Hacking Android Disk Encryption for Fun and Profit

David Gstir

david@sigma-star.at

sigma star gmbh

Hi!

- All things security @ sigma star gmbh
- Security audits of basically everything that runs code
- Engineering and consulting around Linux, embedded systems and security
- Trainings

sigma star gmbh

Smartphone Storage Security

- Smartphones are used for security critical things:
  - Banking (remember when TANs had to be separated from banking device?)
  - Electronic signatures: eIDAS, ID Austria etc.
  - ...
- We take our smartphones to far more places than a regular Notebook - and we lose it more easily
- Needs strict security controls (disk encryption is just one of many)
- Similar threat model as embedded systems
- *At least since EU CRA secure storage is a hard requirement!*

What About Android?

- Android (AOSP - Android Open Source Project) does a lot of this out of the box
- Uses hardware support when possible (e.g. ARM TrustZone)
- Vendors can modify that as they see fit
- But how exactly? And how secure is it?

I recently had the chance to look at this in more detail.

The story goes something like this…

Act 1: A Quest!

A Curious Request

X: Can you help me recover data from my Android phone?

me: Do you have a backup?

X: Well yes, but it is not a regular backup.
It's a low-level disk dump of the encrypted disk.

me: 🤨🤔

sigma star gmbh

The Goal

After meeting in person the situation became clear:

- Samsung Galaxy S21 Ultra
- Was running Android 11
- Got stuck in boot loop
- Owner created low-level dump of internal encrypted storage
- Owner flashed stock Android 12

What I Knew About Android Disk Encryption

- Android uses *fscrypt* to encrypt filesystem contents at file level
- This is the same as in Linux as Google contributed their Android Kernel changes to mainline Linux
- I worked plenty on that in the past (UBIFS fscrypt integration)
- All fscrypt needs is an encryption key which some program hands to the kernel (through a *ioctl(3)* call)
- Assumptions:
  - Key is stored as encrypted “blob” on unencrypted partition
  - ARM TrustZone has to play some part in this
  - User lock code or biometrics have to be involved
  - Should be doable, but surely not as easy as it sounds 😉

Act 2: Hunt for the Key

sigma star
gmbh

First Attempt

- First idea: restore backup and try to boot
- Fail: attempting to downgrade to stock Android 11 did not work
- Samsung prevents downgrade once Android 12 was installed
- Only done in case of major security vulns!
- Reason: Vuln. in their TrustZone key blob mechanism1)
- **This would have made my task that much easier!**
- However: Open sourced tooling (*keybuster*) which is helpful for us

sigma star gmbh

1) Paper *Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design by Shakevsky et al*

Interlude: Mapping the Dungeon

sigma star gmbh

Inner Workings of Android FBE

Since Android 9 there are **multiple** layers of encryption:

1. **Metadata encryption:** *dm-crypt*-like block encryption (below filesystem) plus *fscrypt* on top for some folders
2. **Device encrypted (DE) storage:** *fscrypt* on some folders
3. **Credential encrypted (CE) storage:** again *fscrypt*, but requires user lock code

