

# Community Call

# Ockam Vault Interface

2020.06.24 | Mark Mulrooney

# Agenda

- Ockam Vault Overview & API
- How Ockam Channels use an Ockam Vault
- Demos
- Types of Vaults
- Q & A

# Ockam Vault Interface

- Ockam protocols depend on a variety of cryptographic building blocks to achieve secure end-to-end communication.
- Depending on the devices, these building blocks may be provided by:
  - A software implementation
  - Cryptographically capable hardware component : Trusted Execution Environment, Trusted Platform Module,s, Secure Enclaves, Hardware Security Modules, Crypto Modules, etc.
  - A mix of both software and hardware.
- In order to support a wide variety of cryptographically capable hardware and software libraries, the Vault Interface is defined to provide a set of common cryptographic building block functions to allow the Ockam protocols to call these cryptographic building block functions without any knowledge of the underlying implementation.

```
// BearSSL
br_hmac_drbg_generate(br_hmac_drbg_context *ctx,
                      void *out,
                      size_t len);

// Microchip CryptoAuthLib
atcab_random (uint8_t *rand_out);

// Optiga Trust X
optiga_crypt_random(optiga_rng_types_t rng_type,
                   uint8_t * random_data,
                   uint16_t random_data length);
```

[illegible]

# Ockam Vault Interface

- Types of cryptography currently used or being experimented with in Ockam Vault
  - Public Key Encryption
    - Curve25519
    - NIST P-256
  - Cryptographic Hash Functions
    - SHA-256
  - Message Authentication Code
    - HMAC-SHA256
  - Symmetric Key Encryption
    - AES-GCM-128
    - AES-GCM-256

# Ockam Vault API

- All functions return an **ockam\_error\_t** type.
- A vault object must be passed into to every function. The object contains a vtable to point to the specific implementation that was initialized.
- Any functions that require data that should not be exposed (private key, shared secrets, AES keys) use the abstract **ockam\_vault\_secret\_t** object.
- There is a complementary decrypt function to the encrypt function.

```
ockam_error_t ockam_vault_random_bytes_generate(ockam_vault_t* vault,  
                                                  uint8_t* buffer,  
                                                  size_t buffer_size);  
  
ockam_error_t ockam_vault_sha256(ockam_vault_t* vault,  
                                  const uint8_t* input,  
                                  size_t          input_length,  
                                  uint8_t*       digest,  
                                  size_t          digest_size,  
                                  size_t*        digest_length);  
  
ockam_error_t ockam_vault_ecdh(ockam_vault_t*      vault,  
                                ockam_vault_secret_t* privatekey,  
                                const uint8_t*       peer_publickey,  
                                size_t               peer_publickey_length,  
                                ockam_vault_secret_t* shared_secret);  
  
ockam_error_t ockam_vault_hkdf_sha256(ockam_vault_t*      vault,  
                                        ockam_vault_secret_t* salt,  
                                        ockam_vault_secret_t* input_key_material,  
                                        uint8_t              derived_outputs_count,  
                                        ockam_vault_secret_t* derived_outputs);  
  
ockam_error_t ockam_vault_aead_aes_gcm_encrypt(ockam_vault_t*      vault,  
                                                 ockam_vault_secret_t* key,  
                                                 uint16_t           nonce,  
                                                 const uint8_t*     additional_data,  
                                                 size_t             additional_data_length,  
                                                 const uint8_t*     plaintext,  
                                                 size_t             plaintext_length,  
                                                 uint8_t*         ciphertext_and_tag,  
                                                 size_t             ciphertext_and_tag_size,  
                                                 size_t*            ciphertext_and_tag_length);
```

# Ockam Vault API

- Init functions are **not** part of the Vault API
- Initialization functions are part of the implementation and must be called directly
- Attributes struct will vary depending on the implementation. Default vault is a fairly simple init structure. Hardware devices like Microchip's ATECC608A requires details about the I2C bus in addition to memory and other settings.



```
#include "ockam/error.h"
#include "ockam/memory.h"
#include "ockam/random.h"
#include "ockam/vault.h"

#include "memory/stdlib/stdlib.h"
#include "random/urandom/urandom.h"
#include "vault/default/default.h"

ockam_vault_t vault = { 0 };
ockam_memory_t memory = { 0 };
ockam_random_t random = { 0 };

ockam_vault_default_attributes_t vault_attributes =
{
    .memory = &memory,
    .random = &random
};

// Memory and Random modules must be initialized first

error = ockam_vault_default_init(&vault,
                                &vault_attributes);
if (error != OCKAM_ERROR_NONE) {
    goto exit;
}
```

