## NAME

**ee** — An encryption shell that is easy to use for terminal users.

## SYNOPSIS

**ee** [ **-a** *name*] [ **-s** *integer*]

## DESCRIPTION

**ee** is a simple shell that runs commands from the PATH of the shell from which **ee** was invoked. Additionally, there exist commands that support exchanging and maintaining cryptographic keys, verifying keys, backing up and restoring key stores and using keys to encrypt and decrypt files. Furthermore, there exist commands that fetch attachments from mail and that attach files to mails. The latter commands are platform-specific.

This tool is work in progress. Certain convenience features are still missing, for example, an automatic setup of the necessary directories and permissions. This must be done by hand, see the installation instructions below.

The following options are available:

**-a** *name*
: Runs **ee** with an alternative keystore. This is useful to test exchanging keys and encrypting and decrypting files from the same user account. *name* must consist only of alphanumerical characters.

**-s** *integer*
: Runs **ee** with different display styles, depending on *integer* and the number of built-in styles. Currently, **ee** supports three styles numbered from 0 to 2. Styles can be added and customized by editing files *customization.c* and *customization.h*.

## INSTALLATION

Before running **ee** please create the directory *$(HOME)/.ee/secret/keys* in your home directory. This is where **ee** is going to create and maintain your cryptographic key material.

In a production setting, the directory *secret* should be owned by root and only root should be able to list or traverse this directory. The other directories should be owned by the user who owns the given home directory. In this case, **ee** must be given `setuid`(2) root status. Do not run **ee** as the root user or else it attempts to open a key store for root.

Some commands of the **ee** shell invoke scripts to perform platform-specific operations. These scripts are

*ee_fetch.sh*
: This script fetches the first attachment of the first mail selected in the platform's favorite mail user agent. The current script performs this task for OS X and Apple's Mail.app, using an applescript snippet.

*ee_attach.sh*
: This script opens a new mail window in the platform's favorite mail user agent and attaches a file to it. The current script performs this task for OS X and Apple's Mail.app, using an applescript snippet. In order to send encrypted mail, the attached file must have been encrypted prior.

In order to make these scripts available to **ee** you need to copy them to a directory in your shell's PATH. For example, you can create and use the directory $HOME/bin for that. Before invoking **ee** you should set up the PATH as follows:

```
$ export PATH=$PATH:$HOME/bin
```

## DESIGN RATIONALE

If an adversary can obtain root privileges on your machine then he can subvert or bypass any password protections you have in place, for example, in order to protect `gpg`(1) keys. At the same time, `gpg`(1) requires that you enter your password whenever an operation requests your private encryption key. Since `gpg`(1) key rings can be copied easily by Trojan Horse applications, the password of your key ring must be chosen so

that it resists offline brute force attacks. The same holds for key chains on OS X.

**ee** instead allows you to set a password that is short and easy to remember, and enforces a three-strikes rule on it. Once you have set your password, you must authenticate yourself to **ee** in at most three logins or else your keystore is locked. This protection is enforced based on the *secret* file permissions and the setuid(2) protection of your UNIX system. The password must be entered only at the beginning of a **ee** session. Once **ee** runs, you can perform an arbitrary number of encryptions and decryptions without having to enter your password, until you quit the **ee** shell.

The most important concept of **ee** is that of a *nick* name. A nick is a handle that you use to identify your intended communication partner. It is also what **ee** uses to tell you who the originator of a message is that you decrypt. You begin your communication by establishing a nick for a peer. Once you have done this, you can export the key that you are going to use when communicating with that nick. **ee** maintains a separate (public) key for each of your communication partners. This is advantageous if you wish to travel abroad where you may be forced to disclose keys. It allows you to select beforehand which keys you wish to take with you and with whom to communicate until you are back in a jurisdiction you trust.

In contrast to gpg(1), **ee** offers support for forward secrecy and for backing up, restoring and merging key stores. The philosophy of **ee** is to offer few commands that are easy to understand and use, and to build more complex operations on top of them. In what follows we introduce the functions of **ee** based on use-cases. More frequent use-cases come first.

