Security's corner

SSH*Lock – SSH on Steroids !*

by Franck Porcher, PhD

Whether locally, through an attached terminal, or remotely via SSH, standard access to computers has not evolved much in the last 40 years in spite of continuous technological advances, and still heavily relies on the traditional password-based authentication mechanism which, acting as the unique “protective door” between the world and the computer's valuable vault, most unfortunately represents a *single point of failure.* Should this access mechanism yield to a successful password attack, the targeted system would instantly be exposed and compromised, leaving little defense in terms of security to slow the advance of the attacker towards the computer's precious content (In the case of remote logons, a single SSH layer, no mater how tightly configured, *does not* eliminate the single point of failure).

We present SSH*Lock*, a security scheme based on SSH and devised to completely eliminate these threats in order to render computers virtually impenetrable. SSH*Lock* is not about remote connections per se, but a more general approach to securing and strengthening the logon access mechanism to computers in order to defeat attacks on passwords and remove single point of failure access weaknesses.

SSH*Lock* transposes into software the desirable security features found in physical *security access locks* to provide IT systems with a secure access mechanism in the form of a tight, confined “chamber” used as a pass-thru device between the outside world and the computer's content – two environments that should never be directly in contact with one another – and secured at each end by a reinforced “protective door” for monitoring and access control.

SSH*Lock* brings a wealth of security features which, together, provide for a much safer access to computers over the standard password-based access mechanism, making SSHLock a safer and powerful alternative to any password-based access mechanism, including standard SSH services for remote access.

First, we provide some necessary background related to password security attacks. Then we present SSH*Lock*, its concept and desirable security features, the principles guiding its implementation, and its overall modus operandi. The second part of the article deals with the technical details involved in manually setting-up and administering a *sshlock*. We conclude by presenting the SSHLock software that provides the end user with a simple interface to automate these complex tasks.

Password security attacks

The literature abounds with examples that highlight the considerable harm and damage – political, economic, and social – that may result from unauthorized access to IT systems. Discarding such omnipresent threats remains the greatest challenge of every conscientious system administrator today. Yet, access to computers still heavily relies on the weak traditional <*login* , *password*>-based identification/authentication access mechanism.

In such a mechanism, the login is the name identifying the account one wishes to access, and is generally defined by the system administrator to reflect the final usage of the account (e.g. “webmaster”, “marcel”, “dupont”). On top of this, most IT systems still have a number of accounts whose names are publicly known and well-established by decades of standard practices and habits passed along by generations of system administrators (e.g. “root”, “www”, “apache”, “mysql”). The password is the login's counterpart : an authenticating “secret” most often left to the owner of the account to choose. Together, the clear-text login and the encrypted password constitute an access key to the account. One must know both to be granted access.

Such an access mechanism is plagued with several security weaknesses, and it not a coincidence that *password security attacks* remain the simplest and therefore the most common means to break into IT systems in order to gain unauthorized access to sensitive data and classified information :

- As the unique “security fence” between the world and the valuable content of the computer, this mechanism is a single point of failure, an undesirable feature everywhere security is paramount.
- Systems relying on such an access mechanism (Unices are no exception) are still not configured by default to use the strongest cryptography available to encrypt the passwords (hashing algorithm), resulting in insecure password databases.
- Logins and passwords are still prevalently chosen today to be both semantically significant and therefore easy to remember. Though they are perfectly valid, who wants to remember a login like *“xu0i8qLKjQS”* or a password like *“$, Blz! X2.718^2>e^2?;):nop @ t” ?* As a consequence, and to a large extent, such *access keys* are generally weak from the point of view of security.
- It is still commonly believed that disclosing account names (logins), or making no effort to hide them, does not weaken the security of this access mechanism. This is of course not correct, since the login is indeed an important part of the security access key. Knowing it considerably reduces the effort needed by the attacker to break the key, since only the password is now unknown.

The term “password security attacks”, or simply “password attacks”, refers here to all the available means, methods and techniques that an attacker can deploy to fraudulently obtain or discover valid access keys. Password attacks are mostly used when it is not possible to take advantage of other weaknesses in an encryption system (if any exist) that would make the task easier for the attacker [1].

To fully grasp how such attacks may be carried out, one must realize that the attacker is not always that anonymous individual one may believe, lurking hidden in the shadows of a small, dark, and creepy lab located 20,000 km from home or work.

In most serious cases, the attacker knows what he is doing and does not act randomly. He may well be someone you have been in contact with, for instance :

- A personal enemy.
- An unscrupulous or vengeful staff.
- A political opponent.
- An industrial competitor.

To obtain (steal) the credentials he needs to successfully break into your system, the attacker can rely on a number of options, including :

- **Spying**, the oldest of all methods. If the attacker is near or in the immediate vicinity of its victim, he may search out valuable hints, possibly by observing and digging around the office, under the keyboard or the deck, around the screen, inside the drawers, in the garbage bins, etc.
- **Social Engineering**, a subtle form of spying wherein the attacker tries to gather personal information directly related to the targeted account's owner, and likely to be used as potential valid credentials [2], for instance :
    - Login itself: amongst dozens of accounts, it is not that unusual that one may be using the login itself as a password.
    - Full name, date of birth, social security number, phone number,...
    - A mother may use the name or birth date of her children.
    - A man may use the name of his wife, the date of their anniversary, the name of his secretary or that of his favorite hero.
    - People may use information pertaining to their other half (name, birthday date, phone number, etc.)
    - The attacker may also resort to using strategies of impersonation, for instance pretending over the phone to be the system administrator, or a supervisor of higher rank, asking authoritatively for some access keys, or, conversely, he may impersonate a user and politely ask the technical support to reset the access key.
- **Dictionary attack** and its variations. In contrast to a brute-force attack where a large proportion of the access key space is searched systematically, dictionary attacks use specialized dictionaries as sources of keys deemed most likely to succeed. Dictionary attacks (for instance against easily obtained password databases) often succeed because there is a natural tendency amongst the majority of people to choose short passwords from ordinary words and their simple variants, like appending a digit or a punctuation character, or using a zero for an O (the letter) or vice versa.

Conditions for the success of a password security attack

Whichever way the attacker may choose to mount his attack, its success relies on the two following conditions being satisfied *at the same time* :

- **There are** sufficiently weak access keys within the system so the attacker can expect to uncover some of them within a reasonable time frame (condition a).

AND

- The targeted IT system **allows** *verification*, the process by which an attacker can *verify* the validity of guessed / generated access keys (condition b).

Counter-measure

To defeat this type of attack, the counter-measures are therefore obvious. It suffices in theory that one *only* of the two previous conditions *is not satisfied*, namely :

- (¬ “condition a”) **There *does not* exist** any weak key in the system (counter-measure a).

OR

- (¬ “condition b”) The targeted IT system **prevents** *verification* (counter-measure b).

The “counter-measure a” (the negation of “condition a”) is satisfied if we can design a mandatory, non-derogatory security policy that enforces some stronger cryptography to encrypt the passwords (hashing algorithm) and requires users to use access keys virtually impossible to guess. However, the stronger the policy, the harder the keys are to remember, and the more likely the users will find ways to not forget them, for example by writing them down (Over the years, we have caught two thirds of our students and professional users doing exactly that !).

Therefore, we are left with no option but “counter-measure b”. However, in using the password-based authentication access mechanism, it is impossible to distinguish between an authorized user and an attacker who has acquired the access key due to the difficulty of removing “condition a”. Therefore, “counter-measure b” can only be satisfied if we remove password-based authentication access mechanism altogether or if we add a second, stronger authentication mechanism in order to remove the point of failure identified earlier.

SSH*Lock*

SSHLock does both, and more:

- It disables the password-based authentication access mechanism.
- It provides a sophisticated access mechanism based on two *independent* SSH layers working in series, and relying on strong public-key cryptography *only*, thus adding access redundancy and eliminating the single point of failure identified earlier.
- It provides a *honey-pot* devised to trap intruders.
- It enforces the use of strong passphrases to protect key-pairs.
- It removes the *verification* principle.

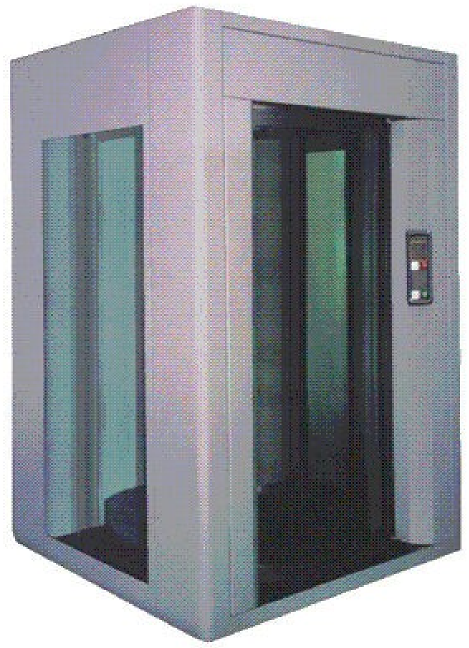
SSH*Lock* draws from physical security access locks their most desirable security features, mocks them up in software, strengthens and combines them together in order to create a strong access mechanism made of a tight, confined “chamber” (the honey pot) used as a decoupling pass-thru device between the outside world and the computer's precious content. Given these two environments must not be in contact with one another, the pass-thru device is secured at each end by a reinforced “protective door” to implement monitoring and tight access control. As a result, the weaknesses identified earlier are all removed, and threats associated with password security attacks are eliminated, making breaking into an SSHLock-protected computer much harder, if not virtually impossible.

Security access lock

A *lock* is typically a small, secure, specialized, confined, restricted “buffer zone” (the chamber) used exclusively as a *pass-thru* device between two subsystems that should not directly be in contact with one another. At each extremity of the chamber is a tight, protective, reinforced door. The two doors do not open simultaneously but only one after the other.