# Ockam Vault Secret

- In addition to providing a common interface to access cryptographic building blocks, the Vault Interface also provides an abstract called Secrets.
- Depending on the type of Vault implementation, keys may be stored in a variety of mediums. The `ockam_vault_secret_t` type allows an abstraction on the key data so the Ockam protocols do not need to understand if key data is stored in buffers, files or hardware.
- Three possibilities for secret keys in Ockam Vault:
  - Full Hardware Abstraction: Keys never leave the external hardware implementation.
  - Software Library: Every function returns key data into buffers.
  - Mixed Implementation: Hardware only stores some keys, buffers are used for other keys.

```
br_ec_keygen(const br_prng_class ** rng_ctx,
             const br_ec_impl * impl,
             br_ec_private_key * sk,
             void * kbuf,
             int curve);

atcab_genkey(ATCADevice device,
             uint16_t key_id,
             uint8_t *public_key);

optiga_crypt_ecc_generate_keypair(optiga_ecc_curve_t curve_id,
                                  uint8_t key_usage,
                                  bool_t export_private_key,
                                  void * private_key,
                                  uint8_t * public_key,
                                  uint16_t * public_key_length);
```



```
ockam_vault_secret_generate(ockam_vault_t* vault,
                             ockam_vault_secret_t* secret,
                             const ockam_vault_secret_attributes_t* attributes);
```

# Ockam Vault Secret API

- The `ockam_vault_secret_t` object is defined similar to `ockam_vault_t`. The specifics of the implementation are hidden from the caller and instead are specific to a vault implementation.
- Many vault secret functions may have varying modes of operation depending on the specific vault implementation
  - May not be possible to import/export keys with external hardware.
  - Can not change elliptic curve secrets to buffers or AES
- Checking error return values for secrets will be imperative for successful use.

[illegible]



# How Ockam Channels use an Ockam Vault

- Ockam currently implements the XX key agreement pattern from the Noise Protocol Framework.
- XX key agreement is used to generate a secure channel between an Initiator and Responder
- The following cryptographic building blocks are implemented in every Vault implementation and required by the Ockam Protocols.
  - SHA-256
  - Secret Generate
  - Secret Public Key
  - ECDH
  - HKDF-SHA256
  - AES-GCM-128/256

# How Ockam Channels use an Ockam Vault

## SHA-256

- Throughout the XX handshake, a chained SHA-256 transcript hash is used to hash all messages exchanged during the key agreement. This guarantees that both initiator and responder saw the same data.
- Both the initiator and responder use SHA-256 and use the results as part of the validation of the handshake.
- The input is the message to be hashed, and can vary in size.
- The generated digest is always a 32-byte value. The **digest\_length** parameter should always be set to `OCKAM_VAULT_SHA256_DIGEST_LENGTH` (which is 32).

```
uint8_t input[] = "Hello World";
size_t input_length = 11;

uint8_t digest[OCKAM_VAULT_SHA256_DIGEST_LENGTH] = {0};
size_t digest_length = 0;

error = ockam_vault_sha256(&vault,
                           &input[0],
                           input_length,
                           &digest[0],
                           OCKAM_VAULT_SHA256_DIGEST_LENGTH,
                           &digest_length);

if(error != OCKAM_ERROR_NONE) {
    // Handle error
}

if(digest_length != OCKAM_VAULT_SHA256_DIGEST_LENGTH) {
    // Handle error
}
```

# How Ockam Channels use an Ockam Vault

## Secret Generate

- Goal is to generate an ephemeral key pair using Curve25519
- A null `ockam_vault_secret_t` must be created and passed in.
- Attributes struct informs vault:
  - Use Curve25519 elliptic curve
  - Key's use is for key agreement
  - It is an ephemeral key pair
    - Some vaults may use more temporary memory locations for ephemeral key pairs.
  - Length is ignored, size of a Curve25519 key is already known.
    - Length field is used for secret buffers and AES keys.

```
ockam_vault_secret_t secret = { 0 };
ockam_vault_secret_attributes_t attributes = { 0 };

attributes.length = 0;
attributes.type = OCKAM_VAULT_SECRET_TYPE_CURVE25519_PRIVATEKEY;
attributes.purpose = OCKAM_VAULT_SECRET_PURPOSE_KEY_AGREEMENT;
attributes.persistence = OCKAM_VAULT_SECRET_EPHEMERAL;

error = ockam_vault_secret_generate(&vault, &secret, &attributes);
if (error != OCKAM_ERROR_NONE) {
    // Handle error
}
```

