STET

Split-Trust Encryption Tool
Attested Session Handling

https://github.com/GoogleCloudPlatform/stet

Keith Moyer - Google (kmoy@google.com)

2022-03-15

Background

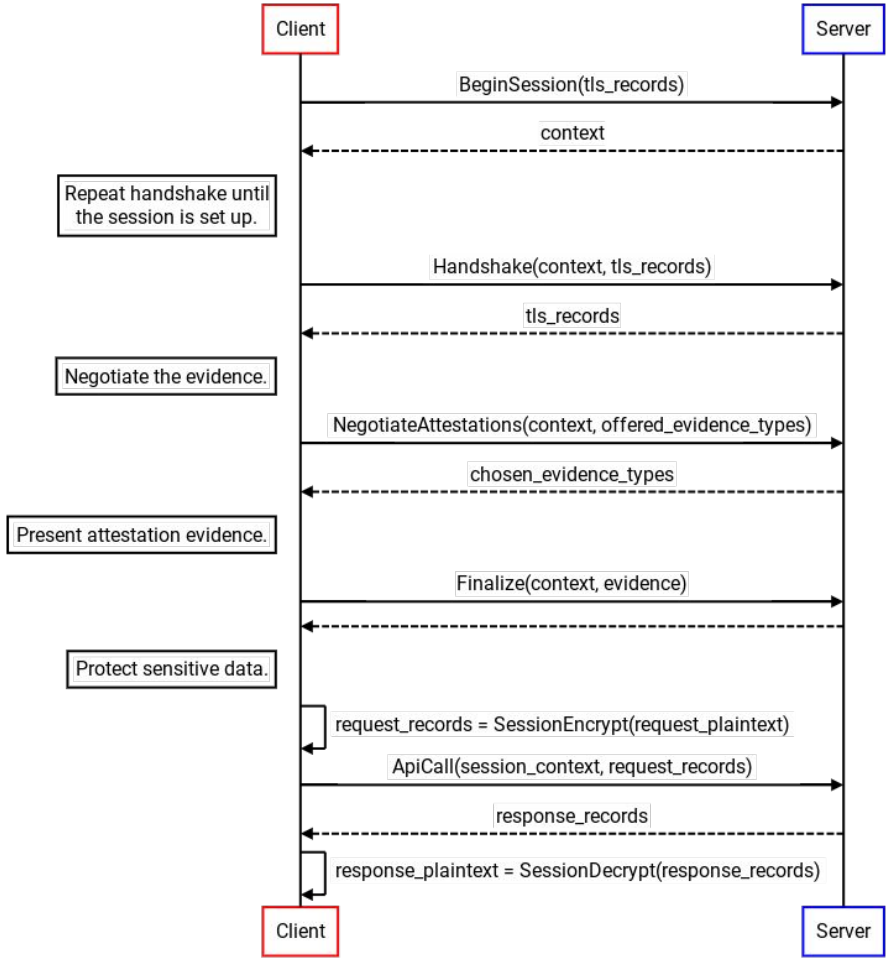
- Client-side encryption tool
  - DEK split into shares and wrapped with separate Key Encryption Keys (KEKs)
- KEKs managed by separate Key Management Systems (KMSs)
  - Alternatively, shares can be locally encrypted with asymmetric keys
- KMS has policy requiring attestation (and specific attestation claims)
  - Client is Attestor
  - Server is Relying Party (and Verifier)
- Developed as *pragmatic* solution
  - Probably has opportunities to incorporate other standards/approaches

Secure Session Requirements

- Operations, parameters, and results encrypted with end-to-end secure session
- Secure session only established if attestation evidence provided
- Must allow intermediate termination (load balancing, etc)
- Uses established protocols with only widely available library features