## EXCHANGING KEYS

This man page explains most but not all commands that **ee** understands. Instead it is meant to get you started using examples that teach you how to accomplish typical goals. You can figure out the supported commands by typing:

```
ee> help
```

Typing a command name without arguments prints a brief usage statement.

In what follows, let's assume that you are "Alice". In order to get started communicating with your friend "Bob", you should do the following:

```
ee> add bob "My dear friend Bob"
ee> export bob
<key will be printed here>
```

This prints the key you need to send to Bob. Cut and paste it into an e-mail to Bob. The key looks best when using a fixed-width font in your e-mail. Note that bob is the nick and Bob is the name. Alternatively, you can print the key to a file, like this:

```
ee> export bob filename
```

Bob should do something like the following. We assume that Bob wishes to call you "alice".

```
ee> add alice "My dear friend Alice"
ee> import alice
<paste the key received from Alice here>
^D
ee> export alice
<key will be printed here>
```

The import command waits for a key to be pasted into the shell. Once Bob has done this, he can finish by pressing ^D. Alternatively, the key can be imported from a file, as in the case of exporting. Bob now has the key from Alice and his own key for Alice.

The export command exports the key that Bob needs to send to Alice. Bob can already encrypt messages for Alice because he has both keys, his own and that of Alice. He must still send his own key to Alice before Alice can decrypt messages. Alice imports the key from Bob in the same fashion, that is, she enters

```
ee> import bob
<paste the key received from Bob here>
^D
ee>
```

If Alice and Bob wish to make sure that they are not victims to a man-in-the-middle attack, they can use the **check** command (see *Preventing Man-in-the-Middle Attacks* below). The **show** command shows the known nicks. Dots to the left of the nicks denote whether the key of that nick has been imported (one dot) and whether the nick has been checked and set to trusted (two dots).

## ENCRYPTION AND FEATURES SPECIFIC TO OS X

On OS X, **ee** supports a **fetch** and an **attach** command. The fetch command saves the first attachment of the currently selected mail in Mail.app under the given name. A typical workflow would be, for example:

```
ee> fetch c
The file should have been saved now.
ee> dec c m
This message is from bob.
ee> less m
```

The attach command does the opposite. It composes a new mail and attaches the given file. A typical workflow would be, for example:

```
ee> emacs m
ee> enc bob m c
ee> attach c bob
```

Here, Alice composes her reply to Bob by editing the decrypted message from our previous example. The second argument to **attach** is optional. If present then **ee** performs an AddressBook lookup for a person record with the *nickname* field value "bob". If such a record exists then the primary e-mail address is entered into the "To:" field of the composed e-mail. Alice still has to press the send button, though, for safety reasons.

The AddressBook lookup is done by a separate tool with the name **ee_ab**, which is included in the **ee** distribution.

## PUBLISHING GENERIC PUBLIC KEYS

**ee** is designed so that key entries are write-once, in order to minimize the opportunity for errors. If Alice created a nick **anybody** and exported and published the key then she would be able to import a key to this nick only once. Sometimes it is desirable to publish a public key that anybody can use to send encrypted messages to Alice. In order to support this model while still keeping the opportunity for errors low, **ee** has a **clone** command. The clone command duplicates a nick with a new name except that it clears the imported key to which the nick is bound. This allows re-using public keys. Here is an example.

Alice creates a nick that anyone can use to contact her, exports the key and publishes it, for example, on her website:

```
ee> add anybody
ee> export anybody
```

Bob downloads the key, creates a nick for Alice and imports the key. Then he exports his key for Alice, encrypts a message for Alice and sends her the key and the ciphertext. Alice receives Bob's key *k* and the ciphertext *c*. She clones **anybody** into **somebody** and imports the received key.

```
ee> clone anybody somebody
ee> import somebody k
ee> dec c m
This message is from somebody.
```

Based on what Alice reads, she decides to use **bob** as the nick for the sender of the message. Hence she renames **somebody** into **bob**.

```
ee> rename somebody bob
```

If Alice accidentally imports a key into **anybody** then she can fix this easily as follows:

```
ee> rename anybody bob
ee> clone bob anybody
```

The **show** command shows whether a nick has an imported key. One or two asterisks to the right of the nick indicate that a key has been imported. Nicks with no asterisk are "open", they do not have an imported key yet. (Note that this is style-dependent, we assumed the default style.)