# How Ockam Channels use an Ockam Vault

## Secret Public Key

- After generating a Curve25519 key pair, the public key must be shared with the other party.
- A buffer of at least the public key length must be passed in.
- The size of the public key is pre-defined in **vault.h**.
- The amount of data copied into the provided buffer should match the size of the public key size. The length field returns the amount that was actually copied into the buffer.

```
uint8_t public_key[OCKAM_VAULT_CURVE25519_PUBLICKEY_LENGTH] = {0};
uint8_t public_key_length = 0;

error = ockam_vault_secret_publickey_get(&vault,
                                         &secret,
                                         &public_key,
                                         OCKAM_VAULT_CURVE25519_PUBLICKEY_LENGTH,
                                         &public_key_length);

if(error != OCKAM_ERROR_NONE) {
    // Handle error
}

if(public_key_length != OCKAM_VAULT_CURVE25519_PUBLICKEY_LENGTH) {
    // Handle error
}
```

# How Ockam Channels use an Ockam Vault

## ECDH

- Once the initiator has received the public key from the responder, the shared secret can be calculated.
- A new secret object is needed for the resulting shared secret.
- The **secret** parameter must contain a private key on the same curve as the **public\_key**.
  - Can't mix a Curve25519 private key with a P256 public key.
- The resulting shared secret will be the same value for the initiator and responder.

```
ockam_vault_secret_t shared_secret = {0};


error = ockam_vault_ecdh(&vault
                        &secret,
                        &public_key[0],
                        OCKAM_VAULT_CURVE25519_PUBKEY_LENGTH,
                        &shared_secret);

if(error != OCKAM_ERROR_NONE) {
    // Handle error
}
```

# How Ockam Channels use an Ockam Vault

## HKDF-SHA256

- After a shared secret has been generated, HKDF-SHA256 is used to expand the shared secret into multiple key values for AES.
- Before calling HKDF-SHA256, a salt value must be loaded into an `ockam_vault_secret_t` object using `ockam_vault_secret_import()`.
  - The salt value is defined by the Ockam protocol.
- The `shared_secret` parameter must contain the generated value from `ockam_vault_ecdh()`.
- The HKDF-SHA256 function outputs 32-byte value secrets. The caller specifies how many 32-byte values to generate. The number of outputs required is defined by the protocol.



```
ockam_vault_secret_t generated_secrets[2] = {0};

error = ockam_vault_hkdf_sha256(&vault,
                                &salt
                                &shared_secret,
                                2,
                                &generated_secrets[0]);

if(error != OCKAM_ERROR_NONE) {
    // Handle error
}
```

# How Ockam Channels use an Ockam Vault

## AES-GCM-128/256

- One of the generated secrets from HKDF is used as the key value for the AEAD AES GCM Encrypt.
  - The protocol dictates when to use each generated secret
  - The output type of HKDF is buffer. The function `ockam_vault_secret_type_set()` will tell vault how that buffer value should be used for AES.
- Additional data and a nonce is passed in for encryption that must also be used for the decrypt.
- The result of the encrypt operation is a ciphertext that matches the length of the input data **and** an additional 16 bytes of tag data appended to the ciphertext.

```
uint16_t nonce = 1;
char* additional_data = "some metadata that will be authenticated but not encrypted";
size_t additional_data_length = strlen(additional_data);
char* plaintext = "some data that will be encrypted";
size_t plaintext_length = strlen(plaintext);
size_t ciphertext_and_tag_size = plaintext_length + OCKAM_VAULT_AEAD_AES_GCM_TAG_LENGTH;
uint8_t* ciphertext_and_tag;
size_t ciphertext_and_tag_length;

error = ockam_memory_alloc_zeroed(&memory, &ciphertext_and_tag, ciphertext_and_tag_size);
if (error) {
    // Handle error
}

error = ockam_vault_secret_type_set(&vault,
    &generated_secret[1],
    OCKAM_VAULT_SECRET_TYPE_AES128_KEY);

if (error) {
    // Handle error
}

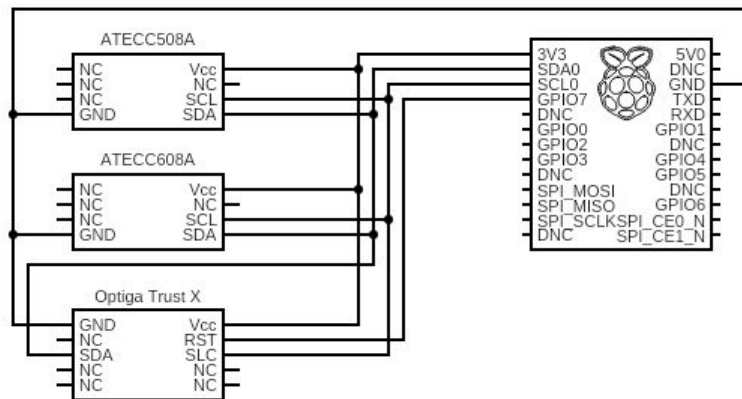
error = ockam_vault_aead_aes_gcm_encrypt(&vault,
    &generated_secret[1],
    nonce,
    (uint8_t*) additional_data,
    additional_data_length,
    (uint8_t*) plaintext,
    plaintext_length,
    ciphertext_and_tag,
    ciphertext_and_tag_size,
    &ciphertext_and_tag_length);

if (error) {
    // Handle error
}
```

