Hyperledger Avalon Cryptography

Dan Anderson, Intel Corporation
Hyperledger Avalon Developer's Forum, August 13, 2020

Agenda

- Avalon Cryptography SDK Overview
- Avalon Work Order Flow
- Cryptographic Algorithms
- Implementations

Avalon's Cryptographic Algorithms

Function	Algorithm	Keysize (bits)	Post-quantum crypto resistant?	Blockchain legacy algorithm?	Comments and usage
Digital signature	ECDSA (curve SECP256k1)	256	No	Yes	Signs WO response digest and worker RSA key
Asymmetric (public key) encryption	RSA (OAEP padding)	3072	No	No	WO AES keys **XXXXXX**
Authenticated, secret key encryption	AES (GCM mode)	256	Yes	No	96b ***unique*** IV, 256b auth tag; WO RR data
Digest	SHA	256	Yes	Yes	WO RR digest
	Source: `common/cpp/crypto/README.md`				**Legend:** WOs = Work orders RR = request & response

Avalon Cryptography SDK: Headers and Examples

- SDK available in C++ and Python
  - Implemented in C++ (OpenSSL and Mbed TLS) and Python (PyCryptodome)
  - Other languages possible (or use another library—these are standard algorithms)
- C++ source, headers, and examples
  - `common/cpp/crypto/[/mbedtls/` , `common/cpp/tests/` ,
  - `listener/avalon_listener/` , `tc/sgx/trusted_worker_manager/`
- Python source and examples
  - `common/crypto_utils/avalon_crypto_utils/` , `common/crypto_utils/tests/` , `tests/test_*.py`
- Find more examples in search box at https://github.com/hyperledger/avalon

Work Order Flow

- 1. Requester creates Work Order
- 2. Worker (in TEE enclave)
- 3. Requester Receiving Response

Avalon Work Order Flow (1 of 3): Requester (aka client) creates Work Order

- Generates 1-time AES-GCM-256 key (*SEK*)
- Encrypts Work Order (WO) with *SEK*
- Encrypts hash of request data with *SEK*
- Encrypts *SEK* with enclave's RSA public key
- (optional) signs hash of request data with verification key

Avalon Work Order Flow (2 of 3): Worker (in TEE enclave)

- Decrypts requester’s 1-time *SEK* with enclave's RSA private key
- Decrypts WO request data
- Calculates SHA-256 hash of request
- Decrypts SHA-256 hash in request and compares it with above
- (optional) verifies hash with requester's public key
- Process WO
- Encrypt WO response data with *SEK*
- Signs hash of response with enclave's private key

Avalon Work Order Flow (3 of 3): Requester Receiving Response

- Retrieves locally-stored worker signing key
- Verifies the WO response signature generated by worker
- Retrieves its own (locally-stored) requester *SEK*
- Decrypts output data items from WO response

- For details see “High Level Execution Flow” in *Hyperledger Avalon Architecture Overview*
  https://github.com/hyperledger/avalon/blob/master/docs/avalon-arch.pdf

Cryptographic Algorithms

- Base64 encode/decode
- SHA-256 hashing
- ECDSA (curve SECP256K1) signatures
- RSA-3072 OAEP signature verification

- AES-GCM 256 secret key
- RSA-3072 public key
- CA chain verification

Base64 encode/decode

- Encodes binary data in printable characters: A-Za-z0-9+/=
  - 0 to 2 "**`=`**" padding characters at end, depending on remaining bit length
- "**`Hyperledger`**" (length 11) encodes as "**`SHlwZXJsZWRnZXI=`**" (len 16)
- Usually used to encode BER/DER format certificates and keys into PEM format
  - delimited by "**`-----BEGIN`**" and "**`-----END`**" lines
  - Avalon uses:

```
-----BEGIN CERTIFICATE-----      -----BEGIN RSA PRIVATE KEY-----
-----BEGIN PUBLIC KEY-----       -----BEGIN RSA PUBLIC KEY-----
-----BEGIN EC PRIVATE KEY-----
```