Session Establishment

Session Establishment



Session Establishment RPC Messages

```
message BeginSessionRequest {
  bytes tls_records = 1;
}

message BeginSessionResponse {
  bytes session_context = 1;
  bytes tls_records = 2;
}

message HandshakeRequest {
  bytes session_context = 1;
  bytes tls_records = 2;
}

message HandshakeResponse {
  bytes tls_records = 1;
}

enum AttestationEvidenceType {
  NULL_ATTESTATION = 1       ;
  TPM2_QUOTE = 2;
  TCG_EVENT_LOG = 3;
}
```

```
message NegotiateAttestationRequest {
  bytes session_context = 1;
  bytes offered_evidence_types_records = 2;
}

message NegotiateAttestationResponse {
  bytes required_evidence_types_records = 1;
}

message FinalizeRequest {
  bytes session_context = 1;
  bytes attestation_evidence_records = 2;
}

message FinalizeResponse {}


message AttestationEvidenceTypeList {
  repeated AttestationEvidenceType types = 1;
}
```

TPM Attestation Serialization

From https://github.com/google/go-tpm-tools

```
message Attestation {
  // Attestation Key (AK) Public Area, encoded as a TPMT_PUBLIC
  bytes ak_pub = 1;
  // Quotes over all supported PCR banks
  repeated tpm.Quote quotes = 2;
  // TCG Event Log, encoded in the raw binary format.
  // Can be SHA-1 or crypto-agile.
  bytes event_log = 3;
  // Optional information about a GCE instance, unused outside of GCE
  GCEInstanceInfo instance_info = 4;
  // A TCG Canonical Event Log.
  bytes canonical_event_log = 5;
  // Attestation Key (AK) Certificate, encoded as ASN.1 DER.
  // Optional.
  bytes ak_cert = 6;
  // Intermediate Certificates for verifying the AK Certificate, encoded as ASN.1 DER.
  // Optional.
  repeated bytes intermediate_certs = 7;
}
```

Attestation Binding

1. Raw TLS session established
2. Attestation evidence types negotiated
3. 32-byte TLS Exported Keying Material (EKM) generated
    - Using label "EXPERIMENTAL Google Confidential Computing Client Attestation 1.0"
    - Used as proof of possession of TLS session key, commonly supported in TLS libraries
    - Client and server can independently generate, but TLS MitM cannot
4. Attestation evidence is requested on client
    - Currently only supports TPM evidence
    - Using hash(“TLSAttestationV1”|<EKM>) as nonce/data to include in evidence
5. Attestation evidence is sent over TLS session
    - Server verifies attestation evidence, including expected EKM (session aborted if invalid)
    - Server associates attestation evidence or claims to session
6. Session finalized and secured RPCs can now be sent, encrypted by session
    - Server evaluates appraisal policy against evidence/claims for each operation
7. Session destroyed

Questions?

Backup

Secure RPCs

- FooRequest and FooResponse are typical RPC request and response messages.
- Serialized protos are TLS session-encrypted and sent as tls_records.
- In the case of STET, we have ConfidentialWrap and ConfidentialUnwrap.

