Festplattenverschlüsselung abseits von dm-crypt und ecryptfs

David Gstir
Richard Weinberger

sigma star gmbh

About Us

Richard Weinberger

- Co-founder of sigma star gmbh
- Linux kernel developer and maintainer
- Strong focus on Linux kernel, lowlevel components, virtualization, security

David Gstir

- CEO at sigma star gmbh
- Informatik TU Graz, IT-Security
- Focus: Cryptography, Security, Linux Kernel, macOS

Agenda

1. Disk encryption theory
2. Linux tools for disk encryption
3. Security Considerations
4. A detailed look on fscrypt
5. Summary

Disk Encryption Theory

Disk Encryption

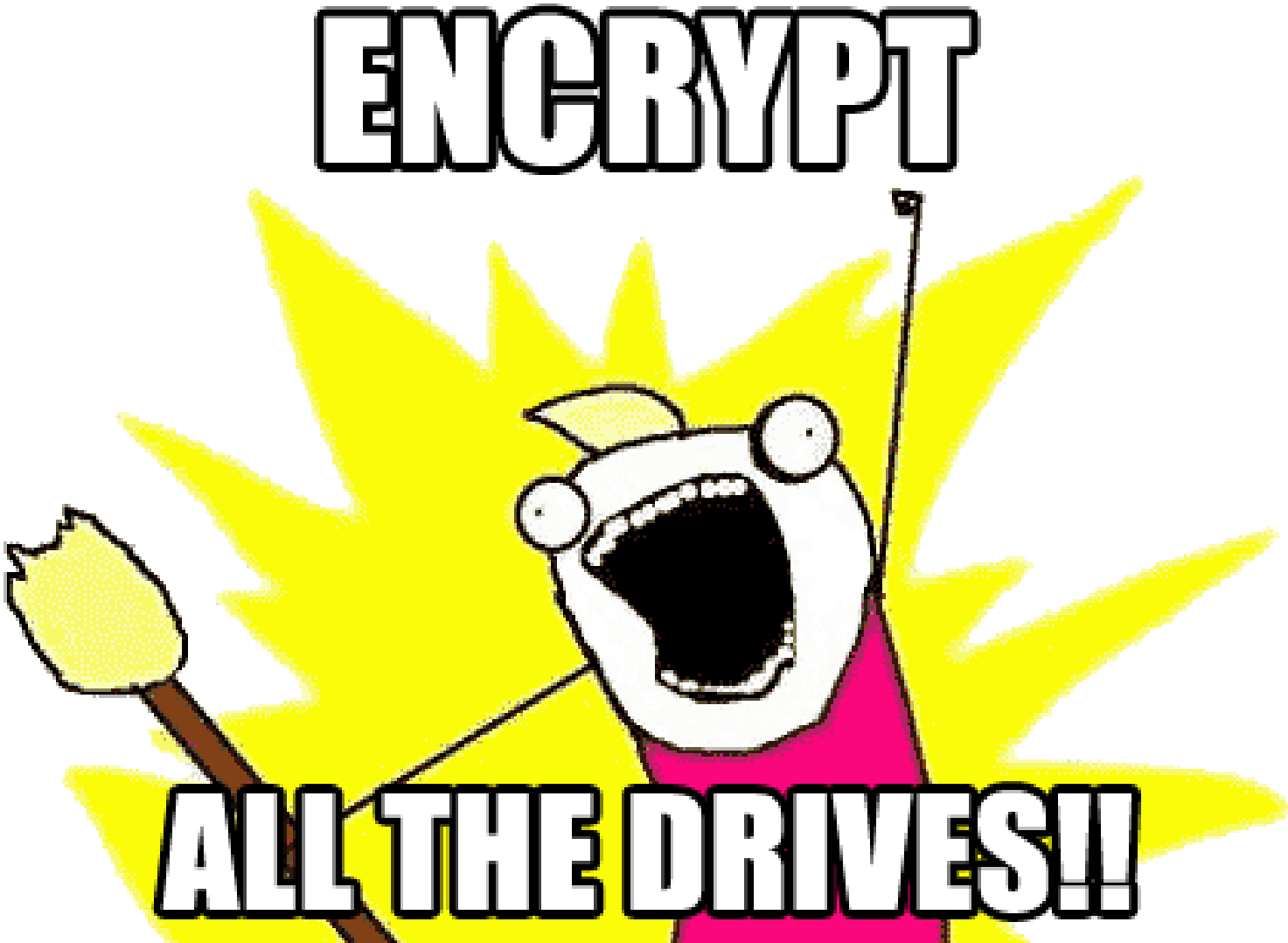
- Encrypt contents of persistent storage (HDD, SSD, etc.)
- Operating system or hardware takes care of encrypting/decrypting data when writing to/reading from storage
- Unlocked by user (eg. during boot)
- Processes and user see decrypted data