Better known examples of this type of lock are *airlocks*, devices that permit the passage of people and objects between a pressure vessel and its surroundings while minimizing the change of pressure in the vessel and loss of air from it. An airlock may be used for passage between environments of different gases rather than different pressures, to minimize or prevent the gases from mixing. [3]

A *security access lock* is a special type of lock used to control and monitor access to sensitive areas, for example airport~~s~~ boarding areas. The most important feature of a security lock is that it is the *only* access to the sensitive area : any other access would only weaken the purpose of the lock. A good example of a security access lock is the type of device used by most modern banks to control and monitor access to the bank (see Figure 1).



To enter the bank using such a security device, one must walk from the outside to the inside by *passing thru* the security access lock as follows :

1. **You are outside the bank**.
2. Arrive from the outside in front of the security access lock.
3. Wait until the security access lock becomes available.
4. Request access to enter the security access lock (usually by pushing a button placed on the outside of the security access lock's external door).
5. When access is granted, a condition usually signaled by a light turning green, push the security access lock's external door and enter the security access lock. Then allow the door to tightly close behind you.
6. **You are now within the *confinement* of the security access lock** : a small chamber between the two doors of the lock where your freedom of movement is greatly reduced due to the fact that this restricted “buffer zone” has *no other destination* than allowing you to *pass through*.
7. Once inside the security access lock and the external door has securely closed behind you, request access to the bank (usually by pushing a second button in front of the security access lock's internal door).
8. When access is granted, push the internal door of the security access lock, enter the bank, and, once again, allow the internal door to tightly close behind you.
9. Once inside the bank and the internal door has securely closed behind you, the security access lock becomes available for another customer to enter the bank.
10. **You are inside the bank.**

Other examples of locks include :

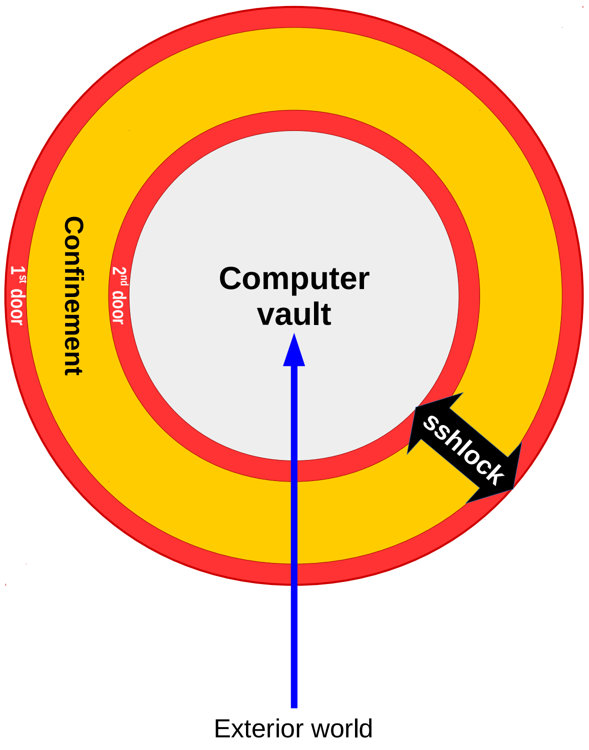
- Airlocks found in space vessels to allow passage between the pressurized environment inside the vessel and the unpressurized outer space outside of it.
- Airlocks found in submarines to allow passage between an air environment inside the submarine and the water environment outside of it.
- Hyperbaric chambers to allow entry and exit while maintaining the pressure difference with the surroundings.
- C*leanrooms* to allow entry into protected environments in which dust, dirt particles, harmful chemicals, and other contaminants are partially excluded by maintaining the inside room at a higher pressure than that of the external world.
- Locks along ships canals connecting different seas or rivers which are not at the same altitude level (e.g. Panama's Canal, Suez Canal, Canal de Bourgogne).

A virtual security lock to access computers

SSH*Lock* mocks-up the most desirable security features found in a physical security lock to create an **sshlock**, that is a *virtual* security access lock made of a highly restricted software environment (the confinement, or honey pot) as the single point of passage and pass-thru device between the outside world and the computer, secured by two tight access control mechanisms, one at each end, configured to work in series and allow passage only from the outside to the inside. The name sshlock is coined from implementing these access controls mechanisms using SSH.

More precisely :

1. An *sshlock* provides an highly restricted software environment, the *confinement* or *honey-pot*, sandwiched between two tight access mechanisms.
2. sshlock's access mechanisms work together to enforce a tight monitoring and access control at each end of the sshlock in order to prevent *verification* and resist intrusion.
3. The sshlock's confinement is so designed to be transparent to any authorized user (bees do not get trapped in their own honey) but to act as a honey-pot (a sticky jail) to trap an attacker/intruder. To that end, sshlock offers *no* facilities to figure out how to activate the sshlock's internal access mechanism to “open” the sshlock's



“internal door”.

4. At any time, one user at most is allowed inside the sshlock's confinement or inside the computer : when a user is inside either one, any other user is made to wait outside the sshlock until the authorized user has left both the computer *and* the sshlock.
5. Once a user is inside the sshlock's confinement, the external access mechanism of the sshlock becomes unavailable (its “external door” is locked).
6. Once inside the sshlock's confinement, the user must know in advance what to do to activate the internal access mechanism to open the sshlock's “internal door” and complete passing thru the sshlock. Indeed, the sshlock's restricted confinement does not allow the user to undertake anything else : one action only will work and the user must know in advance which one it is. Peeking or poking through in search of the hidden sesame will violently eject the uninformed user outside the sshlock; such a user may never want to experience this misfortune again.
7. Once a user is inside the computer, both access mechanisms of the sshlock are unavailable (its two “doors” are locked).

A straightforward implementation of SSH*Lock* can be built upon the following technologies, for their inherent strength and wide availability:

1. An hardened, highly restricted chrooted environment (jail) to mockup the physical security access lock's confinement area.
2. Two OpenSSH sshd servers serving as central access control mechanisms to mockup the security access lock's physical doors.

Chroot & Jail

A `chroot` on a Un*x operating system is an operation that changes the root of the filesystem of the process launched with the `chroot(2)` system call (or with the `chroot(8)` wrapper program). A program that runs without extra privileges in such a modified environment is guaranteed not to have access to files outside this restricted filesystem. The modified environment offered to the chrooted process and its children is commonly called a **chroot jail**.

Manually designing and building a tailored jail environment perfectly suited to the needs of the chrooted task is not for the faint of heart, and is mostly reserved to the well seasoned Un*x user, the main issues being :

1. To know precisely which resources to import into the jail (binaries, libraries, symbolic inks, configuration files, devices, etc.) to allow the task to run properly inside the jail.
2. To import only the strict minimum required by the task, for the more resources are imported into the jail, the weaker the jail may become.
3. To tighten ownership and access rights of the imported resources within the jail as much as possible, so the task still runs properly within the jail while not unnecessarily granting extra privilege.

Properly setting a chrooted environment always depends on the task to be run within, and can prove to be a time consuming operation due to the many steps of unavoidable trials and errors. However, patience and commitment will take one far, and we believe that the potential and overall benefits offered by the chroot mechanism far outweigh its complexities.

The chroot mechanism should not be confused with FreeBSD's `jail` functionality, which is a chroot on steroids aimed at providing lots of additional functionalities and more isolation than a simple `chroot`.

OpenSSH

OpenSSH (OpenBSD Secure Shell) is a free open-source software suite that implements the Secure Shell (SSH) cryptographic network protocol aimed at establishing secure communications between machines connected to an IP network. It is the premier connectivity tool for remote login with the SSH protocol, encrypting all traffic to eliminate eavesdropping, connection hijacking, and other attacks. In addition, OpenSSH provides a large suite of secure tunneling capabilities, several authentication methods, and sophisticated configuration options. [4]

OpenSSH is developed by the "OpenBSD Project" (www.openssh.org) and released under the BSD license. The latest release of OpenSSH is 7.3 (2016/08/01) as of October 2016.

Reliability, robustness, strong security and simplicity-of-use have earned OpenSSH huge popularity since its inception :

- OpenSSH alone totals nearly 90%, or more, of all available SSH implementations.
- OpenSSH is freely available for many Un*x platforms (e.g. BSD, Linux, Solaris, Mac OS X, AIX, HP-UX, Cygwin).
- OpenSSH provides strong cryptography (e.g. AES, ChaCha20, RSA, ECDSA, Ed25519).
- Encryption is started before authentication: no credentials, username, password, or any other confidential information ever travels in clear (unencrypted) over the network during a SSH session.

The OpenSSH software suite provides the following tools:

- **`ssh`**, the OpenSSH client, a substitute for telnet, rlogin, and rsh to establish secure, encrypted connections to remote machines served by a SSH server:

```
$ ssh me@my.remote
$ ssh me@my.remote 'please draw me a-sheep'
$ tar cjf - ~/project | ssh me@somewhere.else 'tar xjpf - -C ~/project'
```

- **`scp`**, a secure substitute for `rcp`:

```
$ scp -pr ~/project me@somewhere.else:project
```

- **`sftp`**, a secure substitute for `ftp`:

```
$ sftp  me@somewhere.else
```

- **`sshd`**, the OpenSSH server.

Default OpenSSH configuration

As stated earlier, password-based authentication access mechanism remains today's prevalent way of accessing computers worldwide. To accommodate this situation, OpenSSH is therefore most often configured by default to accept this weaker mechanism as a valid authentication procedure for remote login (See fig. 3).