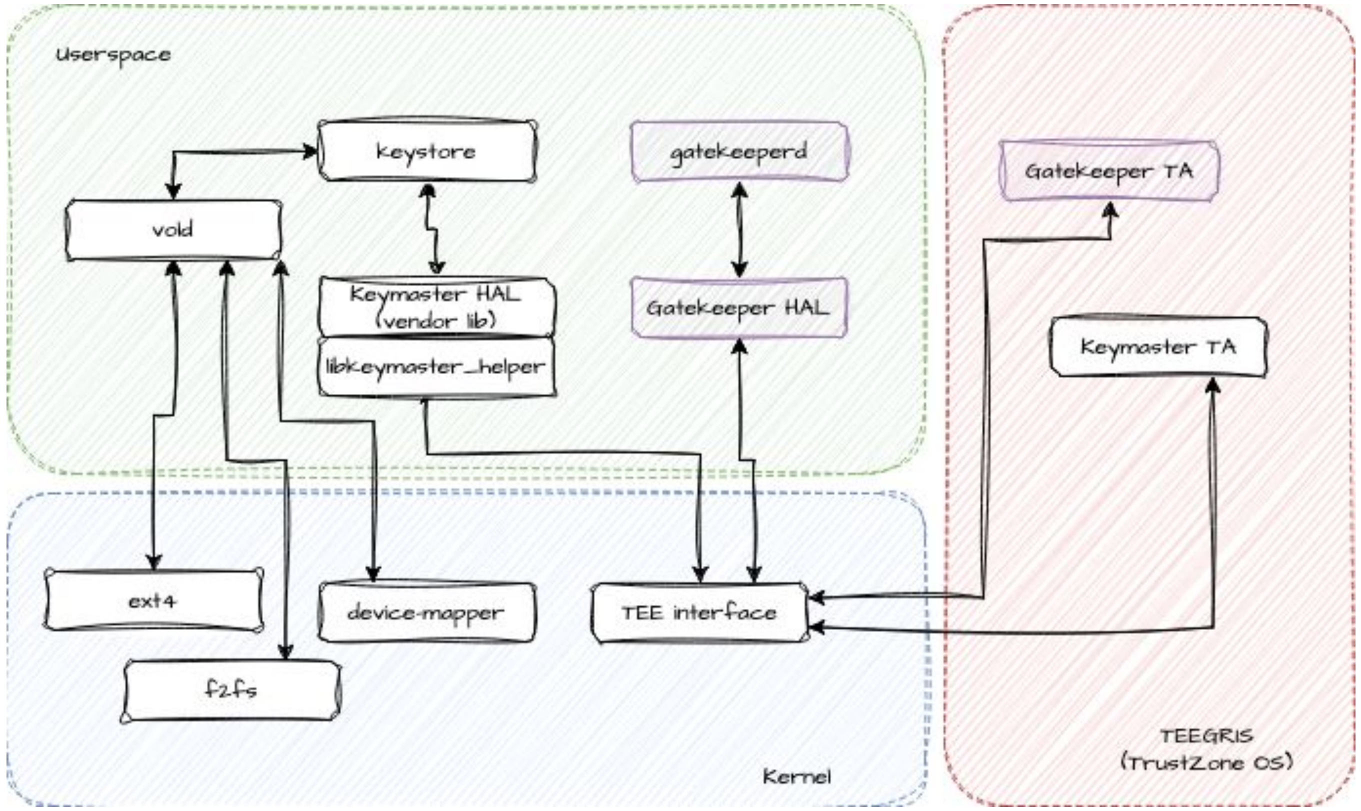
High-Level Mount Logic

Relevant parts of mount flow during boot (mainly *vold* service):

1. Mount */metadata* (not encrypted)
2. Unwrap metadata key
3. Attach DM (Device Mapper) volume *“userdata”* using *dm-crypt* equivalent
4. Mount as */data* (f2fs)
5. Unwrap metadata *fscrypt* key and add to Kernel
6. Unwrap and load DE key
7. On device unlock: Unwrap and load CE key

All key unwrapping is done through TrustZone call

Master of Keys: Android Keystore



sigma star gmbh

Master of Keys: Android Keystore

The Android Keystore API manages all keys:

- Keymaster TA (Trusted App.) in TEE (Trusted Exec. Env. aka TrustZone) is doing unwrap
- Called via a Kernel interface by *keystore* daemon
- *keystore* (and the associated Keymaster HAL library) contain a lot of checks we have to pass for a call to succeed
- Samsung extra: *libkeymaster_helper.so* conctains logic to call Kernel Keymaster TA interface
- Trick from keybuster: bypass *keystore* by simply calling directly into *libkeymaster_helper.so* from our process

Act 2.1: Hunt for the Key_s_!

sigma star
gmbh

Key Blob Storage Files

For metadata encryption AOSP source reveals key blob in */metadata/vold/metadata_encryption/key/*.

