ASSURED

Control Integrity Verification - Attestation Scheme

Alexandros Sampanis

Security Engineer

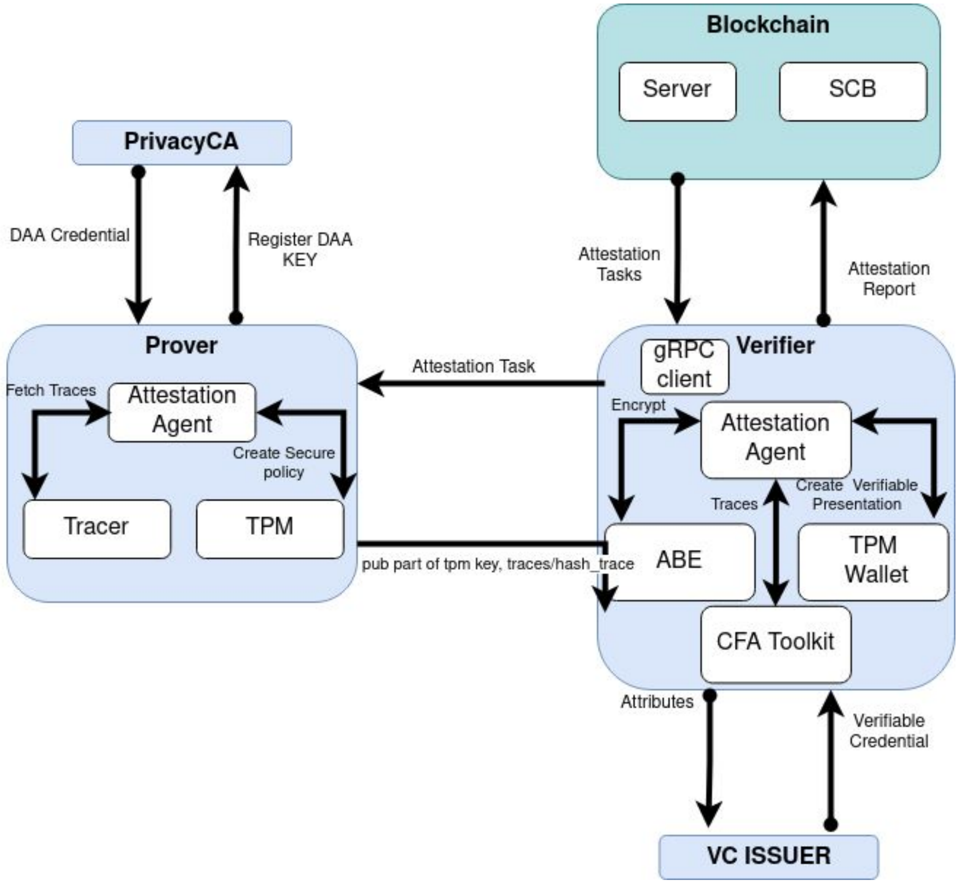
UBITECH Ltd

Demo webinar on Attestation Primitives

Online | 31 May 2023

www.project-assured.eu

ASSURED Architecture Overview



Control Integrity Verification

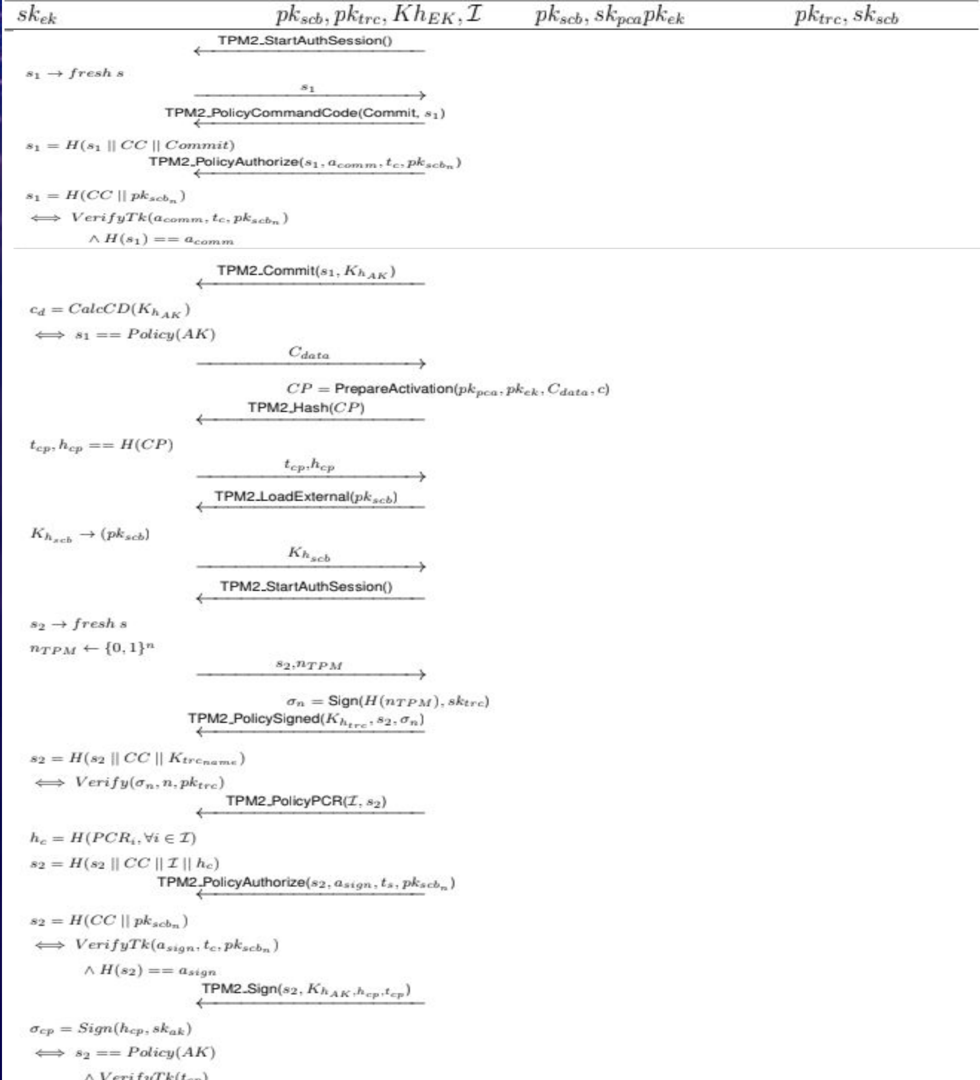
ASSURED Context → Enables enrolled devices to attest to their correct configuration during run-time when requested through a deployed attestation policy

This protocol is responsible for calculating the configuration of a running binary as a hash digest so it can be used from the TPM as a key-restriction usage policy. To make sure that every time we fetch it, a hashed trace of a binary is compared ‘on-the-go’ with the expected one (golden hash), we use the Platform Configuration Registers (PCR) of the TPM to store and have direct access to it through tpm commands like PCR_Extend and policy_PCR.

Key Restriction Usage Policies

In order to integrate Local Attestation into our scheme we exploit the key-restriction usage policies. A Trusted Third Party (SCB) constructs a policy digest, which will be binded with the AK.

