Practical Stuff on Linux

Disk encryption

- dm_crypt (dm = device mapper)

/dev/sda2 ≡ your real hard drive. everything is encrypted.

dm_crypt sits an abstraction layer on top of the real disk:

$ cryptsetup ... /dev/sda2 cryptroot

Now /dev/mapper/cryptroot works like a hard drive. Setup f.s. on cryptroot!

Note: if you have a magnetic HD, first overwrite w/ random bytes.

LUKS does a good job of using passwords to access HD.

- Strong KDF by default.
- Downside: lose plausible deniability
- Upside: deleting the header wipes out the disk (back up LUKS header?)
- Multiple passwords can be used!

Header has something like this:

$E_{pwd_0}(k)$ k ≡ actual AES key for the disk.

$E_{pwd_1}(k)$

⋮

$E_{pwd_n}(k)$

Note: can also encrypt /boot if you have grub 2 as your boot loader.

kernel images live here!

Bad news: cold boot attacks!

DRAM decay slows with colder temperatures...

Most bits might be unchanged after 5 - 10 minutes even with a crude cooling approach!

Forcefully power off computer & reboot (from attacker's usb)

Counter-measure: lock down BIOS - don't boot from usb w/o another pwd.

Counter-counter measure: remove DRAM, place in another computer

Counter-counter-counter measure: solder it??

Other notes: how to find keys? How to recover if they have a few bits toggled?

Bad news: ∃ redundant info that can be used as a sort of error correction (eg. key schedules for symmetric enc, & public keys)

Authentication, etc.

Recall best practices for handling passwords:

- Store hash
- make hash expensive to compute ← slows down dictionary attack
- salt ← prevents "offline" attack

On linux: add "rounds = XXX..." in /etc/pam.d/passwd

Documentation: man crypt
man unix_pam
man shadow

What about non-system passwords?
password safes like keepassX aren't bad.
gpg can also be used for something basic.

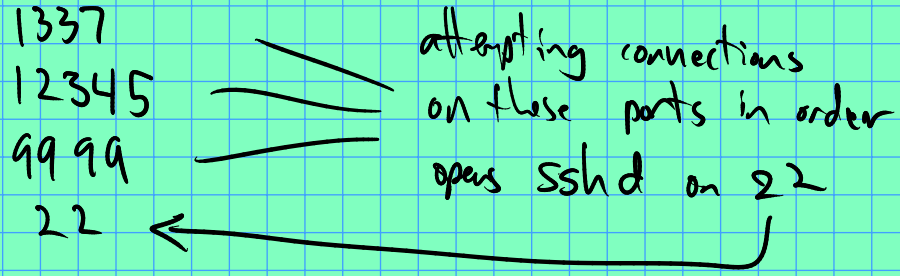
SSH public key authentication FTW!

In your sshd_config look for these:

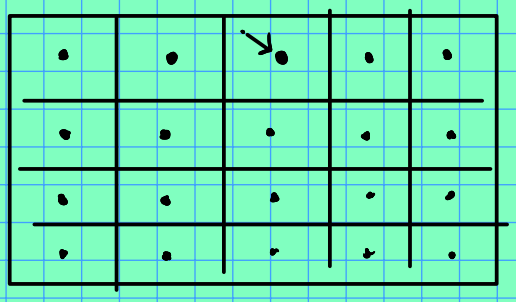
PasswordAuthentication no
ChallengeResponseAuthentication no
AllowUsers yourusername
PermitRootLogin no

Protecting your ssh private key: use ssh-keygen -a rounds

Protecting sshd from randos / port scanners:
cool trick with iptables: "port knocking"



Biometric authentication? How to store w/ similar protections to passwords? Biometric scans might be slightly different each time.
ECC?



Attept 1 : s = scan.

Store salt, $H(\text{salt} \| ECC(s))$

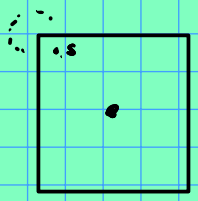
to check new scan, s',

see if $H(\text{salt} \| ECC(s'))$ equals the above.

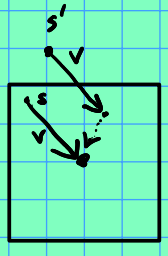
Problem: ideally, any "close" scan should work to authenticate.

This would be true if original happed to be a codeword.

But what if original looks like this:



Solution: "fuzzy sketch": store the ECC vector v in addition to the hash + salt.



Modified scheme: store salt, $H(\text{salt} \| ECC(s))$, v

Now to check a new scan s':

see if $H(\text{salt} \| ECC(v + s'))$ matches.

s' close to s $\Rightarrow$ authentication succeeds.

Leaks small amount of info (v), but not unreasonable.

for other items:

run command as root: use sudo.

Note: can explicitly list commands in the sudoers file...

edit file as root? (e.g. /etc/ssh/sshd_config)

~~sudo vim file~~

sudoedit file ✓

(The latter runs the editor w/ normal privilege to edit a temp file. Copies temp file after editor closes)

run command with lesser privilege?

sudo runuser -u nobody [command]

Note on setuid programs (e.g. sudo!)

These run w/ the permissions of the owner, (usually root!) no matter what user started them!

Useful for things like passwd command!

normal user runs it... and /etc/shadow is modified!

Such programs must be written very carefully...

e.g. the access/open pattern:

```
if ( access("file", —) )
    exit(1);
open("file", ...)
```

access checks if real user would have access to file.

Possible abuse:

```
touch /tmp/file
# run setuid program referencing /tmp/file
# between access + open:
ln -sf /etc/shadow /tmp/file   ⊛
```

"Time of check to Time of Use" bug.

Big picture items... "Reflections on Trusting Trust" by K. Thompson

What if someone modified the C compiler to miscompile certain functions... e.g. the 'login' command.

```
if (looks like compiling 'login()') ...
    // add extra password that is
    // always accepted.
else if (looks like I'm compiling myself...)
    // add the above
    // (as well as this)
```

Now if we revert the C compiler source code to its original form, but still compile w/ a bad binary, the login comand will be backdoored.

Other attacks could be even more subtle:

make binary for encryption produce output that's
- indistinguishable from that of the uncompromised also
- leaks the secret key used to encrypt!!

Ideas: use IV to encrypt key?

use a PRF $F : \{0,1\}^* \longrightarrow \{0,1\} \times \{0,1\}^N$

Compromised algorithm could rejection-sample ciphertexts (using different randomness each time) until $F(C) = (b, i)$ is such that

$$b = K[i]$$

K = victim's encryption key.

Defense: <u>Reproducible Builds</u>

- Establish connection between source code & binary.
- gives instructions by which anyone could produce an identical binary package.

(Debian is doing pretty well, btw.)