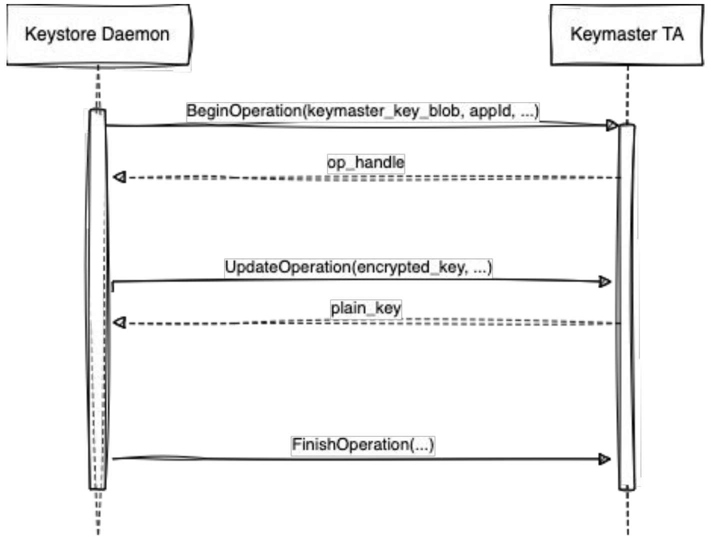
Contains multiple files:

- *secdiscardable*: used the generate *“AppID”*
- *stretching*: contains *nopassword* so we can ignore it
- *encrypted_key*: the key blob we want to decrypt
- *keymaster_key_blob*: the key used by Keymaster TA to decrypt *encrypted_key* - is encrypted with Keymaster TA internal key

Unwrap, Please!

Revere engineering some functions from *libkeymaster_helper.so* gives us:

- *nwd_begin(...)*: starts unwrap with key encryption key. Args: *AppID, key blob, etc.*
- *nwd_update(...)*: performs unwrap with key blob yielding plaintext key
- *nwd_finish(...)*: does cleanup


Keystore Daemon
Keymaster TA
BeginOperation(keymaster_key_blob, appId, ...)
op_handle
UpdateOperation(encrypted_key, ...)
plain_key
FinishOperation(...)

sigma star gmbh

Full Unwrap Code

Pseudocode of unwrap logic using *libkeymaster_helper.so*:

```
unwrap_vold_key() {
      secdiscard = read_file("./secdiscardable");
      app_id = generate_appid(secdiscard); // essentially SHA512 with some constants added
      keyblob = read_file("./encrypted_key");
      kek = read_file("./keymaster_key_blob");
      in_params = generate_in_params(keyblob[:12] /* nonce */ );
      dummy = {0};
      nwd_begin(KM_PURPOSE_DECRYPT, kek, in_params, NULL, &dummy, &handle);
      nwd_update(handle, NULL, keyblob[12:], NULL, NULL, &dummy_cnt, &dummy, &plain_key);
      nwd_finish(handle, NULL, NULL, NULL, NULL, NULL, &dummy, NULL);
}
```

sigma star gmbh

Key Blob Parameters

in_params for Keymaster TA:

