* or *cloak* commands, or text files retrieved from *engraved* QR codes. If the *keyfile* argument is "-" (dash), Tomb will read the key from stdin (blocking).

-n Skip processing of exec-hooks and bind-hooks if found inside the tomb. See the*HOOKS* section in this manual for more information.

-p When opening a tomb, preserves the ownership of all files and directories contained in it. Normally the *open* command changes the ownership of a tomb's contents to the UID and GID of the user who has successfully opened it: it is a usability feature in case a tomb is used by a single user across different systems. This flag deactivates this behaviour.

-o Manually specify mount options to be used when opening a tomb instead of the default *rw,noatime,nodev*, i.e. to mount a tomb read-only (ro) to prevent any modification of its data. Can also be used to change the symmetric encryption algorithm for keys during *forge* operations (default *AES256*) or the LUKS encryption method during *lock* operations (default *aes-xts-plain64*).

-f Force flag, currently used to override swap checks, might be overriding more wimpy behaviours in future, but make sure you know what you are doing if you force an operation.

-s <*MBytes*>
When digging or resizing a tomb, this option must be used to specify the *size* of the new file to be created. Units are megabytes (MiB).

-g Tell tomb to use a asymmetric GnuPG key encryption instead of a symmetric passphrase to protect a tomb key. This option can be followed by *-r* when the command needs to specify recipient(s).

-r <*gpg_id>[,<gpg_id2>]*
Provide a new set of recipient(s) to encrypt a tomb key. *gpg_ids* can be one or more GPG key ID, comma separated. All GPG keys must be trusted keys in GPG.

--kdf <*itertime*>
Activate the KDF feature against dictionary attacks when creating a key: forces a delay of <*itertime*> times every time this key is used. The actual time to wait depends on the CPU speed (default) or the RAM size (argon2) of the computer where the key is used. Using 5 or 10 is a sane amount for modern computers, the value is multiplied by 1 million.

--kdftype *argon2* | *pbkdf2*
Adopt the *argon2* algorithm for KDF, stressing the RAM capacity rather than the CPU speed of the computer decrypting the tomb. Requires the *argon2* binary by P-H-C to be installed, as packaged by most distros. Default is *pbkdf2*.

--kdfmem <*memory*>
In case of *argon2* KDF algorithm, this value specifies the size of RAM used: it consists of a number which is the elevated power of two in bytes. Default is 18 which is 262 MiB (2^18 bytes).

--sudo <*executable*>
Select a different tool than sudo for privilege escalation. Alternatives supported so far are: pkexec, doas, sup, sud. For any alternative to work the executable must be included in the current PATH.

--sphx-user <*username*>
Activate the SPHINX feature for password-authenticated key agreement. This option indicates the <*username*> used to retrieve the password from a sphinx oracle key reachable via TCP/IP.

--sphx-host <*domain*>
Activate the SPHINX feature for password-authenticated key agreement. This option indicates the <*domain*> used to retrieve the password from a sphinx oracle daemon reachable via TCP/IP. This is not the network address of the daemon, which is configured in /etc/sphinx

-h Display a help text and quit.

-v Display version and quit.

-q Run more quietly

-D Print more information while running, for debugging purposes

DEV MODE

--no-color
Suppress colors in console output (needed for string parsing by wrappers).

--unsafe
Enable using dev-mode arguments, i.e. to pass passwords from commandline options. This is mostly used needed for execution by wrappers and testing suite.

--use-random
Use a blocking random source. Tomb uses by default /dev/urandom since the non-blocking source of Linux kernel doesn't degrades the quality of random.

--tomb-pwd <string>
Use string as password when needed on tomb.

--tomb-old-pwd <string>
Use string as old password when needed in tomb commands requiring multiple keys, like *passwd* or *setkey*.

-U Switch to this user ID when dropping privileges.

-G Switch to this group ID when dropping privileges.

-T Switch to this TTY terminal when dropping privileges.