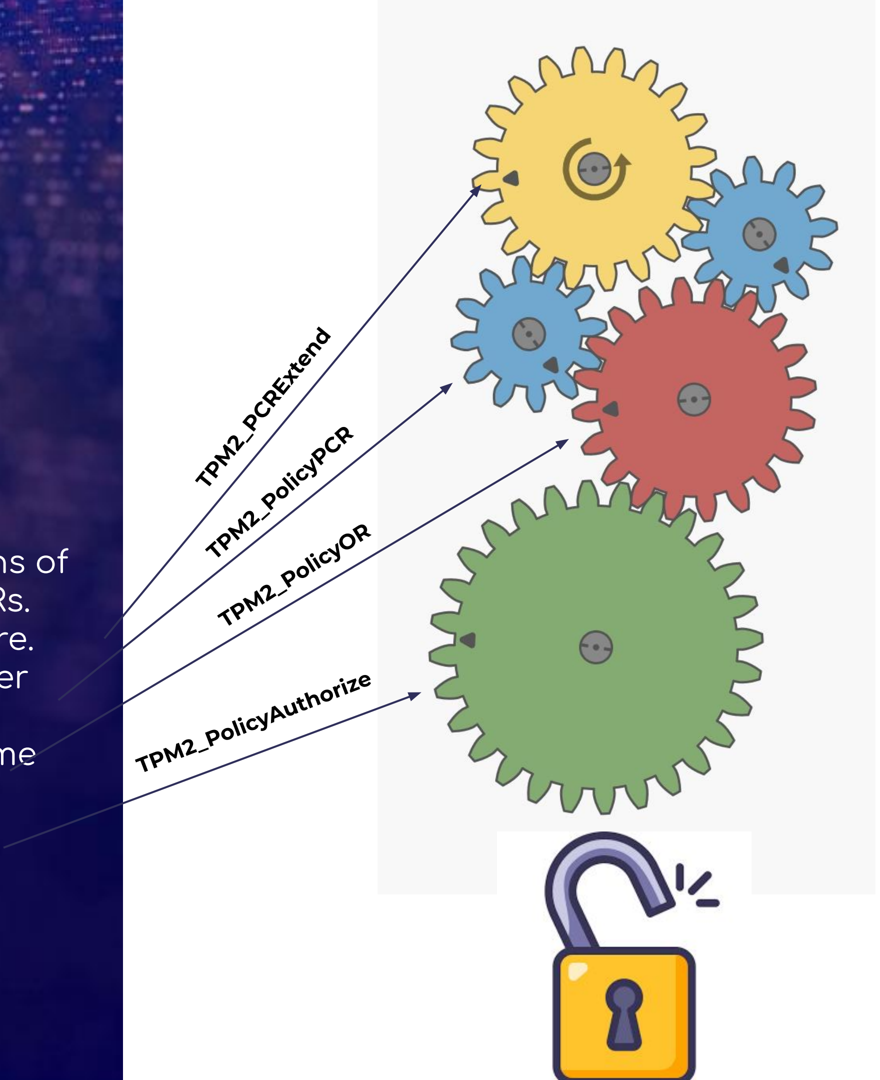
- Load from the Platform Configuration Registers a pre-defined correct state of the Device (TPM2_policyPCR).
- Verify the the freshness and integrity of the the output of the Tracer (TPM2_policyOR | TPM2_policySIGNED).
- Use a certificate from a Trusted Third Party (SCB) to construct the final run-time policy Digest and gain access to the AK key (TPM2_policyAUTHORIZE).
    - Using TPM2_policyAUTHORIZE the protocol can support updates during run-time.



sk_{ek}
$pk_{scb}, pk_{trc}, Kh_{EK}, \mathcal{I}$
$pk_{scb}, sk_{pca} pk_{ek}$
pk_{trc}, sk_{scb}
TPM2_StartAuthSession()
$s_1 \rightarrow fresh\ s$
s_1
TPM2_PolicyCommandCode(Commit, s_1)
$s_1 = H(s_1 \| CC \| Commit)$
TPM2_PolicyAuthorize($s_1, a_{comm}, t_c, pk_{scb_n}$)
$s_1 = H(CC \| pk_{scb_n})$
$\iff VerifyTk(a_{comm}, t_c, pk_{scb_n})$
$\wedge H(s_1) == a_{comm}$
TPM2_Commit($s_1, K_{h_{AK}}$)
$c_d = CalcCD(K_{h_{AK}})$
$\iff s_1 == Policy(AK)$
C_{data}
$CP =$ PrepareActivation($pk_{pca}, pk_{ek}, C_{data}, c$)
TPM2_Hash(CP)
$t_{cp}, h_{cp} == H(CP)$
t_{cp}, h_{cp}
TPM2_LoadExternal(pk_{scb})
$K_{h_{scb}} \rightarrow (pk_{scb})$
$K_{h_{scb}}$
TPM2_StartAuthSession()
$s_2 \rightarrow fresh\ s$
$n_{TPM} \leftarrow \{0,1\}^n$
s_2, n_{TPM}
$\sigma_n =$ Sign($H(n_{TPM}), sk_{trc}$)
TPM2_PolicySigned($K_{h_{trc}}, s_2, \sigma_n$)
$s_2 = H(s_2 \| CC \| K_{trc_{name}})$
$\iff Verify(\sigma_n, n, pk_{trc})$
TPM2_PolicyPCR($\mathcal{I}, s_2$)
$h_c = H(PCR_i, \forall i \in \mathcal{I})$
$s_2 = H(s_2 \| CC \| \mathcal{I} \| h_c)$
TPM2_PolicyAuthorize($s_2, a_{sign}, t_s, pk_{scb_n}$)
$s_2 = H(CC \| pk_{scb_n})$
$\iff VerifyTk(a_{sign}, t_c, pk_{scb_n})$
$\wedge H(s_2) == a_{sign}$
TPM2_Sign($s_2, K_{h_{AK}}, h_{cp}, t_{cp}$)
$\sigma_{cp} = Sign(h_{cp}, sk_{ak})$
$\iff s_2 == Policy(AK)$
$\wedge VerifyTk(t_{cp})$