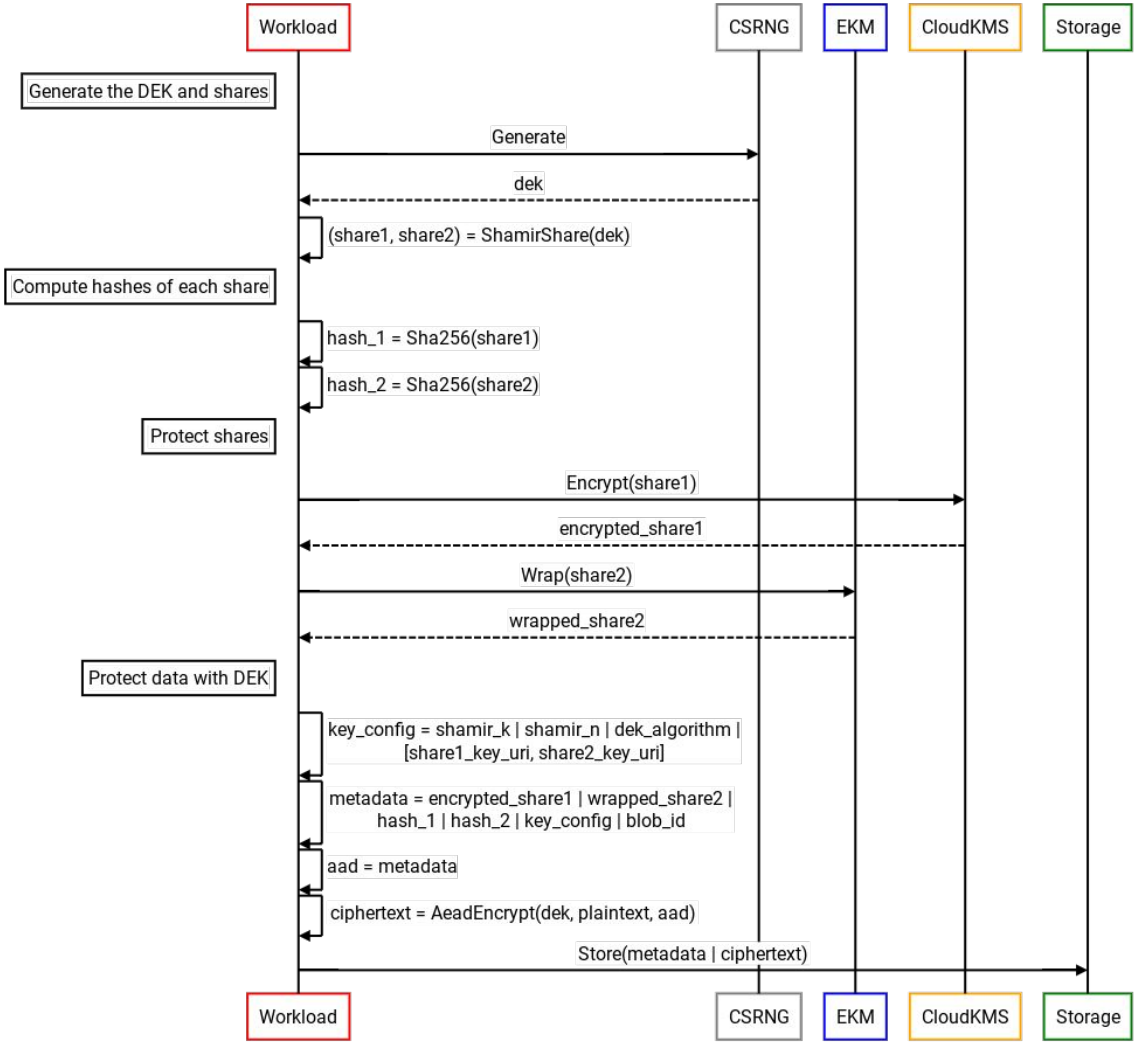
```
message FooRequest {
  string ping = 1;
}

message FooResponse {
  string pong = 1;
}
```