Variants

- Full disk encryption
- File level encryption (eg. user's home directory)

Why Disk Encryption?

- Attacker steals your device
- Can read all your data



Why Disk Encryption?

Won't help against

- Malware and similar stuff you download from the Internet
- Attackers that gain physical access to device while it is running or was recently powered down (*cold boot attack*)
- Modification of bootloader or similar (*evil maid attack*)
- Offline modification of encrypted data
- Even more problems when only partial disk encryption or file level encryption

Full Disk Encryption - FDE

- Encrypts every block written to persistent storage
- Includes file metadata (ie. filesystem internal structures)
- To be more secure, also encrypt swap partition
- On Linux: dm-crypt, TrueCrypt/VeraCrypt
- Also: dm-crypt + LUKS for better usability (eg. changeable disk encryption keys)

Partial Disk Encryption

- Classical use case: encrypt user's home directory
- Eg. through mounting a virtual encrypted disk
- Example: dm-crypt, TrueCrypt/VeraCrypt etc

File-Level Disk Encryption

- Encryption at file (system) level
- Encrypt file contents and certain metadata like file names
- No metdata encryption
- Examples: eCryptFS, EncFS, fscrypt or NTFS file encryption on Windows

Hardware-based Disk Encryption

- Also hardware-based possibilities (Self Encrypting Disks - SED)
- Encryption is done in disk firmware
- Not really better
- Sometimes worse:
  - https://www1.cs.fau.de/filepool/projects/sed/seds-at-risks.pdf
  - https://www.blackhat.com/docs/eu-15/materials/eu-15-Boteanu-Bypassing-Self-Encrypting-Drives-SED-In-Enterprise-Environments.pdf
- Also the NSA might control your HDD firmware: https://www.wired.com/2015/02/nsa-firmware-hacking/
- Firmware might also have bugs in encryption routines
- How often do you update your disk firmware?

The Cryptographic Side

- Data is encrypted using symmetric cipher like AES
- Almost all tools now use it in XTS mode
- Previously CBC mode, but that is less secure

The Encryption Key

- Derived commonly from user password
- Standard key derivation functions (PBKDF2, Argon2)
- Also directly from *keyfile* (eg. dm-crypt)
- Also possible to pair with 2nd factor (2-factor auth)

What if your disk breaks?

- All is lost if you don't have a backup ;)
- Single bit flip can corrupt full block/sector because of cipher mode

Disk Encryption on Linux

dm-crypt

- Encrypts full disk/partition
- Sits below file system
- Every block write is encrypted
- Every block read is decrypted
- Encryption key is derived from user password, or can be key file stored externally
- Common setups today:
  1. Physical storage
  2. dm-crypt + LUKS
  3. LVM
  4. Actual filesystem