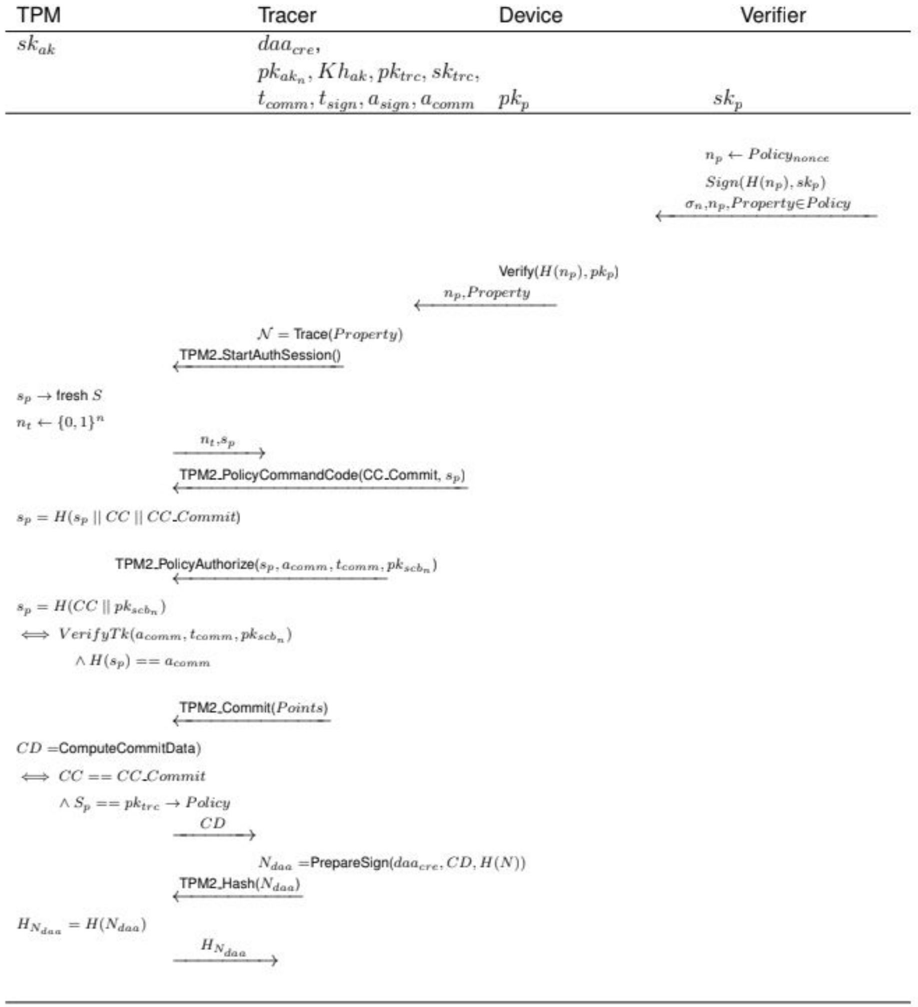
Secure Enrollment

- Why do we need a Secure on-boarding?
  - Key creation with Trust.
  - Block unauthorized key usage.
- How is this achieved?
  - Key Restriction Usage Policies:
    1. Store pre-defined representations of the device state in the TPM's PCRs.
    2. Attest against all allowed software.
    3. Establish trust between the Tracer and the TPM.
    4. Verify that this configuration came from trustworthy source.
- What did we achieve?
  - Local attestation