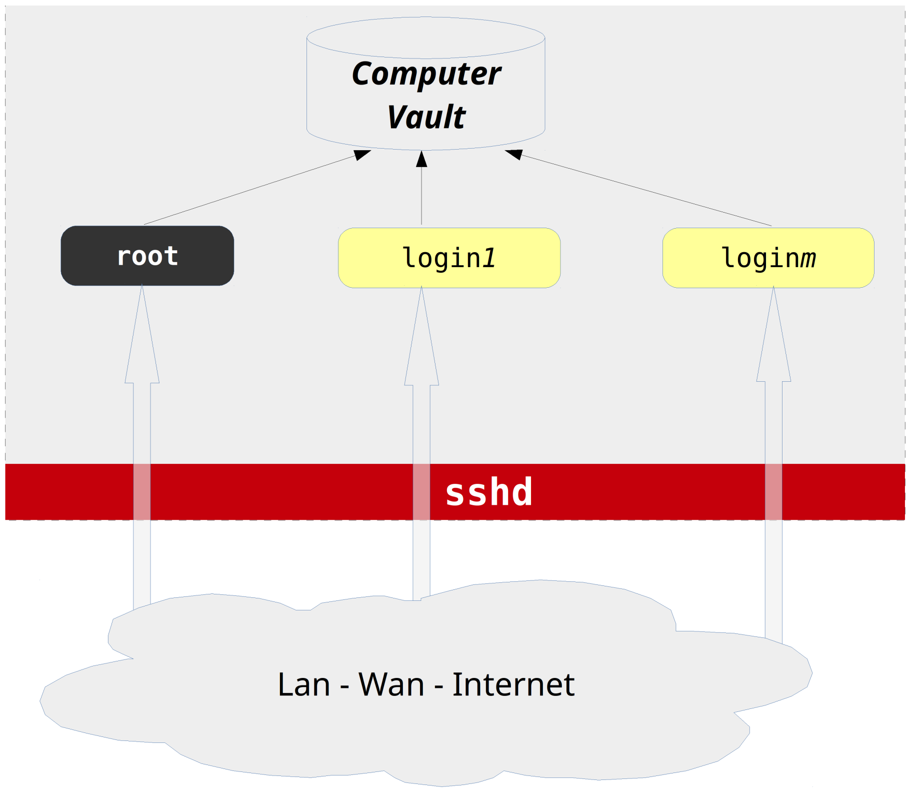


Figure 3 : SSHD default configuration

This is most unfortunate because this standard, yet insecure setup does not take advantage of OpenSSH's wealth of security features to remove the *verification* ("condition b" above) and eliminate the threat posed by password security attacks.

Single-layered access

Moreover, as a single layer access mechanism, this standard configuration of OpenSSH does not eliminate the single point of failure identified previously. The accounts of the system are now being directly exposed to the network via the `sshd` service, which acts as the only security fence between the outside world and the system : if an account is broken into, the whole system is compromised !

Putting SSH on steroids

Our implementation of SSH*Lock* relies on secure configurations of chroot and OpenSSH to bring substantial isolation and security improvements over the preceding default OpenSSH configuration.

This results in an overall secure, layered architecture whose sensitive access layer – previously build upon a SSH-based, single-layer access – is now made up of three layers : the sshlock. Together, these three layers create the strong and secure required access mechanism : an highly restricted software environment, the *confinement* (or honey-pot), sandwiched between two hardened sshd working in series to play the role of the sshlock's "protective doors", one at each end of the confinement (see fig. 4).

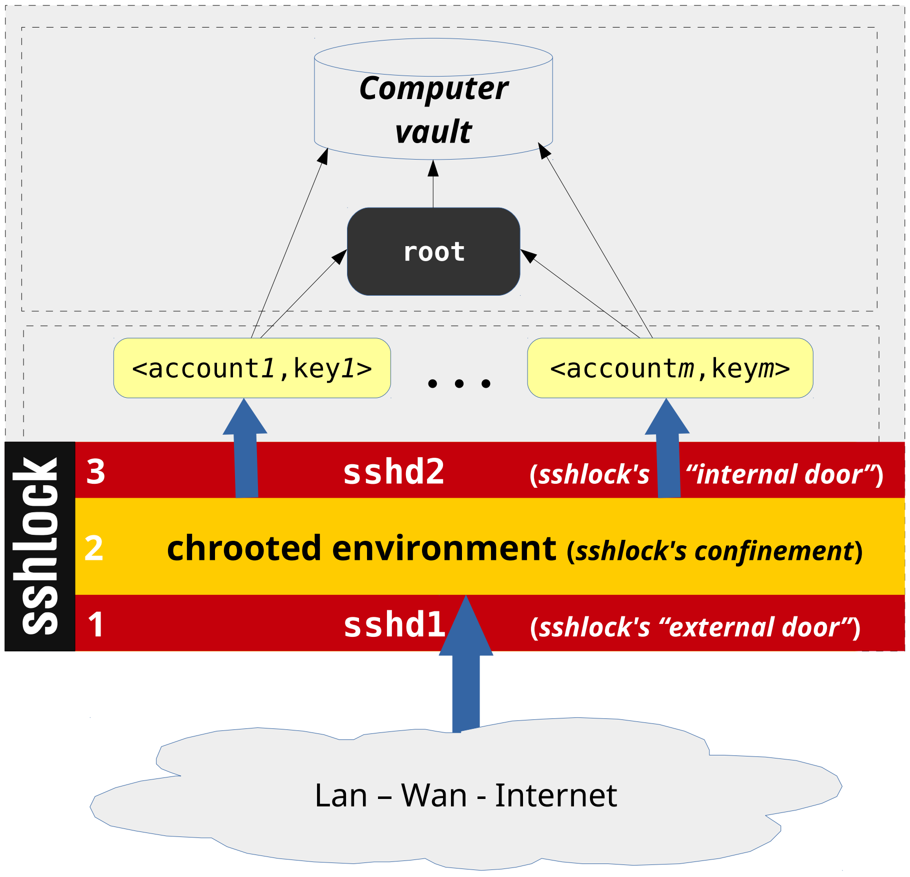

Computer vault
root
<account1,key1>
. . .
<accountm,keym>
sshlock
3
sshd2
(sshlock's "internal door")
2
chrooted environment (sshlock's confinement)
1
sshd1
(sshlock's "external door")
Lan – Wan - Internet

Figure 4 : SSH on steroids

A computer protected by a sshlock as its unique access mechanism is said to be SSHLock-protected, or *SSHLocked* for short :

- An SSHLocked system does not allow local connections through the computer's physical terminal.
- An SSHLocked system can *only* be accessed from a remote computer connecting via SSH.

sshlock's main components

An sshlock is mainly comprised of the following components :

- The **sshlock account**, to implement the sshlock's confinement/honey-pot, configured to be the single, unique point of entry into the whole system, and therefore the only system's account configured to accept incoming remote connections via sshd1. The sshlock account's directory and file structure are tightly set to provide the confinement, that is the highly restricted chrooted environment that serves either as pass-thru device to authorized users from the world on the outside to the computer's vault on the inside.

- **`sshd1`**, an OpenSSH server configured to implement the external access mechanism that controls the “opening” of sshlock's “external door”. sshd1's configuration implements a restricted *n-to-1* connection scheme set up to allow remote computers only (Lan, Wan, Internet) to connect to the local sshlock account only. sshd1 relies on strong public-key cryptography *only* for access control (password authentication is disabled). Any remote guest computer wishing to connect to the SSHLocked system must arrange to have its public key(s) installed into the sshlock account.
- **`sshd2`**, an OpenSSH server configured to implement the internal access mechanism that controls the “opening” of sshlock's “internal door”. sshd2's configuration implements a restricted *1-to-m* connection scheme set up to allow the local sshlock account only to connect to (enabled) internal local accounts *only*. The access control to local accounts from the sshlock account relies also on strong public-key cryptography, and access to local accounts (the computer's vault) is granted on a case-by-case basis by the system administrator.

SSHLock's overall configuration of OpenSSH

- Both OpenSSH services (`sshd1` and `sshd2`) are configured to work cooperatively in series :
  - At most one remote connection to the SSHLocked system may exist at any time (any second connection is denied until the first one is released).
  - sshd2 may accept a connection *only after* sshd1 has accepted one (to let the user inside the sshlock's confinement).
- `sshd1` and `sshd2` disable password-based authentication, relying instead on the strongest public key cryptography only.

SSHLock's configuration of chroot

The sshlock's confinement (the chroot configuration) is so designed to be as transparent as possible to any authorized passing-thru user (bees do not get trapped in their own honey) but to act as a honey-pot (a sticky jail) to trap an attacker/intruder. To that end, sshlock offers *no* facilities to figure out how to activate the sshlock's internal access mechanism (sshd2) to “open” the sshlock's “internal door” :

- Under the keep-it-simple principle, nothing is installed inside the chroot jail that is not absolutely necessary.
- Ownership and access rights are highly restricted, often departing from those found in standard Un*x installations.
- The sshlock account is configured to fire a no-clues, hardened restricted Bash shell (`rbash`) upon `sshd1` accepting the incoming remote connection, so that the uninformed user (most probably an attacker/intruder) would quickly cause an error (incorrect syntax, unavailable command, violation of access rights, etc.) that will immediately result in terminating the sshd1-connection and throwing the user out of the SSHLocked system.

SSHLock's overall hardening

To fully benefit from the inherent security induced by such an improved access mechanism, no weakness should remain in the system that would allow to bypass the sshlock access mechanism. The SSHLocked system must then be hardened in the following way:

- **Disable the standard password logon authentication mechanism**, so the system can only be accessed remotely from remote guest computers through the SSH-based sshlock access mechanism. On most systems, this is usually done by simply locking any valid shell account, that is preventing login via password authentication without disabling the account, for instance by rendering the encrypted password into an invalid string (usually prefixing the encrypted password string with an !).