LUKS: dm-crypt key management

- Enables multiple decryption keys (eg. unlock passwords)
- Generates real disk encryption key (master key) at random
- Encrypt master key with user's encryption key (password)
- Store master key in encrypted form on disk
- On user key change: just re-encrypt master key

eCryptFS

- Encrypts files ontop of an existing filesystem
- Stacked filesystem approach
- Stacking filesystems is problematic on Linux
- Has performance issues
- Setup not trivial

fscrypt

- New approach to do encryption at Filesystem level
- Started with ext4-specific implementation by Google for ChromeOS and Android devices
- Evolved into more generic framework within Kernel
- Currently: ext4, f2fs and ubifs support fscrypt
- ubifs support by David and Richard
- Still in very early stages
- Allows to define encryption policy for directory
- All files and directories within that directory will then be encrypted
- This is better than using a single key for everything from a security POV
- No authentication of data yet
- No metadata encryption

Security Considerations

Real Life

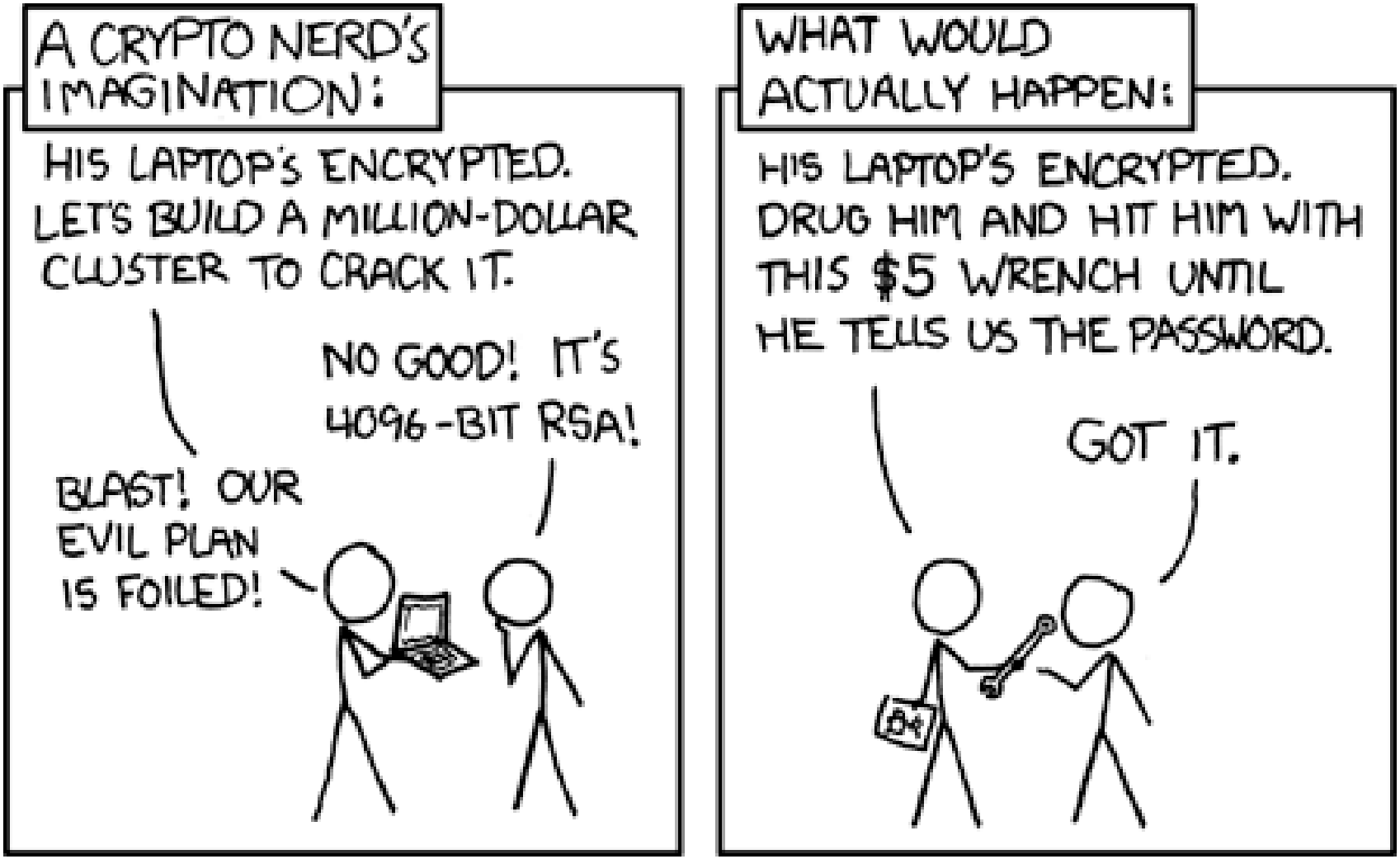


Figure 3: Real Life

Cold Boot Attacks

- (Software-based) disk encryption holds encryption key in RAM
- RAM is not always cleared on shutdown/powerloss
- Data in RAM takes minutes to degrade!
- Attacker can boot from external device and extract key from RAM
- Same holds for standby mode
- Normally OS takes care of clearing encryption key when powering down
- Also disable standby
- Attack also possible on SED firmware
- Better: keep key in CPU register or somewhere outside of RAM
- On Linux: TRESOR (key is in CPU debug register where userspace cannot access it)

Cold Boot Attacks

Figure 4: Cool RAM to get longer attack time

Evil Maid Attack

- Device is shut down
- Attacker modifies system such that malicious operations are performed on next boot
- Eg. malicious bootloader