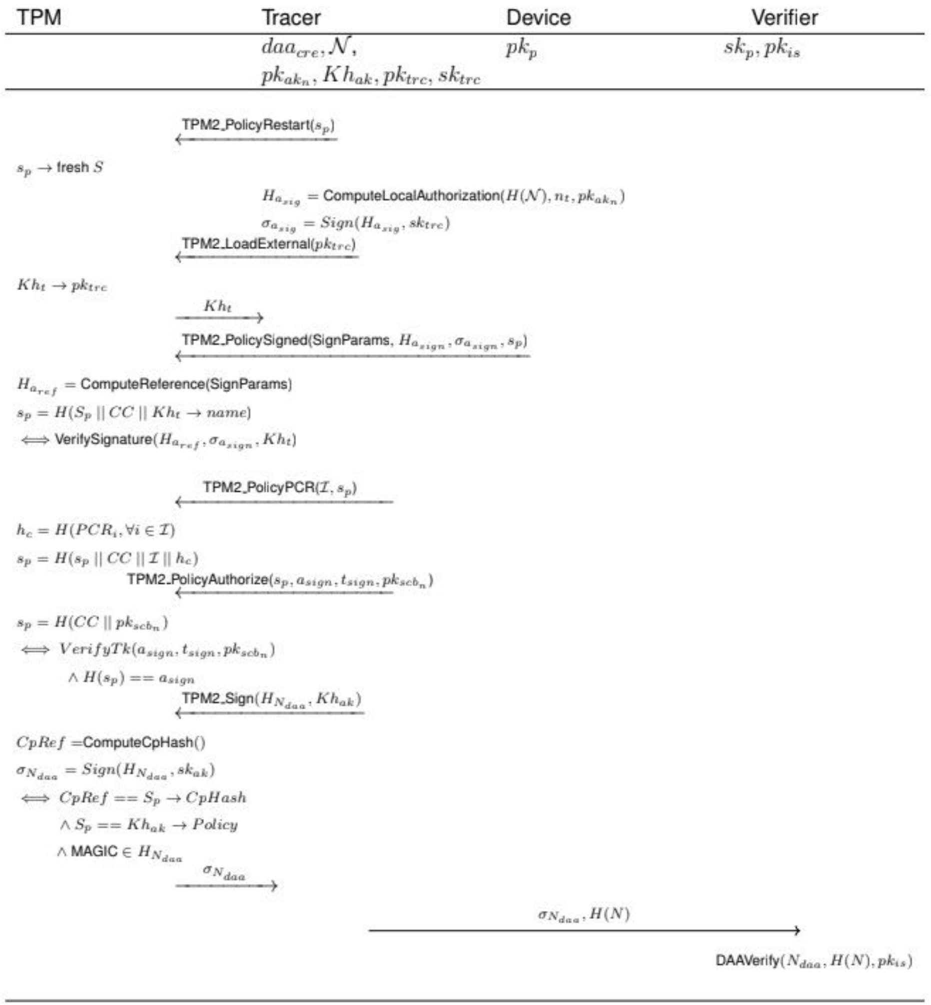
CIV Breakdown

- The service lets the CIV Verifier (through its Attestation Agent) request CIV traces generation
- In turn, the Prover tracer computes the hashes over all loaded libraries and programs in the edge device
- Finally, the tracer signs the generated traces and sends them to the Prover's TPM-based Wallet for verification and signing with the produced AK


TPM
Tracer
Device
Verifier
sk_{ak}
daa_{cre}, pk_{ak_n}, Kh_{ak}, pk_{trc}, sk_{trc}, t_{comm}, t_{sign}, a_{sign}, a_{comm}
pk_p
sk_p
$n_p \leftarrow Policy_{nonce}$
$Sign(H(n_p), sk_p)$
$\sigma_n, n_p, Property \in Policy$
Verify$(H(n_p), pk_p)$
$n_p, Property$
$\mathcal{N}$ = Trace$(Property)$
TPM2_StartAuthSession()
$s_p \rightarrow$ fresh S
$n_t \leftarrow \{0,1\}^n$
n_t, s_p
TPM2_PolicyCommandCode(CC_Commit, s_p)
$s_p = H(s_p \,||\, CC \,||\, CC\_Commit)$
TPM2_PolicyAuthorize$(s_p, a_{comm}, t_{comm}, pk_{scb_n})$
$s_p = H(CC \,||\, pk_{scb_n})$
$\iff VerifyTk(a_{comm}, t_{comm}, pk_{scb_n})$
$\wedge H(s_p) == a_{comm}$
TPM2_Commit$(Points)$
CD =ComputeCommitData)
$\iff CC == CC\_Commit$
$\wedge S_p == pk_{trc} \rightarrow Policy$
CD
N_{daa} =PrepareSign$(daa_{cre}, CD, H(N))$
TPM2_Hash(N_{daa})
$H_{N_{daa}} = H(N_{daa})$
$H_{N_{daa}}$