  Under FreeBSD, disabling all shell accounts may be accomplished as follows:

```
#!/usr/bin/env bash

##
# Depending on your system, possibly check that /usr/local/bin/bash
# and /bin/bash (symlink) are included into /etc/shells
#
function list_shell_accounts () {
   perl -a -F: -n -E '
       BEGIN {%valid_shell=map{$_=>1}@ARGV;@ARGV=()}
       chomp $F[6];
       say $F[0] if exists $valid_shell{$F[6]};
   ' $(cat /etc/shells | grep -E '^/.+') < /etc/passwd
}

for login in $(list_shell_accounts)
do
    pw lock $login
done
```

  Under GNU/Linux systems, we will replace the statement “`pw lock ...`” with “`passwd -l ...`”

- As an extra security, **disable the SSHLock-system's physical console and terminals** so no manual logon is ever possible from the physical terminal attached to the computer. Under FreeBSD, this is simply done by marking “off” the 5th field of any enabled entry of the file `/etc/ttys`.

- **Disable booting the system in single mode**. Under FreeBSD, this is done by marking “insecure” the 4th field of any enabled entry of the file `/etc/ttys`, therefore forcing `init` to ask for the `root` password when the system is going to single-user mode or booted in single-user mode. Under GNU/Linux, and supposing the system is using a grub boot loader, this can be achieved by doing the following :

```
# Edit the file /etc/default/grub
$ sudo vi /etc/default/grub

# Add or enable the following line :
```

```
GRUB_DISABLE_RECOVERY="true"

# Update grub
$ sudo update-grub
```

- **Disable booting the system from the network or an external, removable rescue media**, like a CD-rom or a USB-stick, by means of a password protected BIOS/UEFI custom setup.

SSH*Lock*'s modus operandi

The overall sshlock secure access mechanism operates as follows:

1. Given there is no pending or actual SSH-connection to the SSHLocked system, upon a successful, public-key-authentication-ssh-based remote connection to the SSHLocked system's sshlock account, `sshd1` will grant access and start the remote connection (“opening” the sshlock's “external door”) by firing the hardened restricted bash in the context of the chrooted environment, allowing the user inside the sshlock's confinement.
2. As long as the `sshd1` connection lasts, `sshd1` will refuse any new connection, thus blocking any other access to the system. It should also be noted that any pending connection to the SSHLocked system (a connection being authenticated) prevents any other connection to the system to take place too.
3. Once the SSH-based access to the SSHLocked system has been granted and the user finds himself confined into the chrooted environment, very little can be done ! As with the security access lock of a bank, the user must now find the way to trigger the sshlock's internal access mechanism to “open” the sshlock's “internal door” and walk into the computer's vault. This is basically the only action the user is allowed to do! However, contrary to the security access lock of a bank, there is no visible “button” to press here : the user *must know* what to do prior to walking-in! No peeking/poking around is allowed : the confined and highly restricted chrooted nature of the sshlock's confinement, acting as a honey-pot to the uninformed user, will silently and violently throw the user outside the system upon the first operational error:
    - Most familiar commands generally available in a normal Un*x shell session are here simply forbidden. Only three commands are enabled by default : `date`, `scp`, `ssh`.
    - These commands cannot even be called directly but only through a *command proxy* whose name (“`cmdproxy`” by default), which can be freely chosen at SSHLock deployment time, must also be known in advance (obfuscation).
    - The slightest operational error while inside the sshlock's confinement (for example running a forbidden command, causing a syntax error, running against invalid credentials, changing directories or listing the content of directories) immediately results in violently aborting the current SSH session and connection.
4. Once inside the sshlock's confinement, and upon a successful public-key-authentication-ssh-based local connection to an internal local account (whose access has been previously granted), `sshd2` will grant access and start the local connection to the chosen account, thus “opening” the sshlock's “internal door” and letting the user walk, unrestricted, into the computer.

5. Like with `sshd1`, as long as the `sshd2` connection lasts, `sshd2` will refuse any new local connection from the sshlock account to any internal local account.

Setting up an sshlock

Setting up an sshlock can be broken into the following steps:

1. Create the local sshlock account.
2. Set up its chroot environment.
3. Set and harden the restricted shell.
4. Set `sshd1`.
5. Set `sshd2`.
6. Install public-keys, and grant accesses to internal local accounts.
7. Harden the computer (seen above).

Creating the sshlock account

Creating the sshlock account basically narrows down to creating a new user account : the *sshlock account*. Given that the new entry into the system-wide `/etc/passwd` must work with respect to both referentials, namely the system-wide file system, and that of the chrooted environment, special care must be taken while setting the locations of resources associated with the home directory (6th field) and the shell (7th field) associated with the sshlock entry in /etc/passwd. Say we have the following entry:

```
sshlock:*:1001:1001:SSHLOCK:/sshlock:/restricted/rbash
```

- With respect to the system-wide filesystem, the leading slash of the absolute path “`/sshlock`” refers to the root of the system-wide filesystem, and such a path should correctly refer to the absolute location of the chrooted filesystem's root within the system-wide filesystem.
- With respect to the chrooted environment, the leading slash of the absolute path “`/sshlock`” now refers to the root of the chrooted filesystem's, that is to the resource “`/sshlock/sshlock`” within the system-wide filesystem.
- In the same manner, the shell “`/restricted/rbash`” being only referred to in the context of the chroot jail, it refers to the resource “`/sshlock/restricted/rbash`” within the system-wide filesystem.

In properly setting a chroot environment, including setting relative symbolic links within the jail, this dual view of the file-system proves to be a subtle point where most newcomers seem to stumble.

Setting up the chrooted environment

Setting up the chrooted environment requires good acquaintance with the whistles and bells of the underlying system:

- Identify the file-system structure suitable to the desired task.
- Identify the absolute minimum resources which need to be imported inside the sshlock for the desired task to work reliably and consistently: which devices? Which binaries? Static of shared libraries? Which libraries (not forgetting the elf loader) ? Which configuration files ? Etc.
- Identify the strictest ownership and access rights which still allow the task to work reliably and consistently. This is ok, and even recommended, to depart from those found in the system-wide filesystem, often too lax.

The following listing shows an example of a possible layout for an sshlock chrooted environment (FreeBSD 10.3) :

```
/home/sshlock
|-- [l--x--x--x root     wheel   ]  .ssh -> ./home/sshlock/.ssh/
|-- [d--x--x--x root     wheel   ]  all/
|   |-- [l--x--x--x root     wheel   ]  date -> /rescue/date*
|   |-- [l--x--x--x root     wheel   ]  scp -> /usr/bin/scp*
|   |-- [l--x--x--x root     wheel   ]  ssh -> /usr/bin/ssh*
|-- [dr-xr-xr-x root     wheel   ]  dev/
|-- [d--x--x--x root     wheel   ]  etc/
|   |-- [-r--r--r-- root     wheel   ]  group
|   |-- [-r--r--r-- root     wheel   ]  host.conf
|   |-- [-r--r--r-- root     wheel   ]  hosts
|   |-- [-r--r--r-- root     wheel   ]  inputrc
|   |-- [-r--r--r-- root     wheel   ]  libmap.conf
|   |-- [-r-------- root     wheel   ]  master.passwd
|   |-- [-r--r--r-- root     wheel   ]  nsswitch.conf
|   |-- [-r--r--r-- root     wheel   ]  passwd
|   |-- [-r--r--r-- root     wheel   ]  profile
|   |-- [-r--r--r-- root     wheel   ]  protocols
|   |-- [-r--r--r-- root     wheel   ]  pwd.db
|   |-- [-r--r--r-- root     wheel   ]  services
|   |-- [-r-------- root     wheel   ]  spwd.db
|   `-- [d--x--x--x root     wheel   ]  ssh/
|       |-- [-r--r--r-- root     wheel   ]  ssh_config
|       |-- [-r-------- root     wheel   ]  ssh_host_ed25519_key
|       |-- [-r-------- root     wheel   ]  ssh_host_ed25519_key.pub
|       |-- [-r-------- root     wheel   ]  sshd1_config
|       |-- [-r-------- root     wheel   ]  sshd2_accounts_enabled
|       |-- [-r-------- root     wheel   ]  sshd2_config
|       `-- [-r-------- root     wheel   ]  sshd_banner.legal
|-- [d--x--x--x root     wheel   ]  granted/
|   |-- [l--x--x--x root     wheel   ]  cmdroxy -> /restricted/cmdproxy
|   `-- [l--x--x--x root     wheel   ]  scp -> /all/scp
|-- [d--x--x--x root     wheel   ]  home/
|   `-- [drwx------ sshlock  sshlock ]  sshlock/
|       |-- [-r--r--rw- root     wheel   ]  .bash_history
|       |-- [l--x--x--x root     wheel   ]  .inputrc -> /etc/inputrc
|       |-- [d--x------ sshlock  sshlock ]  .ssh/
|           `-- [-r-------- sshlock  sshlock ]  authorized_keys
|-- [d--x--x--x root     wheel   ]  lib/
|   |-- [-r--r--r-- root     wheel   ]  libc.so.7
|   |-- [-r--r--r-- root     wheel   ]  libcrypt.so.5
|   |-- [-r--r--r-- root     wheel   ]  libcrypto.so.7
|   |-- [-r--r--r-- root     wheel   ]  libmd.so.6
|   |-- [-r--r--r-- root     wheel   ]  libncurses.so.8
|   |-- [-r--r--r-- root     wheel   ]  libthr.so.3
```

```
46   |   |-- [-r--r--r-- root     wheel   ]  libutil.so.9
47   |   `-- [-r--r--r-- root     wheel   ]  libz.so.6
48   |-- [d--x--x--x root     wheel   ]  libexec/
49   |   `-- [---x--x--x root     wheel   ]  ld-elf.so.1*
50   |-- [dr-xr-xr-x root     wheel   ]  proc/
51   |-- [d--x--x--x root     wheel   ]  rescue/
52   |   |-- [---x--x--x root     wheel   ]  date*
53   |-- [d--x--x--x root     wheel   ]  restricted/
54   |   |-- [---x--x--x root     wheel   ]  bash*
55   |   |-- [---x--x--x root     wheel   ]  cmdproxy*
56   |   `-- [l--x--x--x root     wheel   ]  rbash -> bash*
57   |-- [d--x--x--x root     wheel   ]  usr/
58   |   |-- [d--x--x--x root     wheel   ]  bin/
59   |   |   |-- [---x--x--x root     wheel   ]  scp*
60   |   |   `-- [---x--x--x root     wheel   ]  ssh*
61   |   |-- [d--x--x--x root     wheel   ]  lib/
62   |   |   |-- [-r--r--r-- root     wheel   ]  libasn1.so.11
63   |   |   |-- [-r--r--r-- root     wheel   ]  libcom_err.so.5
64   |   |   |-- [-r--r--r-- root     wheel   ]  libgssapi.so.10
65   |   |   |-- [-r--r--r-- root     wheel   ]  libheimbase.so.11
66   |   |   |-- [-r--r--r-- root     wheel   ]  libhx509.so.11
67   |   |   |-- [-r--r--r-- root     wheel   ]  libkrb5.so.11
68   |   |   |-- [-r--r--r-- root     wheel   ]  libroken.so.11
69   |   |   |-- [-r--r--r-- root     wheel   ]  libwind.so.11
70   |   |   `-- [d--x--x--x root     wheel   ]  private/
71   |   |       |-- [-r--r--r-- root     wheel   ]  libheimipcc.so.11
72   |   |       |-- [-r--r--r-- root     wheel   ]  libldns.so.5
73   |   |       `-- [-r--r--r-- root     wheel   ]  libssh.so.5
74   |   |-- [d--x--x--x root     wheel   ]  libexec/
75   |   |   `-- [l--x--x--x root     wheel   ]  ld-elf.so.1 -> /libexec/ld-
elf.so.1*
76   |   `-- [d--x--x--x root     wheel   ]  local/
77   |       `-- [d--x--x--x root     wheel   ]  lib/
78   |           |-- [l--x--x--x root     wheel   ]  libintl.so.8 ->
libintl.so.8.1.5
79   |           `-- [-r--r--r-- root     wheel   ]  libintl.so.8.1.5
80   `-- [d--x--x--x root     wheel   ]  var/
81       `-- [d--x--x--x root     wheel   ]  run/
82           |-- [-r--r--r-- root     wheel   ]  ld-elf.so.hints
83           `-- [-r--r--r-- root     wheel   ]  ld-elf32.so.hints
84
85   23 directories, 60 files
```