HOOKS

Hooks are special files that can be placed inside the tomb and trigger actions when it is opened and closed; there are two kinds of such files: *bind-hooks* and *exec-hooks* can be placed in the base root of the tomb.

bind-hooks
This hook file consists of a simple text file named *bind-hooks* containing a two column list of paths to files or directories inside the tomb. The files and directories will be be made directly accessible by the tomb *open* command inside the current user's home directory. Tomb uses internally the "mount −o bind" command to bind locations inside the tomb to locations found in $HOME. In the first column are indicated paths relative to the tomb and in the second column are indicated paths relative to $HOME contents, for example:

```
mail          mail
.gnupg        .gnupg
.fmrc         .fetchmailrc
.mozilla      .mozilla
```

exec-hooks
This hook file gets executed as user by tomb with the first argument determining the step of execution (*open* or *close*) and the second being the full path to the mountpoint. The *exec-hooks* file should be executable (ELF or shell script) and present inside the Tomb. Tomb executes this hook as user and adds the name, loopback device and dev-mapper device paths as additional arguments for the *close* command.

PRIVILEGE ESCALATION

The tomb commandline tool needs to acquire super user rights to execute most of its operations: so it uses sudo(8) or other configured tools, while pinentry(1) is adopted to collect passwords from the user. Tomb executes as super user only when required.

To be made available on multi user systems, the superuser execution of the tomb script can be authorized for users without jeopardizing the whole system's security: just add such a line to */etc/sudoers*:

```
username ALL=NOPASSWD: /usr/local/bin/tomb
```

To avoid that tomb execution is logged by *syslog* also add:

```
Cmnd_Alias TOMB = /usr/local/bin/tomb
Defaults!TOMB !syslog
```

PASSWORD INPUT

Password input is handled by the pinentry program: it can be text based or graphical and is usually configured with a symlink. When using Tomb in X11 it is better to use a graphical pinentry-gtk2 or pinentry-qt because it helps preventing keylogging by other X clients. When using it from a remote ssh connection it

might be necessary to force use of pinentry-curses for instance by unsetting the DISPLAY environment var.

SWAP

On execution of certain commands Tomb will complain about swap memory on disk when present and *abort if your system has swap activated*. You can disable this behaviour using the *--force*. Before doing that, however, you may be interested in knowing the risks of doing so:

- During such operations a lack of available memory could cause the swap to write your secret key on the disk.
- Even while using an opened tomb, another application could occupy too much memory so that the swap needs to be used, this way it is possible that some contents of files contained into the tomb are physically written on your disk, not encrypted.

If you don’t need swap, execute *swapoff -a*. If you really need it, you could make an encrypted swap partition. Tomb doesn’t detect if your swap is encrypted, and will complain anyway.

DENIABILITY

The possibility to have an encrypted volume which is invisible and cannot be detected is called "deniability". The cryptographic layer of the device mapper in Linux (dm-crypt) does not implement deniability. Tomb is just a wrapper on top of that and it doesn’t add cryptographic deniability. However a certain way of using tomb can facilitate a weak sort of deniability outside of the scenario of seized devices and forensic analysis of files and blocks on disc.

For instance to eliminate any trace of tomb usage from the shell history ZSh users can activate the "HISTIGNORESPACE" feature and prefix all invocations of tomb with a blank space, including two lines in ".zshrc":

```
export HISTIGNORESPACE=1
alias tomb=’ tomb’
```

PASSWORD INPUT

Tomb uses the external program "pinentry" to let users type the key password into a terminal or a graphical window. This program works in conjunction with "gpg-agent", a daemon running in background to facilitate secret key management with gpg. It is recommended one runs "gpg-agent" launching it from the X session initialization ("~/.xsession" or "~/.xinitrc" files) with this command:

```
eval $(gpg-agent --daemon --write-env-file "${HOME}/.gpg-agent-info")
```

In the future it may become mandatory to run gpg-agent when using tomb.