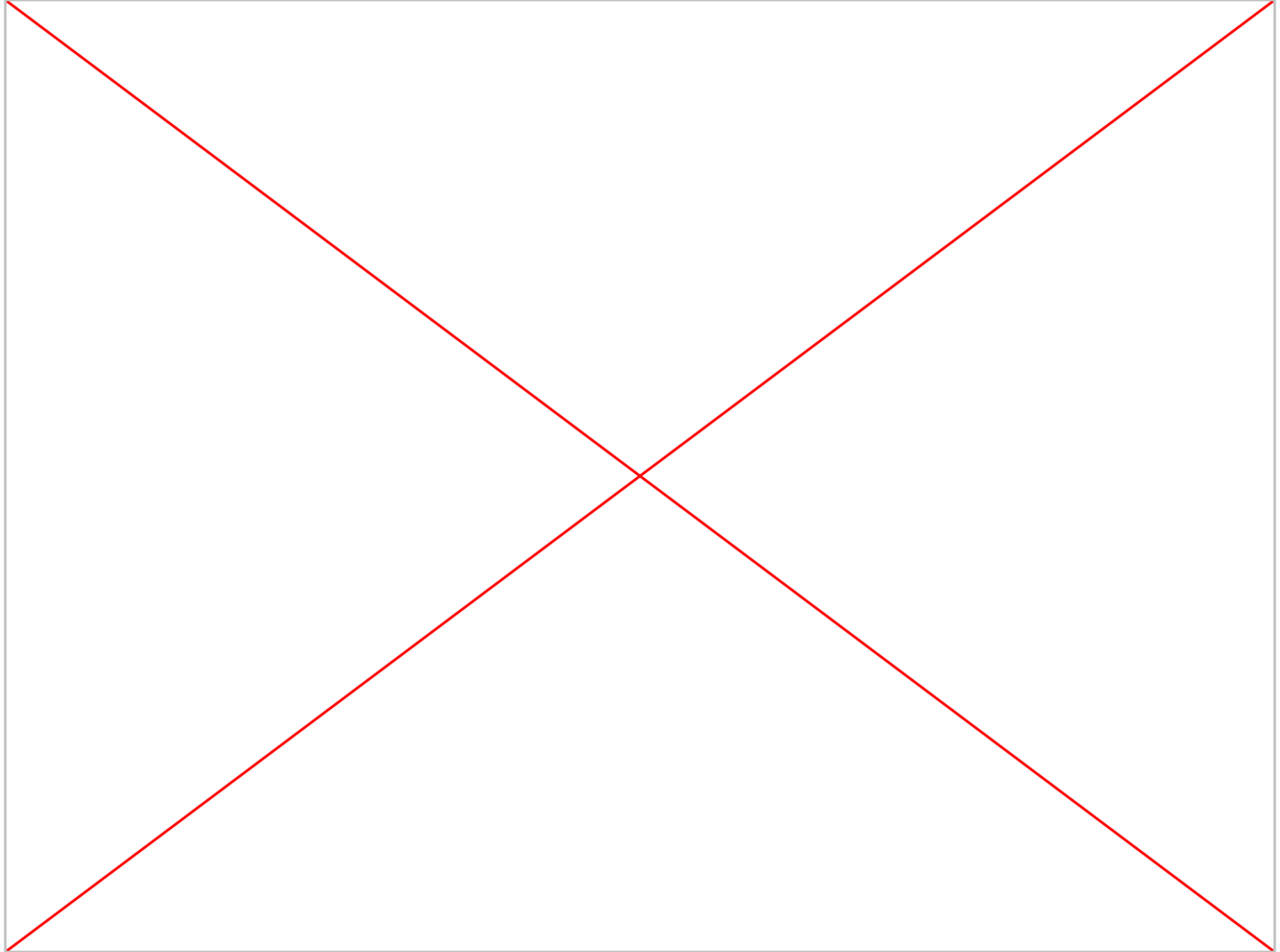
- 256-bit AES key
- 128-bit GCM MAC (no padding, 128-bit min MAC length)
- Nonce from key blob
- Tag *AppID*: needs the *AppID* generated from *secdiscardable* file (SHA512 with some constants added)
- Tag *TAG_NO_AUTH_REQUIRED*: no user credentials needed
- Tag *TAG_ROLLBACK_RESISTANCE* (if possible, re-tries without afterwards)

Delving Deeper

Now we can mount userdata partition (*/data*) which holds the key blobs for *fscrypt*:

- */data/misc/vold/user_keys/de/0/*: user DE (device encrypted)
- */data/misc/vold/user_keys/ce/0*: user CE (credential encrypted)
- */data/unencrypted/key*: needed to access above folders
- Unwrapping keys in */data/unencrypted/key* and */data/misc/vold/user_keys/de/0/* only required minimal changes to our existing unwrap logic