sshlock's confinement (home directory)

Lines 33-38, where the user lands upon being granted access by `sshd1`.

Ownership

Apart from resources line 33, 37, 38, imposed by OpenSSH, no resource belongs to the sshlock user.

Access rights

- **Directories**

  Almost all directories have the 0111 mode (traverse permission only), to prevent the sshlock user from glancing at their content. Some notable exceptions for :

- The sshlock account's home directory within the jail (line 34), which requires the owner's write permission (mode 0700) to allow resources between the outside world and the computer's vault to transit via the sshlock's confinement using `scp` (imagine going to the bank to replenish your account : you must be permitted to carry the cash along with you while transiting through the security lock !).
- `/dev` (line 7) and `/proc` (line 50) resources, which require the read flag (mode 0555).
- The `.ssh` directory of the sshlock account (line 37), which also requires the read flag (mode 0500).

- **Executable binaries**

  All executable binaries have the mode 0111 (executable permission only), including the statically compiled elf loader, not to be forgotten (line 49).

- **Binary libraries**

  All binary libraries have the 0444 mode (read permission only).

- **Text files**

  Configuration files have the mode 0444 or 0400 (read permission only - lines 9-21, 23-29, 35, 36, 38).

- **Exceptions**

  `.bash_history` (line 35) is the only regular file with the write bit set, to allow recording the sshlock user's activity. However, this file belonging to root, it cannot be removed nor rewritten by the sshlock user who does not have the right to use any shell redirection mechanism, and therefore cannot invoke any command like '`date > .bash_history`'.

Setting up the restricted shell

Creating a chroot jail associated to the sshlock account is only the first step into crafting a highly confined and restricted environment. The second step consists in providing the sshlock account with a restricted shell like `rbash` (see `bash(1)`) as the standard shell to be fired upon connection. This is enforced in the /etc/passwd configuration file seen in the example above.

A restricted shell is used to set up an environment more controlled than the standard shell. Restricted Bash behaves identically to Bash with the exception that the following are disallowed, forbidden or not performed, generating an error instead:

- Changing directories with the **`cd`** builtin command.
- Setting or unsetting the values of some important environment variables like **`SHELL`**, **`PATH`**, **`ENV`**, or **`BASH_ENV`**.
- Specifying command names containing the slash character (“/”).
- Specifying a filename containing a “/” as an argument to the **`.`** or **`source`** builtin commands.
- Specifying a filename containing a slash as an argument to the **`-p`** option to the **`hash`** builtin command.

- Importing function definitions from the shell environment at startup.
- Parsing the value of **SHELLOPTS** from the shell environment at startup.
- Redirecting output using the >, >|, <>, >&, &>, and >> redirection operators.
- Using the **exec** builtin command to replace the shell with another command.
- Adding or deleting builtin commands with the **-f** and **-d** options to the **enable** builtin command.
- Using the **enable** builtin command to enable disabled shell builtins.
- Specifying the **-p** option to the **command** builtin command.
- Turning off restricted mode with **set +r** or **set +o restricted**.

Hardening the restricted shell

We will harden that tight configuration even more by crafting a special rbash startup file made to enforce extra restrictions, the most important being to restrict the PATH and to exit on the first error :

```
#
# /etc/profile - rbash startup file (excerpts)
#

# Restrict commands
export PATH=/granted

# Exit on first command returning a non zero status (error)
set -e
```

Given PATH is restricted to `/granted` (relatively to the chroot) and the user cannot use "/" in its commands or arguments, the only available command is **cmdproxy**. Running any other command will result in an error that will immediately terminate the SSH session. This mandatory command will in turn launch any available commands not directly accessible (`date`, `scp`, `ssh`).

One may wonder whether all that seeming obfuscation is needed and/or useful. For instance, why not copy the commands under /all, instead of having links? The answer is not theoretical nor rhetorical, simply pragmatical. Never knowing what these binaries actually do and whether they may rely internally on other commands, the simplest solution is to mockup the original resources subtree as we install them inside the chroot jail. For instance, take the case of scp, which internally launches /usr/bin/ssh. If ssh were not to be where it is expected, that is /usr/bin/ssh, then scp would fail as a result.

As for the special directory "`/restricted`", it is a particular container used to lock some necessary commands which we voluntarily do not make available to the sshlock user. For instance, though the bash shell must be available within the chroot jail to be launched upon a valid sshd1 connection, we do not want the sshlock user to have the option to launch it at whim as a subshell, which could lock him out of the restricted shell.

Setting up OpenSSH

Properly setting up OpenSSH is central to the security of SSH*Lock* overall operation.

Security and cryptography

Under the “keep it simple” principle, we will disable any configuration option we will not be using, and will only resort to the strongest cryptography options.

Protocol: we disable the SSH-1 protocol altogether.

Key exchange: given recent revelations from ex-consultant at NSA Edward Snowden that NSA willingly inserts backdoors into softwares, hardware components and published standards, some researchers (including Bruce Schneier and Dan J. Bernstein) have expressed their lack of confidence in NIST-published curves such as nistp256, nistp384, nistp521 in which it is believed that NSA had a word to say in their definition, and for which constant parameters, including the generator point, are defined without explanation. These curves are not the most secure or fastest possible for their key sizes, and researchers think it is possible that NSA have ways of cracking NIST curves. It is also interesting to note that SSH belongs to the list of protocols the NSA claims to be able to eavesdrop. Having a secure replacement would make passive attacks much harder if such a backdoor exists.

As such, we will be using the high-security Curve25519 Elliptic curve Diffie-Hellman algorithm proposed in 2006 by Dan J. Bernstein, its main strengths being its record-setting speeds, its constant-time run time and resistance against side-channel attacks, and its lack of nebulous hard-coded constants.

Key signing (server authentication): we will be using the super fast, super secure Ed25519 Edwards curve Digital Signature Algorithm (Ed25519) public-key signature system designed by Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang, a digital signature scheme using a variant of the Schnorr signature based on Twisted Edwards curves, designed to be faster than existing digital signature schemes without sacrificing security.

Client/User authentication: as explained in the introduction, we will disable password authentication and resort to using public-key authentication only, where we will enforce key-pairs to be protected by strong passphrases, knowing that nothing secret will ever be exchanged over the network.

Confidentiality: symmetric ciphers are used to encrypt the data after the initial key exchange and authentication have been completed. We will be using the newer Bernstein's ChaCha20 symmetric cipher for Internet confidentiality, standardized in RFC 7905.

sshd1

sshd1, an OpenSSH server, is sshlock's external access mechanism configured to implement SSH*Lock*'s “external door”, that is for monitoring and controlling access into the sshlock's confinement from remote computers only (Lan, Wan, Internet):

- Public-key authentication only.
- Only listen to <u>incoming remote connections</u>; do not answer incoming local connections.