The name **anybody** is not special. Alice can mainain different identities by publishing different public keys.

## ENCRYPTING MESSAGES TO ONESELF

Occasionally, one may wish to encrypt files for private use. The files are not meant to be decrypted by anyone else. Such a scenario does not require an exchange of keys because the sending and receiving party are the same.

In order to encrypt files privately, one creates a *private* nick, literally. For example, Alice types:

```
ee> add private
ee> enc private m c
```

which encrypts file *m* to ciphertext file *c*. Decryption works as usual. The nick **private** has a special meaning. It tells **ee** that it is supposed to create a key entry with an additional public key. Hence, **ee** simulates a key exchange internally. The "other" private key is deleted and never used.

If multiple private keys are desired then this can be achieved straightforwardly by renaming the private nick and by creating a new private one, as in the following example:

```
ee> add private
ee> rename private spare
ee> add private
```

Private nicks are set to "checked" state because their authenticity is implicit.

By adding the nick *passwd*, one generates a private nick with keys derived from a password in a deterministic fashion instead of generating keys randomly. Hence:

```
ee> add passwd
Please enter a password (max. 64 chars):
ee> delete passwd
```

creates and deletes a nick that is easily re-created by using the same password again, even if the key store is lost or deleted.

## BACKING UP AND RESTORING KEY STORES LOCALLY

The *passwd* nick we introduced previously is useful to perform local backups. In the following example, we create a password-protected backup of our key store, then we wipe our key store clean, and we restore it afterwards.

```
ee> add passwd
Please enter a password (max. 64 chars):
ee> backup passwd local.keys
ee> destroy
ee> quit
$ ee
ee> add passwd
Please enter a password (max. 64 chars):
ee> restore local.keys
From: passwd
Date: Fri Jul 31 00:42:31 2015
Done with 7 entries.
```

(The command **destroy** isn't implemented in **ee**. For illustration, we assume that it removes all keys.) Since **restore** merges key stores into the current one, this strategy can be used to apply different levels of protection to different sets of keys.

## BACKING UP AND RESTORING KEY STORES REMOTELY

In order to prepare for a remote backup, you install **ee** on a backup computer and on your production computer. Next, you perform a key exchange between the two computers. In the following example, we assume that *oak* is the backup machine and *monster* is the production machine and we assume that keys have been exchanged between them. The key pair on *oak* has nick monster and vice versa.

On monster:

```
ee> backup oak monster.keys
ee> scp monster.keys oak:/tmp/monster.keys
```

On oak, you import the backup by typing:

```
ee> restore /tmp/monster.keys
From: monster (unchecked)
Date: Fri Jul 31 00:39:28 2015
Done with 7 entries.
```

If monster crashes at some point in time and all data is lost then you repeat the procedure in the opposite direction.

The restore function takes care not to loose data. Restore operations are idempotent. If conflicts arise because Alice attempts to consolidate her key stores on multiple machines then the conflicting entries are not imported. Typically, the conflicts can be resolved by renaming a nick and performing another restore. Entries that are mirror twins of an existing entry (for example, the nicks oak and monster) are skipped automatically.

## PREVENTING MAN-IN-THE-MIDDLE ATTACKS

**ee** does not support key signing or certificates. This is a feature and not a bug. Instead, it supports checking the integrity of exchanged keys over an authenticated channel, for example, voice communication.

Alice invokes the **check** command with a nick. This yields a challenge and its expected response. Alice tells Bob the challenge. Bob invokes the **check** command with his nick for Alice and the challenge. This yields a response. Alice and Bob compare their responses and if they are equal then with a certain probability, the keys are integer. If the responses differ then it is certain that either one of them has imported a wrong key or an attack is in progress. An example follows.