CIV Breakdown (2)

- The AK can only sign upon approval from the tracer, as well as holding correct PCR values
- In order to allow this, the tracer must sign a nonce
- This is done by starting a session and sending the nonce to the tracer. The tracer then signs the nonce and returns it
- The client then executes PolicyPCR in the session, to verify that we are in a correct state
- After this the tracer's public key is loaded into the TPM and policySigned with the signed nonce and the loaded key is executed
- If this is a success, the PolicyOR command can be executed successfully (as we satisfy the signing policy), and policyAuthorize can correctly be executed with the authorization ticket



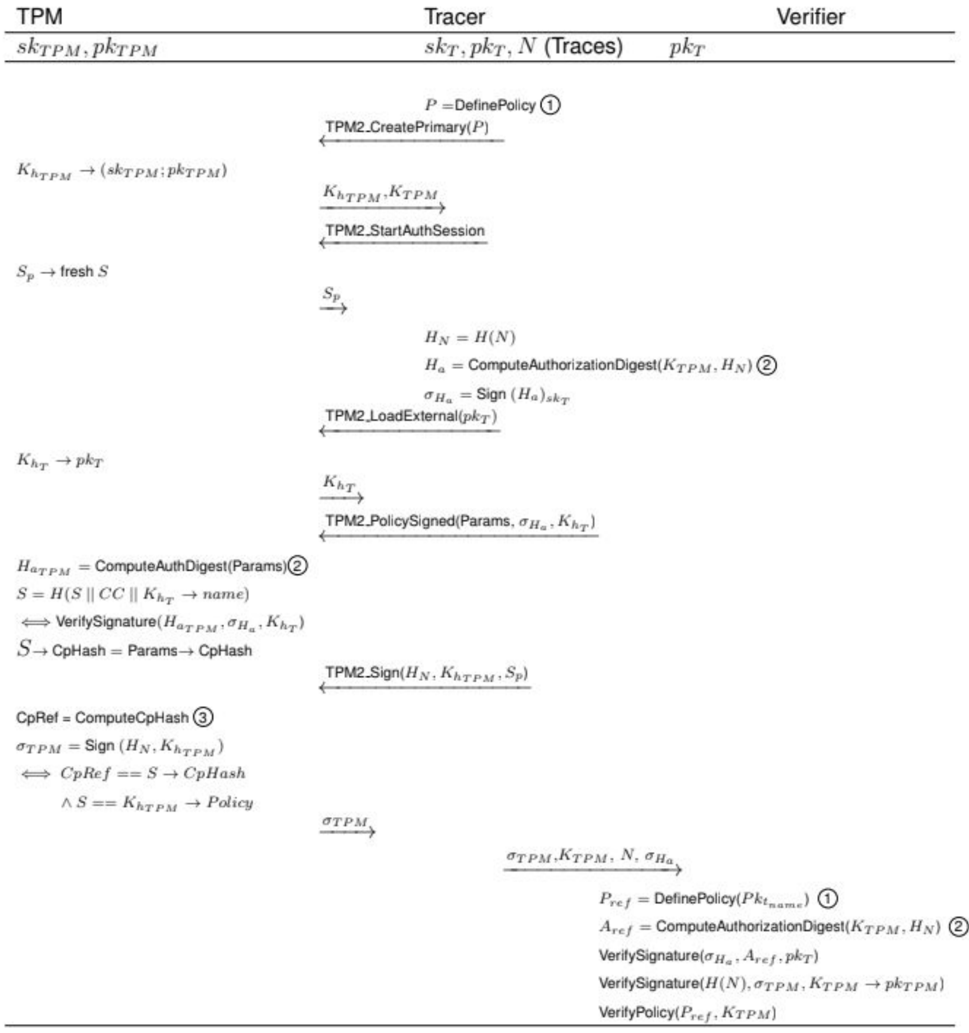
TPM
Tracer
Device
Verifier
$daa_{cre}, \mathcal{N},$
$pk_{ak_n}, Kh_{ak}, pk_{trc}, sk_{trc}$
pk_p
sk_p, pk_{is}
TPM2_PolicyRestart(s_p)
$s_p \rightarrow$ fresh S
$H_{a_{sig}}$ = ComputeLocalAuthorization($H(\mathcal{N}), n_t, pk_{ak_n}$)
$\sigma_{a_{sig}} = Sign(H_{a_{sig}}, sk_{trc})$
TPM2_LoadExternal(pk_{trc})
$Kh_t \rightarrow pk_{trc}$
Kh_t
TPM2_PolicySigned(SignParams, $H_{a_{sign}}, \sigma_{a_{sign}}, s_p$)
$H_{a_{ref}}$ = ComputeReference(SignParams)
$s_p = H(S_p \,||\, CC \,||\, Kh_t \rightarrow name)$
$\Longleftrightarrow$ VerifySignature($H_{a_{ref}}, \sigma_{a_{sign}}, Kh_t$)
TPM2_PolicyPCR($\mathcal{I}, s_p$)
$h_c = H(PCR_i, \forall i \in \mathcal{I})$
$s_p = H(s_p \,||\, CC \,||\, \mathcal{I} \,||\, h_c)$
TPM2_PolicyAuthorize($s_p, a_{sign}, t_{sign}, pk_{scb_n}$)
$s_p = H(CC \,||\, pk_{scb_n})$
$\Longleftrightarrow VerifyTk(a_{sign}, t_{sign}, pk_{scb_n})$
$\wedge H(s_p) == a_{sign}$
TPM2_Sign($H_{N_{daa}}, Kh_{ak}$)
$CpRef$ =ComputeCpHash()
$\sigma_{N_{daa}} = Sign(H_{N_{daa}}, sk_{ak})$
$\Longleftrightarrow CpRef == S_p \rightarrow CpHash$
$\wedge S_p == Kh_{ak} \rightarrow Policy$
$\wedge$ MAGIC $\in H_{N_{daa}}$
$\sigma_{N_{daa}}$
$\sigma_{N_{daa}}, H(N)$
DAAVerify($N_{daa}, H(N), pk_{is}$)