- Only accept connections to the local sshlock account; connections to any other account are prohibited.
- Upon a successful connection, lock the user into the confined, restricted environment associated to the sshlock account (hardened chroot jail).
- Do not allow more than one connection at any time, actual or pending.

```
#
# SSHD1_CONFIG EXCERPTS
#

# Listen networks for incoming remote connections only
ListenAddress       192.168.1.12:22000

# Disable the SSH-1 protocol altogether
Protocol            2

# CLIENT/USER AUTHENTICATION
PubkeyAuthentication            yes
PasswordAuthentication          no

# ACCESS CONTROL
MaxAuthTries    1

# Deny local connections (IP addresses of the network interfaces)
DenyUsers       *@192.168.1.12
DenyUsers       *@127.0.0.1

# Incoming Remote Connections to the local sshlock account only
AllowUsers      sshlock@*

# Allow at most 1 simultaneaous sshlock session at any time.
MaxSessions     1

# Allow 1 pending unauthenticated connection at any time
MaxStartups     "1:100:1"

# Lock the incoming user into the sshlock jail (OpenSSH 4.9p1 or later)
ChrootDirectory /home/sshlock
```

sshd2

sshd2, an OpenSSH server, is sshlock's internal access mechanism to implement SSH*Lock*'s “internal door”, that is for controlling and monitoring access into the computer's vault via internal local accounts:

- Public-key authentication only.
- Only listen to incoming local connections; do not answer incoming remote connections.
- Only accept connections to internal accounts that have granted access to the sshlock account.
- Do not allow more than one connection at any time, actual or pending.

sshd1 and sshd2 play complementary roles and their configuration are very similar:

```
#
# SSHD2_CONFIG EXCERPTS
#

ListenAddress    localhost:22000

AllowUsers       micha@localhost
AllowUsers       wolfy@localhost
AllowUsers       baltamos@localhost
AllowUsers       scarlette@localhost
AllowUsers       bella@localhost
AllowUsers       sultan@localhost
AllowUsers       nell@localhost
AllowUsers       cesar@localhost
AllowUsers       junior@localhost
AllowUsers       jo@localhost
AllowUsers       marius@localhost
AllowUsers       ptitloup@localhost
AllowUsers       douce@localhost
AllowUsers       snowflower@localhost
```

Default `ssh_config`

Used by the sshlock account as a generic template to connect to any enabled internal local account via sshd2:

```
Host localhost
    Ciphers                          chacha20-poly1305@openssh.com
    ClearAllForwardings              yes
    KexAlgorithms                    curve25519-sha256@libssh.org
    MACs                             hmac-sha2-512-etm@openssh.com
    PasswordAuthentication           no
    Port                             22000
    PreferredAuthentications         publickey
    Protocol                         2
    PubkeyAuthentication             yes
    RSAAuthentication                no
    SendEnv                          LANG LANGUAGE LC_*
    VisualHostKey                    yes
```

SSHLock's management of public-keys and access to internal local accounts

Passing-thru the sshlock is done in two steps:

1. Connecting from a remote guest client to the SSHLocked-system's sshlock account via sshd1. This requires that any remote guest client wishing to connect to the SSHLocked-system has its ED25519 public key installed into the sshlock account of the SSHLocked-system.
2. Connect from the SSHLocked-system's sshlock account to the final local destination (an internal account) via sshd2. This requires that the sshlock account stores ED25519 key-pairs for each authorized user, and that the public-key counterpart of these keys are installed on a case-by-case basis into the local accounts that are granting access to the sshlock account.

Identities

The purpose of SSH*Lock* is to accommodate any authorized user to pass-thru the sshlock to eventually connect to an authorized local account. As such, SSH*Lock* must allow to locally and securely store the ED25519 cryptographic key-pairs of any authorized user in order to let them access their final destination account within the SSHLocked-system via sshd2.

Identities is SSH*Lock*'s local identification mechanism by which authorized users are uniquely identified within the system, and by which their cryptographic keys are uniquely associated to them within the sshlock's confinement, to be further exported into any local account granting access.

Using the SSHLock software

Setting up an sshlock manually is not trivial; it is time consuming and error prone. Because no existing open source or commercial software met our requirements when we urgently needed a strong, reliable and perennial solution, we wrote SSHLock, the software that closely implements our SSH*Lock* specifications. Over the years, SSHLock has become a steady product one can trust to deliver the following advantages:

- To the best of our knowledge, SSHLock is the only software of its kind that provides this secure access mechanism to computers in order to defeat password attacks and render them virtually impenetrable to attackers.
- SSHLock provides many useful options to create, obfuscate and administer an sshlock, saving weeks of anguish and headache to the savvy system administrator who only wants the job (well) done.
- SSHLock provides the end user with a simple interface that makes it a breeze to remotely connect to SSHLocked-system.

Supported platforms

In its current version (1.21 as of November 2016), SSHLock primarily targets FreeBSD systems (FreeBSD, PC-BSD, FreeNAS, ...) and Debian GNU/Linux software distributions (Ubuntu family, Mint, Knoppix). However, people are welcome and encouraged to request that we port this software to other platforms.

Where to get the SSHLock software

SSHLock can be freely download by visiting the page:

`http://www.franckys.com/products/sshlock/index.html`

Software requirement

SSHLock will work best on FreeBSD and GNU/Linux platforms fairly up to date. SSHLock will not work with versions of OpenSSH prior to 4.9p1 (we need the ChrootDirectory option), and versions of Bash prior to 4.0 (we use some Bash advanced features :( ).

Obfuscation

SSHLock provides safe defaults out of the box for a number of parameters :

Parameter	Option	Default value
Name of the sshlock account	`-u`	**sshlock**
root of the chroot filesystem	`-r`	**/home/sshlock***
Non standard SSH Port	`-p`	**22000**
Name of the Command-proxy	`-w`	**cmdproxy**

(*) Set by the system

However, customizing these parameters on a case-by-case basis is *highly* recommended to make your own SSHLock deployment less predictable and therefore more secure.

SSHLock main operations

Using SSHLock is best explained by walking through a real session. Please be aware that SSHLock requires root privileges, and that to stay on the safe side, it is recommended to keep a console open at all times on the host system while initially installing and running the software.