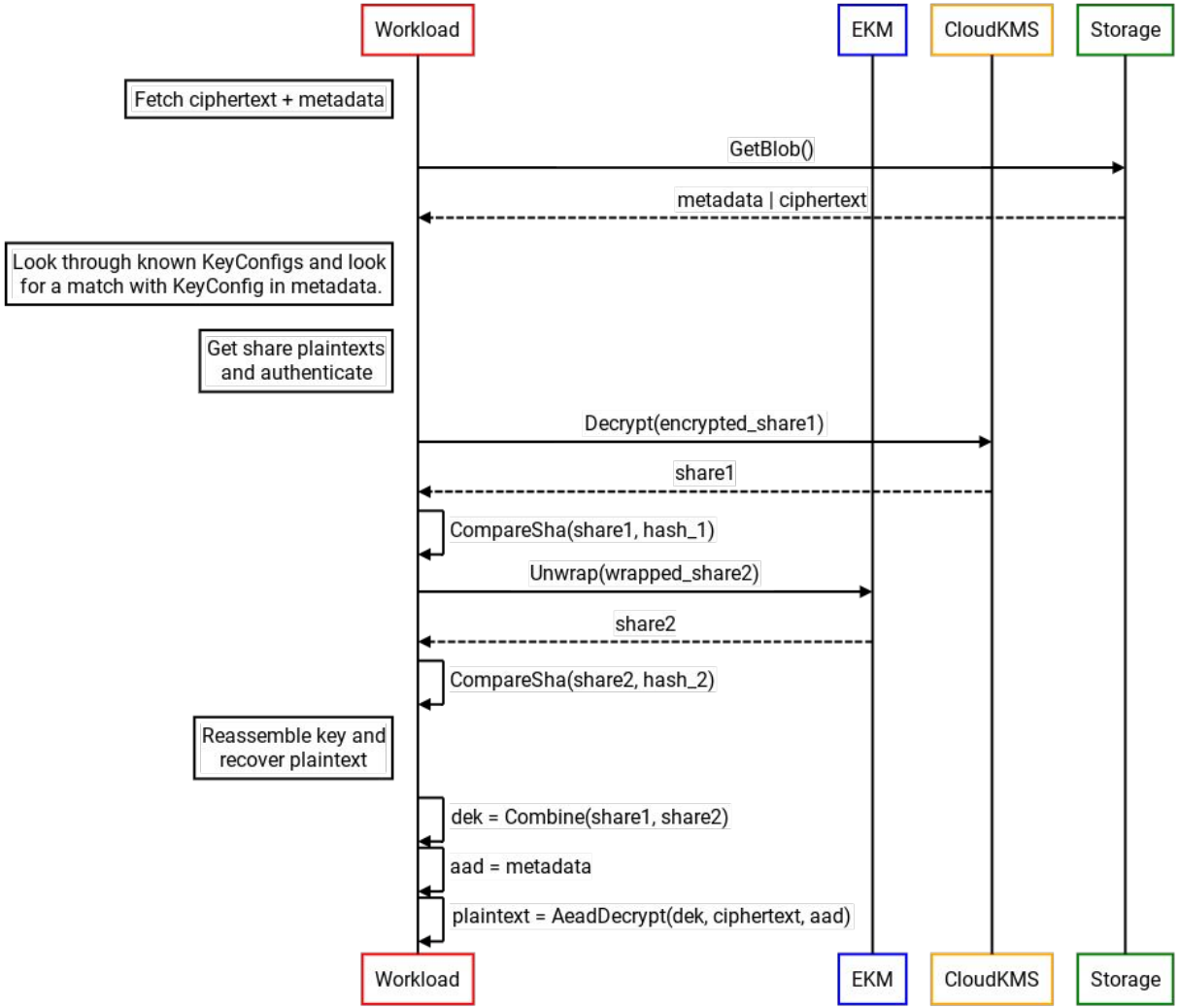
```
message ConfidentialFooRequest {
  bytes session_context = 1;
  bytes tls_records = 2;
}

message ConfidentialFooResponse {
  bytes tls_records = 1;
}
```

Encryption


Workload
CSRNG
EKM
CloudKMS
Storage
Generate the DEK and shares
Generate
dek
(share1, share2) = ShamirShare(dek)
Compute hashes of each share
hash_1 = Sha256(share1)
hash_2 = Sha256(share2)
Protect shares
Encrypt(share1)
encrypted_share1
Wrap(share2)
wrapped_share2
Protect data with DEK
key_config = shamir_k | shamir_n | dek_algorithm | [share1_key_uri, share2_key_uri]
metadata = encrypted_share1 | wrapped_share2 | hash_1 | hash_2 | key_config | blob_id
aad = metadata
ciphertext = AeadEncrypt(dek, plaintext, aad)
Store(metadata | ciphertext)
Workload
CSRNG
EKM
CloudKMS
Storage

Decryption


Workload
EKM
CloudKMS
Storage
Fetch ciphertext + metadata
GetBlob()
metadata | ciphertext
Look through known KeyConfigs and look for a match with KeyConfig in metadata.
Get share plaintexts and authenticate
Decrypt(encrypted_share1)
share1
CompareSha(share1, hash_1)
Unwrap(wrapped_share2)
share2
CompareSha(share2, hash_2)
Reassemble key and recover plaintext
dek = Combine(share1, share2)
aad = metadata
plaintext = AeadDecrypt(dek, ciphertext, aad)
Workload
EKM
CloudKMS
Storage