- Device is booted and disk unlocked by user
- Malicious code is executed without user noticing!

Dealing with the evil maid

- Just encryption is not enough
- An attacker can always change encrypted data
- Authentication of encrypted data can help

What if attacker can install code (new bootloader, modified UEFI)?

- Having the disk encrypted and booting from CD-ROM/USB is worse
- Almost impossible to be sure on common x86 hardware
- *UEFI secure boot* and a chain of trust is good but not perfect -> still requires trust in vendor binaries
- We need open hardware without any firmware blobs
- Linux IMA/EVM subsystem detects file modifications: http://linux-ima.sourceforge.net/

Malleability Attacks

- Attacker is also able to modify encrypted data on disk without user noticing
- Data ist just encrypted, but not authenticated
- Attacker can modify known binary (eg /bin/bash) that it performs malicious action
- Practical attack shown against dm-crypt with AES-CBC
- http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/
- Specific AES-CBC attack preventable by using AES-XTS (not ideal)
- Would be fully preventable by using authentication, but...

Data Authentication

- Every encrypted byte is also authenticated
- Works eg. by using AEAD modes like AES-GCM, or MACs like HMAC-SHA256
- Requires to store additional authentication data on disk
- Takes up more space and decreases performance
- Hard to be done properly!
- Easier at file level than on block level (dm-crypt)

Strong Passwords

Figure 5: Strong Passwords

Single Encryption Key Problem

- Using the same encryption key for all disk data can be problematic (many GBs of data)
- Attacker can find identical plaintexts by looking for identical ciphertexts
- (Actually a bit more complex, but outcome is the same)
- This is an information leak
- Using same key for limited amount of data is better (like fscrypt does)

Should I use Disk Encryption then?