Here is a typical scenario: say Franck is installing SSHLock on his server at work (freebsd.at.work – in blue below) so he can safely administer it remotely from home (freebsd.at.home – in dark yellow below).

```
#
#  freebsd.at.home (remote guest client)
#

# List the SSHLocked-systems I am already administering from home
[franck@freebsd.at.home: ~]$ cd ~/sshlock; ls
ai.pf/   franckys.com/   ipbx.sv.ca/   webserv.sv.ca/

# Create a new entry for my freebsd server at work :
[franck@freebsd.at.home: ~/sshlock]$ mkdir freebsd.at.work; cd freebsd.at.work
```

```
#
# freebsd.at.work  (sshlock service)
#

# Check my SSHLock kit
[franck@freebsd.at.work: ~]$ cd ~sshlock ; ls
sshlock     cmdproxy

# Create an sshlock with “name:mysshlock”, “chrootfs:/mysshlock”, “port:22022”,
# “cmdproxy:please”, “machine name:freebsd.at.work”
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -u mysshlock -r /mysshlock
-p 22022 -w please -N freebsd.at.work -C
==
== sshlock - Version 1.21 -- lundi 24 octobre 2016, 04:55:09 (UTC+0100) ==
==
== System: freebsd
== Root:   root
== Wheel:  wheel
== Checking environment ==
...
== Creating sshlock account ==
...
>>> ------------
>>> Please find "sshlock-keygen", a script to generate ED25519 cryptographic
>>> key-pairs you need on your guest client machines to connect to any
>>> SSHLocked-system.
>>>
>>> Please find "sshlock-connect-freebsd.at.work", a script to connect to
>>>  "freebsd.at.work" (public DNS name) from any guest client machine.
>>>
>>> Please export both scripts to any machine you wish to use as guest clients
>>> to connect to the SSHLocked-system at "freebsd.at.work",and learn how to use
>>> them:
>>>
>>> 1) Generation of ED25519 cryptographic key-pairs on your guest client
>>>    machine
>>>     $ sshlock-keygen -h
>>> 2) Export and install your keys on freebsd.at.work (see helps)
>>> 3) Connection to freebsd.at.work :
>>>    $ sshlock-connect-freebsd.at.work -h

# Display the resulting sshlock current configuration (stored by default in file
# /etc/sshlock.conf)
[franck@freebsd.at.work: ~/sshlock]$ sudo cat /etc/sshlock.conf
CONFIG_SSHLOCK_DNSNAME="freebsd.at.work"
CONFIG_SSHLOCK_CONFIGFILE="/etc/sshlock.conf"
CONFIG_SSHLOCK_USERNAME="mysshlock"
CONFIG_SSHLOCK_HOMEDIR="/mysshlock"
CONFIG_SSHLOCK_SSHD_PORT="22022"
CONFIG_SSHLOCK_PROXY_WRAPPER="please"
CONFIG_SSHLOCK_CLIENT_SSHCONFIG="sshlock-connect"
CONFIG_SSHLOCK_ACTION='help'

# Follow the instructions above and forward the 2 scripts to my guest machine
# freebsd.at.home
[franck@freebsd.at.work: ~/sshlock]$ ls
sshlock     sshlock-keygen     cmdproxy     sshlock-connect-freebsd.at.work

[franck@freebsd.at.work: ~/sshlock]$ scp sshlock-keygen sshlock-connect-
freebsd.at.work franck@freebsd.at.home:sshlock/freebsd.at.work
Password for franck@freebsd.at.home:
sshlock-keygen                          100% 2846    2.8KB/s   00:00 ETA
sshlock-connect-freebsd.at.work         100% 3653    3.6KB/s   00:00 ETA
```

```
#
#  freebsd.at.home (client)
#

# Check that I received the 2 scripts from franck@freebsd.at.work
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ls -l
total 12
-r-xr-xr-x  1 franck  franck    3653 Nov 21 20:41 sshlock-connect-freebsd.at.work
-r-xr-xr-x  1 franck  franck    2846 Nov 21 20:41 sshlock-keygen

# Create an ed25519 key-pair that I will use to connect to my sshlocked-system
# at freebsd.at.work
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ./sshlock-keygen -i
franck@freebsd.at.home  -P 'Dumb!Pa55phrase'
>>>
>>> Your new key pair is now available at :
```

```
>>>
>>>    ~/.ssh/ed25519.franck@freebsd.at.home   (private key)
>>>    ~/.ssh/ed25519.franck@freebsd.at.home.pub    (public  key)
>>>
>>> You now need to import this public key to your remote SSHLocked-system
>>>
>>> 1) Forward the file: [~/.ssh/ed25519.franck@freebsd.at.home.pub] to the
>>>    sysadmin of the sshlocked-system you wish to connect into, along with
>>>    the dentity you wish to use.
>>>
>>> 2) The sysadmin of the remote SSHLocked-system will run the following command
>>>    to install your key and grant you access into the system :
>>>    $ sshlock -i "franck@freebsd.at.home" -K ed25519.franck@freebsd.at.home.pub
>>>
>>> When this is done, you will be able to connect to the remote SSHLocked-system
>>> using the command :
>>>    $ sshlock-connect-freebsd.at.work franck@freebsd.at.home
>>>

# Let's check my new ed25519 key-pair
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ls
authorized_keys
ed25519.franck@ai.pf                    ed25519.franck@ai.pf.pub
ed25519.franck@franckys.com             ed25519.franck@franckys.com.pub
ed25519.franck@freebsd.at.home          ed25519.franck@freebsd.at.home.pub
ed25519.franck@ipbx.sv.ca               ed25519.franck@ipbx.sv.ca.pub
ed25519.franck@webserv.sv.ca            ed25519.franck@webserv.sv.ca.pub
known_hosts

# Follow the instruction: export my new public key to my sshlocked-system at work
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ scp
~/.ssh/ed25519.franck@freebsd.at.home.pub franck@freebsd.at.work:sshlock
Password for franck@freebsd.at.work:
ed25519.franck@freebsd.at.home.pub           100%   101    0.1KB/s   00:00 ETA
```

```
#
# freebsd.at.work  (sshlock service)
#

# Check that I received the franck@freebsd.at.home's public key
[franck@freebsd.at.work: ~/sshlock]$ ls
ed25519.franck@freebsd.at.home.pub      sshlock-connect-freebsd.at.work
sshlock                  sshlock-keygen            cmdproxy

# Install that public key under the identity 'franck@freebsd.at.home'
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -i franck@freebsd.at.home -K
ed25519.franck@freebsd.at.home.pub
> Imported public-key: [ed25519.franck@freebsd.at.home.pub] for identity:
[franck@freebsd.at.home] successfully installed!
1 imported public-keys were successfully installed in sshlock

# Grant identity 'franck@freebsd.at.home' access to my local account 'franck'
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -i franck@freebsd.at.home -k
franck -P 'An0ther!Dumb&Passphra53'
Generating an ed25519 key pair for identity: [franck@freebsd.at.home]
> Access to local account: [franck] was successfully granted to identity:
[franck@freebsd.at.home].
1 local accounts/identityies were enabled
Configuring/reconfiguring OpenSSH sshd2 server

# Review allgranted access rights
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -l
```

```
== Identities authorized to connect to this computer via sshlock ==
franck@freebsd.at.home: ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIKMh8VdDI50WIvEOmkeWNuu0yxFH+qh5LcU/D9PKxkYW
franck@freebsd.at.home
1 accesses are currently granted to this computer via sshlock
== Accounts accepting local connections from the sshlock ==
franck enabled for: franck@freebsd.at.home
1 accesses are currently granted to local accounts

# Everything seems ok : start the sshlock service
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -S
== Enabling sshlock ==
Mounting volume: [/mysshlock/dev]
Mounting volume: [/mysshlock/proc]
Stopping sshd daemons: [674]
SSHLock service is running: [7972 7974]
```

```
#
#  freebsd.at.home (client)
#

# Let's manually connect to my new SSHLocked-system freebsd.at.work
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ./sshlock-connect-
freebsd.at.work -i franck@freebsd.at.home

Legal notice
------------
Any fraudulent or non authorized connexion to this
computer equipment by any means will be prosecuted
according to the law.

Information légale
------------------
L'auteur de connexions frauduleuses ou non autorisées
à cet équipement informatique, par quel que moyen que
ce soit,  sera poursuivi  judiciairement selon la loi
en vigueur au moment de l'infraction.

Enter passphrase for key '/home/franck/.ssh/ed25519.franck@freebsd.at.home':

Last login: Mon Nov 21 22:28:25 2016 from 192.168.1.1

# I am in. It works! I just passed the sshlock's “external door” and I'm now
# within the sshlock's very restricted confinement.
(session monitored) 06:32:08 mysshlock@freebsd.at.work [~] $

# Be cautious man... see if I can make it thru to my final destination !
(session monitored) 06:32:08 mysshlock@freebsd.at.work [~] $ please ssh -i
~/.ssh/franck@freebsd.at.home franck@localhost

The authenticity of host '[localhost]:22022 ([127.0.0.1]:22022)' can't be
established.
ED25519 key fingerprint is SHA256:rchWOsE+97/JADu6sTh+bqTORdbj5olMVv3dO8OUvH0.
+--[ED25519 256]--+
|             |
|             |
|             |
|     . . o   |
|      = S o   . .|
|     =.B = . . = |
|     o& B . . + +|
|   .oBo@ + o . =E|
|   o==B.o ..=. .+|
+----[SHA256]-----+
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? Yes

Legal notice
------------
Any fraudulent or non authorized connexion to this
computer equipment by any means will be prosecuted
according to the law.

Information légale
------------------
L'auteur de connexions frauduleuses ou non autorisées
à cet équipement informatique, par quel que moyen que
ce soit,  sera poursuivi  judiciairement selon la loi
en vigueur au moment de l'infraction.

Enter passphrase for key '/mysshlock/.ssh/franck@freebsd.at.home':

Last login: Mon Nov 21 19:45:17 2016 from 192.168.1.12
To see the last 10 lines of a long file, use "tail filename". To see the
first 10 lines, use "head filename".
-- Dru <genesis@istar.ca>

# It worked!  I'm now remotely connected to franck@freebsd.at.work frome home !
[franck@freebsd.at.work: ~]$ cd sshlock; sudo ./sshlock -s
SSHLock service is running: [7972 7974]

# I can now administer my new sshlock system from home, let's harden it :)
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -FH
== Hardening the sshlock ==
     > Locking accounts to prevent console logon :
             Account: [root] now locked
             Account: [franck] now locked
             Account: [wolfy] now locked
     > Making booting in single mode require root password

# Let's add some command to the sshlock's confinement (not recommended) :
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -B whoami,ls
== Adding resource : binary ==
Binary: [whoami]
Static version: [/rescue/whoami]
Binary's final source: [/rescue/whoami]
Destination: [/mysshlock//rescue/whoami]
== Adding resource : binary ==
Binary: [ls]
Static version: [/rescue/ls]
Binary's final source: [/rescue/ls]
Destination: [/mysshlock//rescue/ls]

# Leave the vault and go back to the sshlock's confinement...
[franck@freebsd.at.work: ~/sshlock]$ exit
logout
Connection to localhost closed.

# I'm back to the sshlock's confinement, with 2 new commands at my fingertip ;)
(session monitored) 06:43:11 mysshlock@freebsd.at.work [~] $ please whoami
mysshlock

# (I don't have the right to list the content of my ~/.ssh)
# Let's generate an error and witness first hand the violent eject !
(session monitored) 06:43:24 mysshlock@freebsd.at.work [~] $ please ls .ssh
ls: .ssh: Permission denied
Connection to freebsd.at.work closed.
```

```
# Back home, it all worked, let's have a cold beer :)
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$
```

```
#
# freebsd.at.work  (sshlock service)
#

# Everything works. Let's deploy sshlock as an autostart service
[franck@freebsd.at.work: ~/sshlock]$ sudo ./sshlock -I
Autostart service 'sshlock' successfully installed!

# Let's see...
[franck@freebsd.at.work: ~/sshlock]$ sudo /etc/rc.d/sshlock status
SSHLock service is running: [7972 7974]
```

Word of caution

Creating multiple sshlocks within the same machine

SSHLock does preclude creating multiple sshlocks within a given machine. However, it should be mentioned than doing so does not increase the security. In such a situation, special care should be taken to insure the following:

- The multiple sshlocks should not compete for the same ports (option -p).
- The multiple sshlocks should use different sshlock names (option -u) and different roots (the best is to leave that blank and let the system use safe defaults).
- The configuration files do not compete for the same location (option -f).
- Only one can be installed by default as an autostart service (they compete for the same system files).

Adding more commands to the base

Though adding more commands to the chrooted environment is possible through options -B and -b, this is never necessary since the only intended purpose of the sshlock is to pass thru the sshlock's confinement.

Please be aware that the more commands are added to the sshlock base, the weaker it potentially may become.

Not using passphrase to protect your ed25519 keys

SSHLock only runs on the machine that is to be protected, not on the remote clients. Therefore, even though the “`sshlock-connect-<host>`” scripts do their best, SSHLock cannot enforce the remote user to use strong passphrases to protect the generated ed25519 key-pairs.

Not using strong passphrases is strongly discouraged, as anyone getting your keys will be able to impersonate you and successfully connect to the first stage of the SSHLocked-system. Though this person will likely not go too far within the sshlock's confinement, you will still have to prove that was not you!

PASS-THRU mode

The command “`sshlock-connect-<host>`” generated by SSHLock during the creation of an sshlock on a host system, and to be exported to all remote guest clients wishing to connect to this SSHLocked host, can be used in several ways. One of them is the *pass-thru* mode, which allows the user to connect directly to his final local account on the SSHLocked-system without dealing with all the complexities involved when passing manually through the sshlock, including the highly restricted sshlock's confinement.