Interlude: Demo || GTFO



Act 3: Attack of the Hidden Chip

sigma star
gmbh

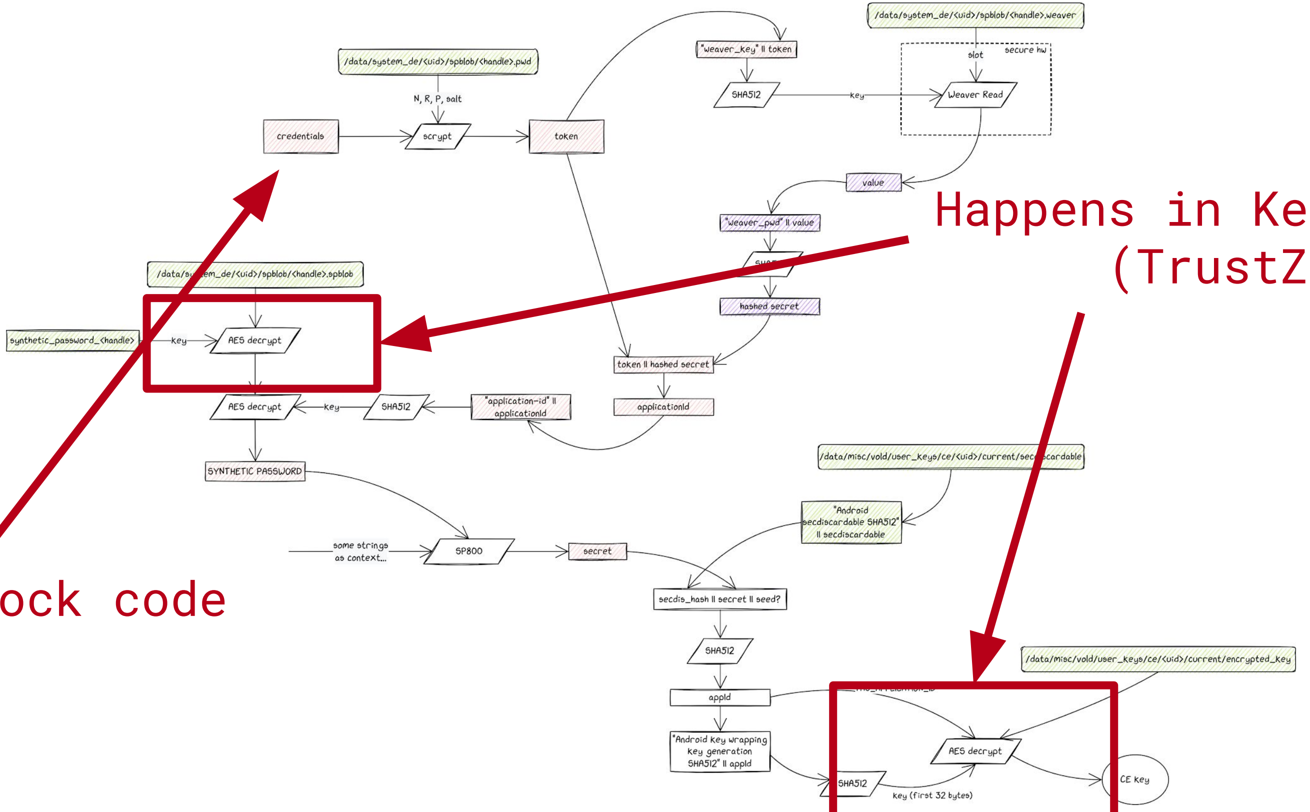
What Happened So Far...

- Flashing Android 12 did not invalidate key blobs from backup
- No lock code or biometrics needed yet
- *TrustZone (Keymaster TA) does not care who calls it and what state the Android OS is in (rooted or not)!*

Introducing: The Weaver Security Chip

- Newer phones have what AOSP calls a “Weaver” security chip - similar to a Key-Value store
- The Samsung Galaxy S21 Ultra is the first Samsung flagship phone having this chip
- During initial setup, Android enrolls a random secret key there (i.e. the value of the Key-Value store)
- This **secret key** is combined with the **lock code** and used as input to Keymaster TA unwrap call