# Demos

- Demos
  - Default Software Vault
    - Full software implementation
    - Uses BearSSL
  - Microchip ATECC608A
    - Full hardware implementation
    - Uses CryptoAuthLib
  - Microchip ATECC508A
    - Mix of software and hardware
    - Uses CryptoAuthLib
  - Optiga Trust X
    - Mix of software and hardware
    - Uses optiga-trust-x
- None of the demos were compiled on the Raspberry Pi. The C library makes use of cross compilers to be able to build for the Pi on any host.

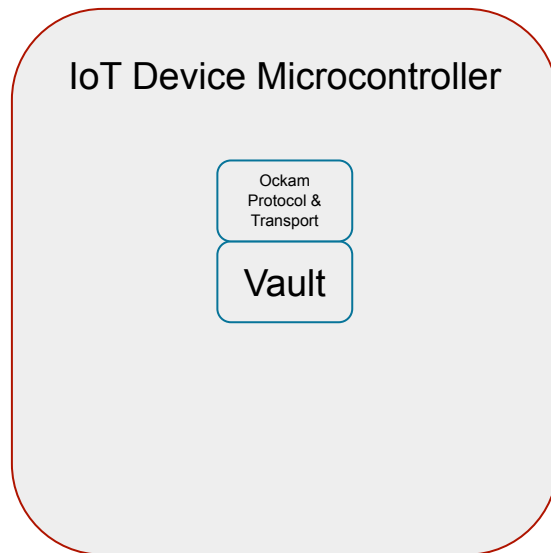
## Demo Hardware Layout





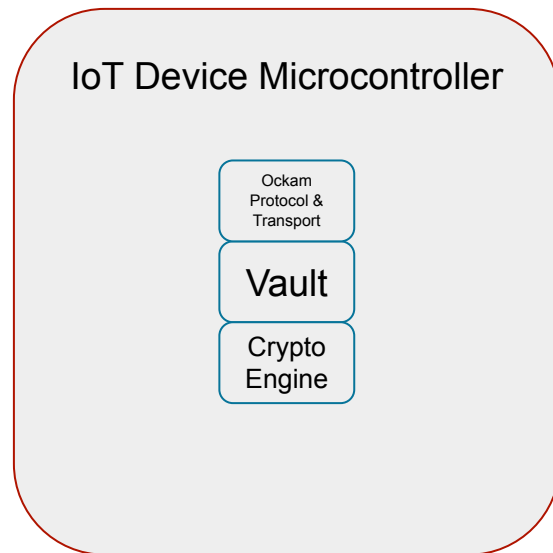
# Types of Vaults

- **Pure Software Vault**
  - Software library that contains all crypto functions
  - Keys stored in buffers/files
  - May be able to take advantage of optimized functions for processor architecture



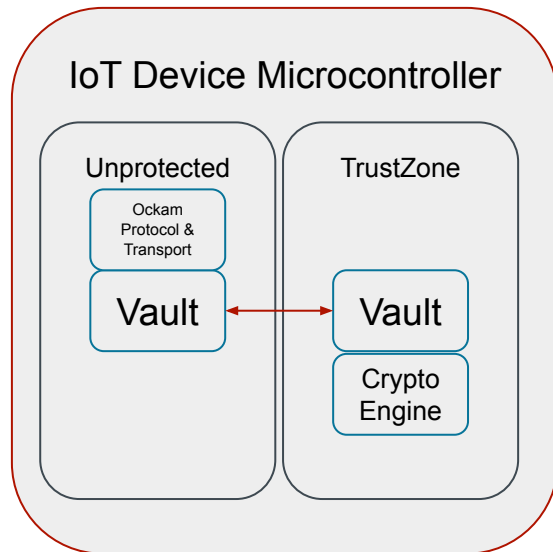
# Types of Vaults

- **Software Vault w/ Crypto Engine**
  - Software library that contains all crypto functions
  - Keys stored in buffers/files
  - Some or all crypto functions can be sped up by the onboard crypto engine