SHARE A TOMB

A tomb key can be encrypted with more than one recipient. Therefore, a tomb can be shared between different users. The recipients are given using the *-r* (or/and *-R*) option and if multiple each GPG key ID must be separated by a comma (,). Sharing a tomb is a very sensitive action and the user needs to trust that all the GPG public keys used are kept safe. If one of them its stolen or lost, it will be always possible to use it to access the tomb key unless all its copies are destroyed. The *-r* option can be used in the tomb commands: *open*, *forge setkey*, *passwd*, *bury*, *exhume* and *resize*.

SPHINX (PAKE)

Using the package libsphinx ⟨https://github.com/stef/libsphinx⟩ and its python client/daemon implementation pwdsphinx ⟨https://github.com/stef/pwdsphinx⟩ is possible to store and retrieve safely the password that locks the tomb. Using this feature will make it impossible to retrieve the password without the oracle

sphinx server running and reachable. Each key entry needs a username and a domain specified on creation and a password that locks it.

SPHINX makes it impossible to maliciously retrieve the password locking the tomb key without an attacker accessing both the server, the sphinx password and the tomb key file.

EXAMPLES

- Create a 128MB large "secret" tomb and its keys, then open it:

```
tomb dig -s 128 secret.tomb

tomb forge secret.tomb.key

tomb lock secret.tomb -k secret.tomb.key

tomb open secret.tomb -k secret.tomb.key
```

- Open a Tomb using the key from a remote SSH shell, without saving any local copy of it:

```
ssh user@my.shell.net 'cat .secrets/tomb.key' | tomb open secret.tomb
```

- Open a Tomb on a remote server passing the unencrypted local key on stdin via SSH, without saving any remote copy of it:

```
gpg -d .secrets/tomb.key | ssh server tomb open secret.tomb -k cleart
```

- Create a bind hook that places your GnuPG folder inside the tomb, but makes it reachable from the standard $HOME/.gnupg location every time the tomb will be opened:

```
tomb open GPG.tomb -k GPG.tomb.key
echo ".gnupg .gnupg" > /media/GPG.tomb/bind-hooks
mv ~/.gnupg /media/GPG.tomb/.gnupg && mkdir ~/.gnupg
tomb close GPG && tomb open GPG.tomb -k GPG.tomb.key
```

- Script a tomb to launch the Firefox browser every time is opened, keeping all its profile data inside it:

```
tomb open FOX.tomb -k FOX.tomb.key
cat <<EOF > /media/FOX.tomb/exec-hooks
#!/bin/sh
if [ "$1" = "open" ]; then
  firefox -no-remote -profile "$2"/firefox-pro &
fi
EOF
chmod +x     /media/FOX.tomb/exec-hooks
  mkdir /media/FOX.tomb/firefox-pro
```

- Script a tomb to archive Pictures using Shotwell, launching it on open:

```
tomb open Pictures.tomb -k Pictures.tomb.key
cat <<EOF > /media/Pictures.tomb/bind-hooks
Pictures Pictures
```

```
EOF
        cat <<EOF > /media/Pictures.tomb/exec-hooks
#!/bin/sh
if [ "$1" = "open" ]; then
  which shotwell > /dev/null
  if [ "$?" = "0" ]; then
    shotwell -d "$2"/Pictures/.shotwell &
  fi
fi
EOF
      chmod +x /media/Pictures.tomb/exec-hooks
```

BUGS

Please report bugs on the Github issue tracker at ⟨https://github.com/dyne/Tomb/issues⟩

One can also try to get in touch with developers via the #dyne chat channel on *https://irc.dyne.org*.

COPYING

AVAILABILITY

The most recent version of Tomb sourcecode and up to date documentation is available for download from its website on *https://tomb.dyne.org*.

SEE ALSO

cryptsetup(8)

pinentry(1)

gpg-agent(1)

GnuPG website: https://www.gnupg.org

DM-Crypt website: https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

LUKS website: https://gitlab.com/cryptsetup/cryptsetup/wikis/home