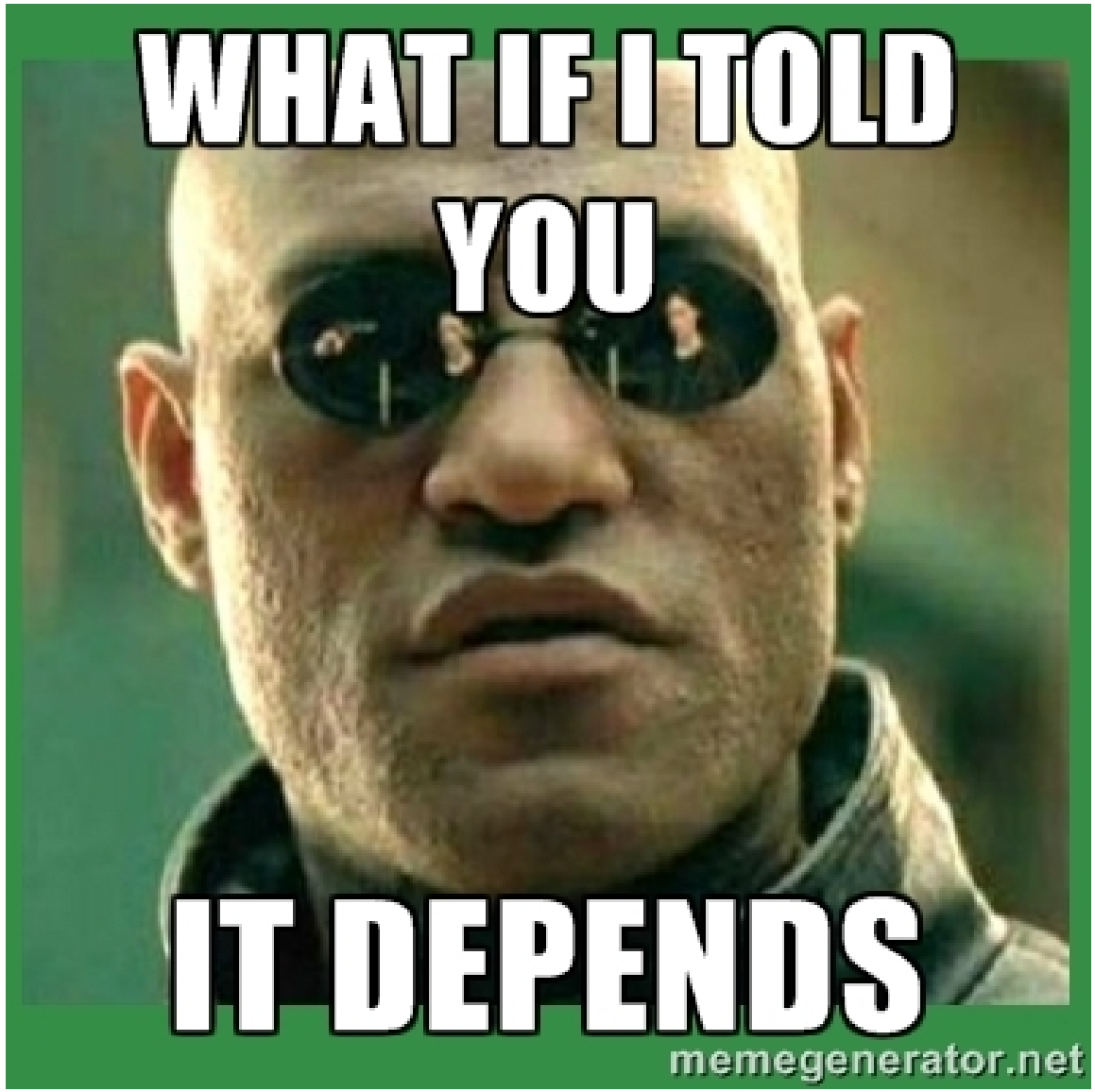


Figure 6: Well, it depends...

Should I use Disk Encryption then?

- Yes, but...
- **Disk encryption is a last resort!**
- Doing encryption at uppler layers (eg app layer) can be much more specific to use case
- Disk encryption often does not even know which block belongs to which file (eg. dm-crypt)
- At application layer we have much more possibilities and finer control (eg. PGP/GPG)

Proper Disk Encryption

- Getting secure setup is hard:
  - Encrypted swap
  - No encryption keys in RAM
  - Verified boot process
- Secure disk encryption code is hard:
  - Secure cryptographic primitives (AES-XTS)
  - Strong passwords/key material
  - 2-factor authentication
  - Implementation vulnerabilities
- If you get all of that right, your still not perfectly secure:
  - Malicious HDD firmware
  - Data authentication
  - ...