This is what Alice types:

```
ee> check bob
    For nick: bob
   With tags: My dear friend Bob

  Check with: 31764
Compare with: 44953

You are in trouble if the comparison fails!
```

Alice calls Bob and tells him the challenge 31764. This is what Bob types:

```
ee> check alice 31764
    For nick: alice
   With tags: My dear friend Alice

Compare with: 44953

You are in trouble if the comparison fails!
```

Subsequently, Alice and Bob compare the responses, in this case 44953. Since they match, Alice can set the "checked" flag on Bob's nick, as shown next:

```
ee> check bob ok
```

Bob does likewise. If Alice and Bob are paranoid then they can go through multiple challenges and responses. This further decreases the chances that an attack is not detected. Three rounds should satisfy the ultra-paranoids among us. The security guarantees of this checking procedure are independent of the computational power of the adversary.

Once checked, a nick shows up with two dots to its left in the listing of nicks. The second dot carries over in authority is transfered from one nick to a new one, as explained below.

## FORWARD SECRECY AND TRANSFER OF AUTHORITY

**ee** provides forward secrecy through a transfer of authority from one nick to a new nick. In order to move to a new key, Alice creates a new nick for the existing nick of Bob, for example, the nick bob2. Next, she creates a *transfer ticket* from bob to bob2 using the **transfer** command. This yields a text output that is similar to that of exporting a key. Alice sends the output to Bob and asks him to create a new nick as well.

Bob creates a new nick for Alice, say alice2, and imports the transfer ticket with the **confirm** command. Next, Bob creates a transfer ticket from alice to alice2 and sends it to Alice. Alice imports the transfer ticket as Bob did. On both imports, **ee** should confirm that the first nick authorizes the second nick. An example follows.

Alice has a nick "bob" already. She issues the following commands:

```
ee> add bob2
ee> transfer bob bob2
<transfer ticket is output here>
```

She sends the transfer ticket to Bob and receives his ticket in return. She then enters:

```
ee> confirm bob bob2
<paste transfer ticket from Bob here>
^D
Success, bob confirms bob2.
ee>
```

Bob has a nick "alice" already. When he receives Alice's transfer ticket, he issues the following commands:

```
ee> add alice2
ee> confirm alice alice2
<paste transfer ticket from Alice here>
^D
Success, alice confirms alice2.
ee> transfer alice alice2
<transfer ticket is output here>
ee>
```

Creating and confirming transfer tickets implicitly imports the new keys associated with the nicks alice2 and bob2. Alternatively, Alice and Bob can export and import the keys of their new nicks alice2 and bob2, first. In a second step, they can create and confirm the transfer tickets as shown before. The process of creating and confirming transfer tickets can be repeated without harm. All it does is check that the owner of the first nick knows the secret keys of the second nick and vice versa.

For as long as Alice keeps the old nick "bob" around, she can decrypt old messages addressed to the old nick. However, she should stop using the old nick and should delete it as soon as she is sure that Bob and her have confirmed the transfer. If she keeps old nicks around then an adversary who breaks into her computer can steal the old keys and use them to decrypt old messages.

## IMPLEMENTATION NOTES AND ACKNOWLEDGEMENTS

**ee** has no dependencies other than *libc* and consists of about 5K source lines of C code. This already includes a copy of Daniel J. Bernstein's *tweetnacl*. Its small size is meant to facilitate source code audits. For comparison, the source code of gpg(1) is in the order of 170K source lines of code. **ee** features command line editing with history and pipes. The design of the *check* command evolved from discussions with Benjamin Gueldenring at Freie Universitaet Berlin.

## SEE ALSO

gpg(1)

## AUTHOR

The **ee** shell was written by Volker Roth in 2015. The current maintainer is Volker Roth <volker.roth@fu-berlin.de>.

## BUGS

The memory mapped key store is not protected against concurrent access. It might be possible to get it into an inconsistent state using pipes and internal commands.