For instance, say that we are using the identity `franck@freebsd.at.home` (as seen above during our little scenario) to connect directly to the account `franck` on the SSHLocked-system `freebsd.at.work` using this script. It suffices to enter the following command, followed by the two required passphrases in sequence, and we are all done:

```
#
#  freebsd.at.home (client)
#

# Let's automatically connect to my final destination on my new sshlocked
# system freebsd.at.work
# Notice the addition of the option -p (pass-thru)
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ./sshlock-connect-
freebsd.at.work -i franck@freebsd.at.home -p

Legal notice
------------
Any fraudulent or non authorized connexion to this
computer equipment by any means will be prosecuted
according to the law.

Information légale
------------------
L'auteur de connexions frauduleuses ou non autorisées
à cet équipement informatique, par quel que moyen que
ce soit,  sera poursuivi  judiciairement selon la loi
en vigueur au moment de l'infraction.

Enter passphrase for key '/home/franck/.ssh/ed25519.franck@freebsd.at.home':

bash: warning: setlocale: LC_ALL: cannot change locale (fr_FR.UTF-8): No such
file or directory
The authenticity of host '[localhost]:22022 ([127.0.0.1]:22022)' can't be
established.
ED25519 key fingerprint is SHA256:pj+7LrAGfTN08DNZa5vgEWO/kZZV+p+Tgpg7hRTmJRg.
+--[ED25519 256]--+
|       Eo    ..  |
|      ..++....   |
|       +o*+=.    |
|      . OoO  .   |
|   . . oSB.=  .  |
|  . o +o..*..  .o|
|   . +.o o.. . +.|
|    o .....   . .|
|   .   o==.      |
+----[SHA256]-----+
No matching host key fingerprint found in DNS.
```

```
Are you sure you want to continue connecting (yes/no)? Yes
Failed to add the host to the list of known hosts
(/home/sshlock/.ssh/known_hosts).

Legal notice
------------
Any fraudulent or non authorized connexion to this
computer equipment by any means will be prosecuted
according to the law.

Information légale
------------------
L'auteur de connexions frauduleuses ou non autorisées
à cet équipement informatique, par quel que moyen que
ce soit,  sera poursuivi  judiciairement selon la loi
en vigueur au moment de l'infraction.

Enter passphrase for key '/home/sshlock/.ssh/franck@freebsd.at.home':

Last login: Wed Nov 23 05:02:15 2016 from 127.0.0.1
The Moon is Waning Crescent (29% of Full)

# I have reached franck@freebsd.at.work, my local, non restricted account at
# work, after successfully passing automatically thru the sshlock.
05:20:38:152:0 [franck@freebsd.at.work: ~ ]$ ls
projets/     system/
```

We can also launch commands directly :

```
#
#  freebsd.at.home (client)
#

# Let's automatically connect to my final destination on my new sshlocked
# system freebsd.at.work
# Notice the addition of the option -p (pass-thru)
[franck@freebsd.at.home: ~/sshlock/freebsd.at.work]$ ./sshlock-connect-
freebsd.at.work -i franck@freebsd.at.home -p ls
...
Enter passphrase for key '/home/franck/.ssh/ed25519.franck@linux.at.home':
...
Enter passphrase for key '/home/sshlock/.ssh/franck@linux.at.home':

projets
system

Connection to freebsd.at.work closed.
```

SSHLock's pass-thru mode perfectly illustrates the elegance of SSHLock's interface, as simple to simple to use as that of standard SSH, and that makes it a breeze to remotely connect to and use an SSHLocked-system:

```
[SSH]     $ ssh -i franck@freebsd.at.home franck@freebsd.at.work
[SSHLock] $ sshlock-connect-freebsd.at.work -i franck@freebsd.at.home
```

```
[SSH]     $ ssh -i franck@freebsd.at.home franck@freebsd.at.work ls
[SSHLock] $ sshlock-connect-freebsd.at.work -i franck@freebsd.at.home -p ls
```

SSH*Lock* versus SSH

SSHLock is not about remote connection per se, but a more general approach to securing and strengthening the logon access mechanism to computers in order to defeat attacks on passwords and render computers virtually impenetrable.

No matter how tightly configured, a single-layer SSH for remote access – by far the standard OpenSSH Configuration – is a simple line of defense that does not eliminate the single point of failure identified earlier. In other words, a single-layer SSH *does not* isolate the computer from the outside world : if it yields to an attacker, the computer is compromised.

In comparison, by providing a tightened passing-thru device (the sshlock's confinement, aka honey pot), SSH*Lock* isolates the computer's valuable content from the outside world : as opposed to the previous scenario, should the sshlock's first SSH layer yield to an attacker, the intruder is trapped by the honey pot instead of walking free inside the computer, without any possibility to peek and poke around!

SSH*Lock* is not about having two cryptographic keys, but about having two independent SSH-based access layers working cooperatively in series, and tightly configured to let an authorized user pass-thru the honey pot in two mandatory steps.

The security benefits brought in by SSH*Lock* in its own attempt to remedy the weaknesses identified earlier cannot be achieved by means of a simpler set-up. Should one set oneself to remove one of the two access layers or the sshlock's confinement, or both, or both, it won't be SSH*Lock* any more !

Conclusion

We have presented SSH*Lock*, a security scheme based on SSH to eradicate the threats commonly associated with the traditional password-based standard access mechanism to computers.

SSH*Lock* has been carefully devised to transpose into software the desirable security features found in physical security access locks in order to provide IT systems with the means to properly remove all weaknesses inherent to the password-based authentication access mechanism:

- By enforcing a mandatory two-steps access, SSH*Lock* eliminates the single point of failure identified earlier.
- By disabling password authentication and relying on the strongest cryptography available in OpenSSH, SSHLock removes the threats associated with weaker cryptography (“counter-measure a” above) and eliminates *verification*, the process by which an attacker can *verify* the validity of guessed / generated access keys (“counter-measure b” above).
- By providing for a layered-access architecture, including a restricted area between the two access mechanisms (the confinement or honey-pot), SSH*Lock* protects the system's valuable system accounts from being directly exposed to the network.

SSHLock is a transparent solution that upgrades existing systems to bastion hosts virtually impossible to break into, without ever forcing the company's access keys security policy to be modified.

SSHLock is naturally suited at securing servers, though it can also be used as a simple, yet powerful solution for securing remote access to Un*x workstations. To accommodate the later situation, it may prove practical to relax some of SSHLock's most restrictive hardening constraints by re-enabling system logon thru the system's physical terminal, possibly using OPIE or Kerberos authentication.

SSHLock as a valuable alternative to standard access mechanism and SSH service

By bringing a wealth of security features that, together, provide for a much safer access to computers over the standard password-based access mechanism, SSHLock is a safer, secure and powerful alternative to any password-based access mechanism, including standard SSH services for remote access.

Indeed, the credentials needed to successfully access an SSHLock-protected system, namely two strong passphrase-protected cryptographic key-pairs, knowledge of the TCP/IP ports, sshlock account and command proxy used the SSHLock deployment, name of the enabled (remote) local accounts, are orders of magnitude stronger than that of the standard password-based authentication access mechanism – often a simple, weak password.

However, this extra security does not have to come at a cost. Indeed, SSHLock provides the end-user with a simple, yet powerful interface, comparable in its ease of use to that of SSH, that makes it a breeze to connect to an SSHLock-protected system, thus hiding the complexities of manually passing thru an sshlock, an obvious benefit given the wealth of extra security brought up by SSHLock compared to a single layer SSH access.

SSHLock has been our private solution of choice for many years. Having professionally deployed it on a number of sensitive machines, mainly servers, it has consistently proven to be highly successful in hardening access to them without ever hindering using them. The SSHLock software is accessible at `http://www.franckys.com/products/sshlock/index.html`

Further work

We are currently investigating promising technologies that could be used to provide extra layers of protection around SSHLock and shield computers even more against the danger of the exterior world:

- **Port-knocking**. *Port-knocking* is like walking to a close bank and knocking on the outside door to politely request that the security access lock be activated so you can enter the bank and access your safe. Adding port-knocking to SSHLock would allow to hide sshd1, making it an on-demand service accessible only upon request, thus offering an extra layer of security.
- Setting up **Tor hidden services** for SSHLock would provide an additional layer of encryption and server authentication. Given the configuration would only accept connections from the hidden service or from the LAN, and will not disclose sshd1 fingerprint, people looking at the external traffic will never know the IP address, and will be unable to scan and target other services running on the server.

- Key storage. We could make good use of “`ssh-keygen -o -a $number`” to slow down cracking attempts by iterating many times the hash function [5].

About the author :

Franck PORCHER earned a PhD in Computer Science in Paris, France, where he started his career as a researcher for a decade within the flagship of the national aviation industry.

Wishing to explore more of the world and his own ideas, Franck has since reoriented himself as a startup entrepreneur, university professor, general purpose software engineer and open-source developer for the last twenty years, roaming between Tahiti, for its crystal-clear lagoons, Belize, for its equatorial exuberance, British Columbia, for its lush rain forests and wild mountains, and Paris, for its delicious breakfasts.

Franck is now a full-time software product editor and publisher.

You can easily find Franck on LinkedIn at : https://bz.linkedin.com/in/franckporcher

You are welcome to email Franck at : franck.porcher@gmail.com

References

[1] https://en.wikipedia.org/wiki/Brute-force_attack
[2] https://en.wikipedia.org/wiki/Social_engineering_(security)
[3] https://en.wikipedia.org/wiki/Airlock
[4] https://www.openssh.com/
[5] https://stribika.github.io/2015/01/04/secure-secure-shell.html