Integrity of Tracer / Authentication of Traces

- **The Tracer is responsible for continuously monitoring the processes executed in the device it belongs to, and collects information that is required in the context of the attestation schemes.**
- **This information can include control flow graphs used in Control-Flow Attestation (CFA), and hashes of configuration properties used in Configuration Integrity Verification (CIV).**
- **The Tracer is executed as a user space program and needs to be added to our Trusted Computing Base.**
  - *In order to prove the validity of the measurements that the TPM receives from the Tracer and uses in the context of the implemented attestation protocols, we employ a Pre-Installed Key*
  - *This key will be used to send signed traces to the Verifier, who is responsible for verifying their integrity.*
  - *Protocol securing the integrity of the reported traces. Protects against replay attacks and impersonation attacks, and ensures the integrity of the traces during correct protocol execution.*

Traces Integrity && Authentication

- The TPM shares with the Tracer the Attestation Key’s name.
- For every invocation of the Tracer the TPM creates a fresh policy session and shares the session nonce.
- The Tracer then calculates an authorization digest which includes the session nonce, the Attestation Key’s name and the hashed traces.
- Tracer Signs the authorization digest with its private key.
- The TPM loads the Tracer’s Public Key and executes TPM2_policySigned with the policy session that was created specifically for this challenge.
- If the correct key was used to sign the Authorization Digest, then the session digest should now match the policy digest of the TPM key
- We can now use the AK to sign data from the tracer, for verification purposes