Map to the Key



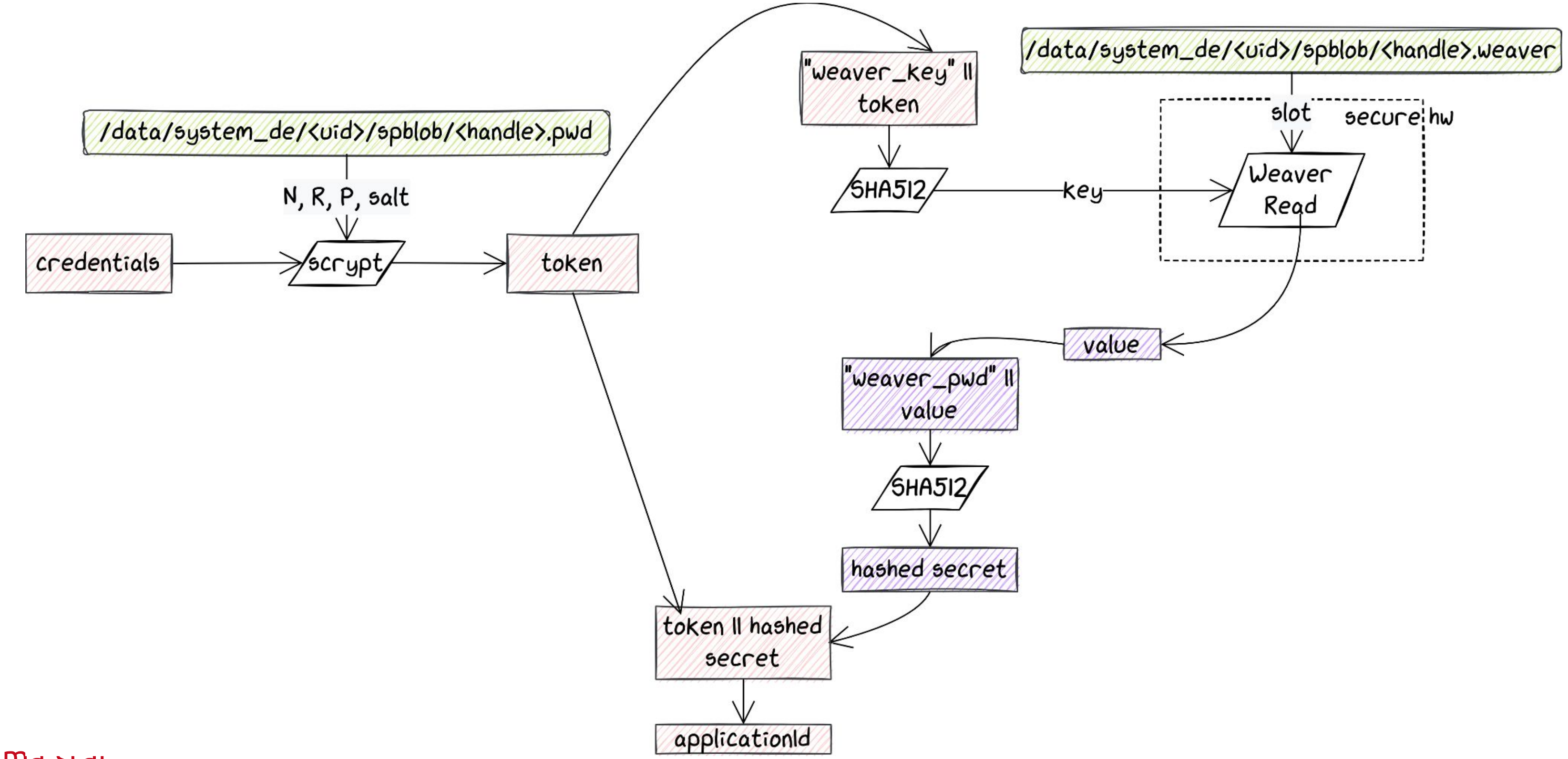
/data/system_de/<uid>/spblob/<handle>.pwd
N, R, P, salt
credentials
scrypt
token
"weaver_key" ll token
SHA512
key
/data/system_de/<uid>/spblob/<handle>.weaver
slot
secure hw
Weaver Read
value
"weaver_pwd" ll value
hashed secret
token ll hashed secret
applicationId
/data/system_de/<uid>/spblob/<handle>.spblob
synthetic_password_<handle>
key
AES decrypt
AES decrypt
key
SHA512
"application-id" ll applicationId
SYNTHETIC PASSWORD
some strings as context...
SP800
secret
/data/misc/vold/user_keys/ce/<uid>/current/secdiscardable
"Android secdiscardable SHA512" ll secdiscardable
secdis_hash ll secret ll seed?
SHA512
appId
"Android key wrapping key generation SHA512" ll appId
/data/misc/vold/user_keys/ce/<uid>/current/encrypted_key
AES decrypt
SHA512
key (first 32 bytes)
CE key