A detailed look on fscrypt

fscrypt

- Every inode will get its own encryption key derived from the master key
- If an attacker manages to break one File, only that file is exposed
- Filenames are encrypted too
- root can delete encrypted file without knowing the key
- Encrypting an ASCII string results to binary garbage
- If you readdir() an encrypted directory without a key it'd return binary
- Userspace expects stings, not binary, therefore base64 encoded strings are returned.
- Base64 makes data longer, what happens if you have files that reach the maximum file length?

Showcase: fscrypt for UBIFS

- fscrypt is generic, so adding fscrypt to a new filesystem should be easy
- NOT
- In order to support fscrypt we had to change UBIFS in some ways
- Internally UBIFS assumes that filenames are strings, any byte except the NUL byte is allowed. An encrypted filename can have 0 bytes anywhere.
- We had to replace all strcmp() with memcmp()
- UBIFS supports transparent compression, when do you compress? Before encryption? After? Not at all?

More details on encrypted filenames

- Turned out to be much more work than expected
- On Linux (and any other modern OS) a filesystem has to offer a function to lookup files.
- It receives a filename and returns either an error code or a file handle
- Many cases have to be handled

1. no encryption: requested name is used 1:1 for search on disk
2. encryption enabled, key present: requested filename is plain text and has to be encrypted to find it on disk
3. encryption enabled, key not present: requested filename is base64 of cipher text and needs to be decoded first
4. see next slide

More details on encrypted filenames (cont'd)

- A filename can have up to 255 bytes, without NUL byte.
- When no key is present a base64 encoded string will be returned.
- Filenames with maximum length will get larger due to the 8 to 7 bit translation of base64.
- Could break userspace.
- Limiting the maximum filename length to less than 255 bytes is also not good since applications cannot know whether fscrypt is enabled or not for some directory.
- For long filenames fscrypt returns a base64 encoded unique lookup cookie to userspace as filename.
- So the 4th case is: encryption enabled, key not present, long name: decode the cookie and use it for a cookie based lookup.
- Sounds more easy than it is.

More details on encrypted filenames (cont'd)

- Internally UBIFS stores all information in a B-Tree where the 32 bit hash of the filename is used for lookups.
- 32 bit hashes can clash, so lookup works like a hash table lookup, using the hash you find the right bucked, then you do a string-compare to find the right entry.
- Easy for regular lookups since UBIFS knows the filename at lookup time and computes the hash on its own.
- In the 4th case we don't have the filename, only a cookie.
- Looking up just based on 32 bit hash is error prone: birthday attack!
- A evil user can create ~100000 files in the same directory and trigger a hash collision
- Can result in an undeletable file or deletion results of delete of some other file

More details on encrypted filenames (cont'd)