TPM
Tracer
Verifier
sk_{TPM}, pk_{TPM}
sk_T, pk_T, N (Traces)
pk_T
$P =$ DefinePolicy ①
TPM2_CreatePrimary(P)
$K_{h_{TPM}} \rightarrow (sk_{TPM}; pk_{TPM})$
$K_{h_{TPM}}, K_{TPM}$
TPM2_StartAuthSession
$S_p \rightarrow$ fresh S
S_p
$H_N = H(N)$
$H_a =$ ComputeAuthorizationDigest(K_{TPM}, H_N) ②
$\sigma_{H_a} =$ Sign $(H_a)_{sk_T}$
TPM2_LoadExternal(pk_T)
$K_{h_T} \rightarrow pk_T$
K_{h_T}
TPM2_PolicySigned(Params, σ_{H_a}, K_{h_T})
$H_{a_{TPM}} =$ ComputeAuthDigest(Params)②
$S = H(S \mid\mid CC \mid\mid K_{h_T} \rightarrow name)$
$\iff$ VerifySignature$(H_{a_{TPM}}, \sigma_{H_a}, K_{h_T})$
$S \rightarrow$ CpHash = Params$\rightarrow$ CpHash
TPM2_Sign$(H_N, K_{h_{TPM}}, S_p)$
CpRef = ComputeCpHash ③
$\sigma_{TPM} =$ Sign $(H_N, K_{h_{TPM}})$
$\iff CpRef == S \rightarrow CpHash$
$\wedge S == K_{h_{TPM}} \rightarrow Policy$
σ_{TPM}
$\sigma_{TPM}, K_{TPM}, N, \sigma_{H_a}$
$P_{ref} =$ DefinePolicy$(Pk_{t_{name}})$ ①
$A_{ref} =$ ComputeAuthorizationDigest(K_{TPM}, H_N) ②
VerifySignature$(\sigma_{H_a}, A_{ref}, pk_T)$
VerifySignature$(H(N), \sigma_{TPM}, K_{TPM} \rightarrow pk_{TPM})$
VerifyPolicy(P_{ref}, K_{TPM})

Security Properties

Replay attacks	not possible by an adversary, because every time PolicySigned is executed, it requires the Session Nonce. The attempt of a replay attack would be detected when the TPM calculates the reference value of the authorization digest
Traces integrity	If the TSS changes the Tracer's values that have been transmitted, this change would be detected during the verification phase, since the signatures would not match the ones from the provided traces
Tracer's impersonation	not possible, as the adversary cannot access the Tracer's private key and it would be unable to provide a valid signature over the Authorization Digest

Impersonati-on Attacks

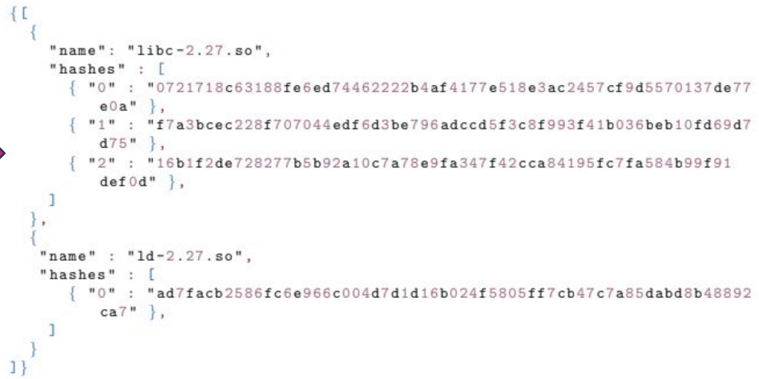
Replay attacks

Traces integrity

Tracer's CIV Invocation

→ The tracer infers the location of each page loaded in memory for the libraries, reads the content of the page, and computes a hash of 64 bytes using the SHA-256 algorithm.
→ The produced trace thus contains a set of hashes representing the in-memory configuration of the device

Representation



```
{[
  {
    "name": "libc-2.27.so",
    "hashes" : [
      { "0" : "0721718c63188fe6ed74462222b4af4177e518e3ac2457cf9d5570137de77
         e0a" },
      { "1" : "f7a3bcec228f707044edf6d3be796adccd5f3c8f993f41b036beb10fd69d7
         d75" },
      { "2" : "16b1f2de728277b5b92a10c7a78e9fa347f42cca84195fc7fa584b99f91
         def0d" },
    ]
  },
  {
   "name" : "ld-2.27.so",
   "hashes" : [
      { "0" : "ad7facb2586fc6e966c004d7d1d16b024f5805ff7cb47c7a85dabd8b48892
         ca7" },
    ]
  }
]}
```

THANKS

PROJECT-ASSURED.EU

@Project_Assured

ASSURED project is funded by the EU's Horizon2020
programme under Grant Agreement number 952697