Hashing with SHA-256

- Avalon hashes are 32 byte SHA-256 digests
- Work order/WO response hash is concatenation of:
  - Nonce, work order ID + worker ID + workload ID + requester ID
- Digital Signatures sign the hash of data, not the data itself

Signatures with ECDSA (curve SECP256K1)

- ECDSA signatures using curve SECP256K1 is a Bitcoin/Ethereum/Blockchain convention
- API assumes message is already hashed with SHA-256 (pass hash to API)
- ECDSA signature is the EC point expressed as two 256 bit (X,Y) coordinates
- Signature 71 bytes when DER-encoded in hexadecimal (Bitcoin convention)
- Example signature:
  `30450221008e6b04abffea7dab1d2c6190619096262e567fa9f94be337953aab`
  `8742158d1c022034bd23799bc27308ce645191c43c16d5fb767e6cb5ab002442`
  `7194cbba59783c`
- Keys in PEM format
  - **`-----BEGIN EC PRIVATE KEY-----`** and **`-----BEGIN PUBLIC KEY-----`**

Signature verification with RSA-3072

- Verifies RSA-3072 (384 byte) signatures
- OAEP padding (not PKCS#1)
- Given a
  - cert (RSA public key),
  - message, and
  - signature (not hash),
  - then verify the message was signed by the corresponding RSA secret key
- API internally performs SHA-256 hash on message
  - since it's the hash that is signed
- Avalon verifies RSA signatures, but does use RSA to sign messages

Secret key encryption with AES-GCM 256

- AES secret (aka symmetric) key encryption
- GCM mode provides an Authentication tag (aka MAC); 16 bytes
  - appended to encrypted output; verifies encrypted text not altered
- IV ***must*** be unique (sometimes IV is called a nonce); 12 bytes
  - Repeat: IV ***must*** be unique
  - IV prevents “codebook” (aka “dictionary”) attacks
  - All hope is lost if IVs are repeated (not unique)
  - Two APIs: one API assumes IV is prepended to encrypted text
    - Another API has IV passed as a separate parameter
- Keys are 32 bytes and are represented as a 64 hex character string

Public Key encryption with RSA-3072 (OAEP padding)

- 384 byte signatures
- OAEP padding (not PKCS#1)
- Keys are PEM encoded; begin with `-----BEGIN PUBLIC KEY-----` and `-----BEGIN PRIVATE KEY-----`
  - Also accepted (but not created) is: `-----BEGIN RSA PRIVATE KEY-----`

Certificate Authority (CA) chain verification

- Verified root of trust for a cert
  - In our case a RSA public key cert
- Given a cert “A” and a CA cert “C” verifies cert “A” is valid
  - (valid = signed by “C”)
- Intermediate CA certs allowed. E..g.
  - Cert “A” signed by CA cert “B1” signed by CA cert “B2” signed by CA cert “C”
- Cert “C” and “B1”, “B2”, . . . all must be CA certs

Cryptographic Implementations

- OpenSSL
- Mbed TLS
- PyCryptodome

Implementation: OpenSSL

- Current implementation for C++
- Advantages:
  - Fast (assembly optimizations)
  - Popular (many examples out there)
  - Standards based
- Disadvantage: large footprint (TCB)
  - Convoluted and poorly-documented API

Implementation: Mbed TLS

- Mbed TLS sponsored by ARM
- Formerly known as PolarSSL
- Advantages
  - Used by OpenEnclave SDK (Microsoft)
  - Small footprint, portable (C)
  - Good documentation with (limited) examples
  - Standards base
- Disadvantages
  - Slow (no assembly)
  - Lack of command line utilities (can substitute with OpenSSL CLIs)

Implementation: PyCryptodome

- Native Python crypto (no C/C++ library dependency)
- For Python, Avalon previously used OpenSSL with C++ swig wrapper
- Advantages
  - Popular, well-documented, examples available
  - Standards based
- Disadvantages
  - No assembly optimization

Questions and Comments?

- Video will be posted at https://wiki.hyperledger.org/display/avalon/Meetings
- Chat channel at https://chat.hyperledger.org/channel/avalon
- Mailing list at https://lists.hyperledger.org/g/avalon