Happens in Keymaster TA (TrustZone)

User's lock code

source: blog.quarkslab.com annotations: /me

Weaver Security Chip Interaction



sigma star gmbh

source: blog.quarkslab.com

Unrecoverable Key

- It looks like the Weaver chip is reset when a new Android OS first set up
- Cannot derive all the inputs for Keymaster TA to decrypt the CE key blob
- Brute force is also not possible: search space too large (32-byte value)
- Only chance: Samsung changed AOSP code and messed up or I missed something (totally possible 😉)

Act 4: Fun and ~~Profit~~ Insights

sigma star gmbh

Insights: Android Storage Security

- Android has good multi-layered approach for disk encryption
- More fine-grained control over who can access what parts of disk
- Still no per-App encryption though - once CE key is known, user’s App data is decryptable
- Disk encryption is only useful in combination with other mechanisms: verified boot, strict SELinux policy, etc.
- TrustZone OS does not care about state of Android OS at all
- Privileged exploit can still talk to TrustZone as we did and unwrap key blobs (not only for disk encryption)

Insights: Encrypted Backup Recovery

- The encrypted Disk Backup could not be recovered
- Without the device this would be impossible to do
- Backup your smartphone!
- Encrypting the backup is good, but be sure to backup the encryption key too
- Other devices (embedded and regular) should adopt some of the concepts from Android

Chances for a sequel to this quest: minimal - though I did just discover one more thing to test today 😅

Credits

- Huge thanks to the owner of the Android phone! 😁
- Keybuster tooling and writeup on Samsung TrustZone hacking: https://github.com/shakevsky/keybuster
- Quarkslab Blog post on Android FBE: https://blog.quarkslab.com/android-data-encryption-in-depth.html
- Frida for dynamic analysis on device: https://frida.re
- Ghidra for reversing: https://ghidra-sre.org
- Android AOSP and Android Code Search: https://cs.android.com

sigma star gmbh

The End.
Questions?

David Gstir

david@sigma-star.at

sigma star gmbh

Bonus: Post-Credits Scene

sigma star
gmbh

Are Evil Maid Attacks Possible?

Not out of the box, but:

- Bypass of verified boot will enable it
- Unlocked bootloader is a problem
- Downgrade attacks are a problem (usually possible to do)
  - Samsung did properly prevent downgrade in case of major vuln. in our case

sigma star gmbh

Metadata Encryption

- Lowest encryption layer and first to unlock during boot
- Called *dm-default-key* - pretty much the same as *dm-crypt* in Linux
- Encrypts storage blocks and sits beneath filesystem
- Key is added to Kernel via device mapper ioctls: *DM_DEV_CREATE, DM_TABLE_LOAD, DM_DEV_SUSPEND*
- Similar to fscrypt all we need is the encryption key and hand it to the Kernel

Device Encrypted Storage

- Second layer of encryption
- Encrypts part of storage that need to be accessible right after boot (before lock code is provided)
- Uses *fscrypt*
- Encrypts based on per-directory policy (only parts of filesystem is encrypted)
- Key is added to Kernel *ioctl(3)*: *FS_IOC_ADD_ENCRYPTION_KEY*

sigma star gmbh

Credential Encrypted Storage

- Last encryption layer protecting user data (profile)
- Also uses *fscrypt*, so similar to DE storage
- Requires biometrics or passcode to unlock
- Will be hardest part as requires (more) interaction with TrustZone