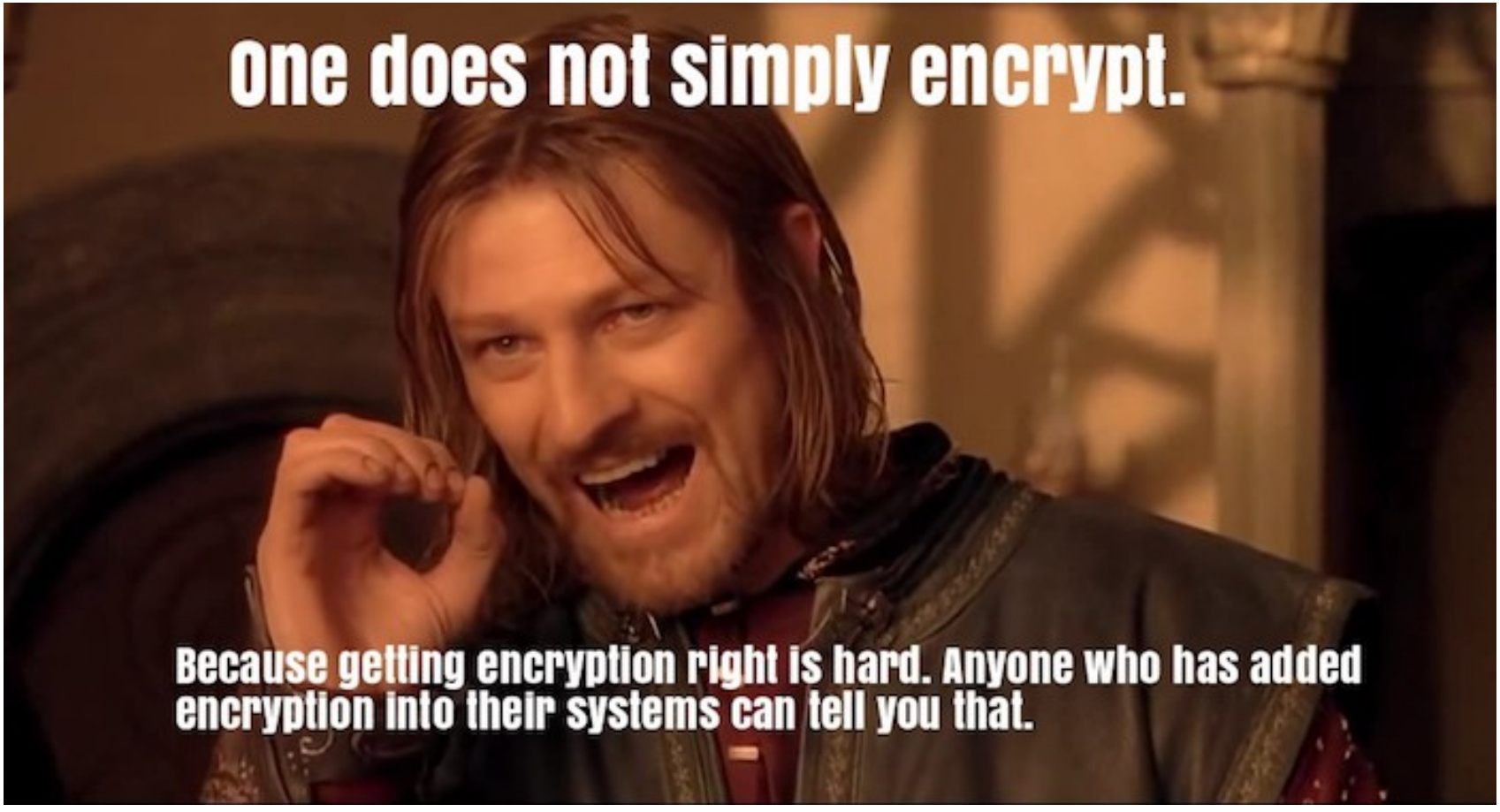
- To deal with that we implemented another 32 bit cookie in UBIFS
- Using both cookies we can do 64 bit lookups
- First 32 bit cookie is r5_hash(filename), what to use for the second cookie?
- another_hash(filename)?
- We used a 32bit random number. It is nothing better or worse than yet another hash of the filename.
- So, the cookie as presented to userspace is good enough for lookups
- To be very sure we include also the last few bytes of the encrypted filename
- Side effect: 64 bit lookup cookies allowed us finally to support seekdir()/telldir() via NFS on UBIFS.

Summary

Disk Encryption and mobile devices

- Actually more complex: encryption key bound to user's unlock code
- Android uses ARM's TrustZone to store encryption key
- Is broken at least for certain devices: https://arstechnica.com/security/2016/07/androids-full-disk-encryption-just-got-much-weaker-heres-why/
- Apple's iOS uses a more advanced system where encryption key **never** leaves Secure Coprocessor (SecureEnclave)
- See https://www.apple.com/business/docs/iOS_Security_Guide.pdf
- Evict key when device is locked/in standby?
- What to do while device is locked and emails/SMS/... arrive?

Disk Encryption is hard...



Disk Encryption is hard...

- Many things to get wrong
- Then still only secure against certain scenarios
- Better to do encryption at higher layer
- (Full) disk encryption is last resort
- fscrypt is new approach file system level with more control

Thank you!

Questions, Comments?

David Gstir
david@sigma-star.at

Richard Weinberger
richard@sigma-star.at