# Types of Vaults

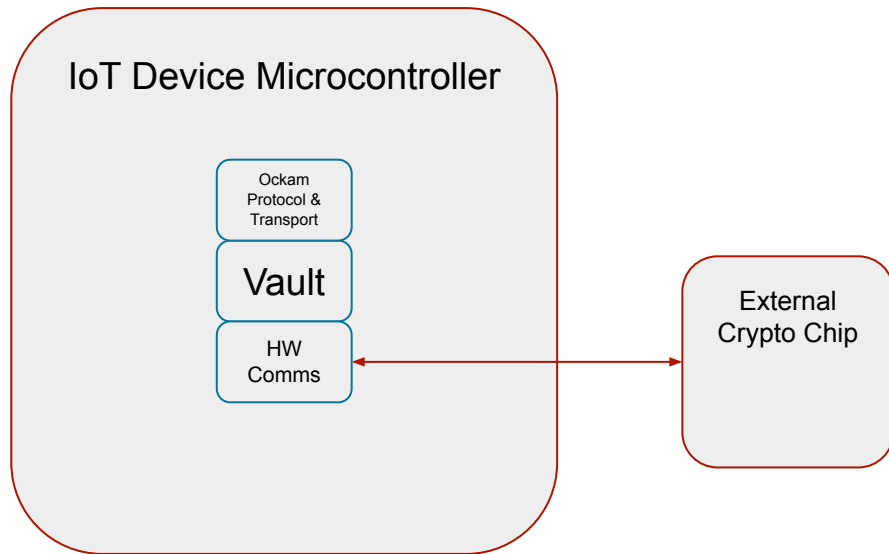
- **Software Vault w/ Crypto Engine and TrustZone protection**
  - Software library that contains all crypto functions. Operates an interface in the unprotected zone and sends/receives data to the TrustZone software via SMC or similar secure API call.
  - Keys stored behind TrustZone barrier.
  - Some or all crypto functions can be sped up by the onboard crypto engine



# Types of Vaults

- **Hardware Vault**

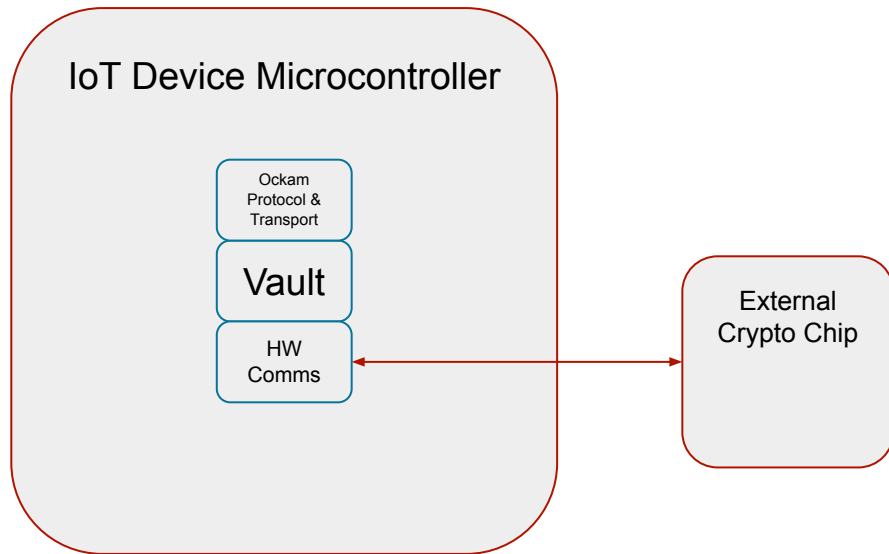
- Uses vendor library or Ockam-developed library to make external calls via a hardware communication protocol (I2C, SPI, UART, USB, etc.)
- Keys stored in external hardware device
- Specifically designed hardware for crypto operations, no overhead on microcontroller other than making API calls to device.
- A hardware vault may any of the following:
  - Trusted Execution Environment
  - Trusted Platform Module
  - Secure Enclaves
  - Hardware Security Modules
  - Crypto Modules



# Types of Vaults

- **Hardware Vault**

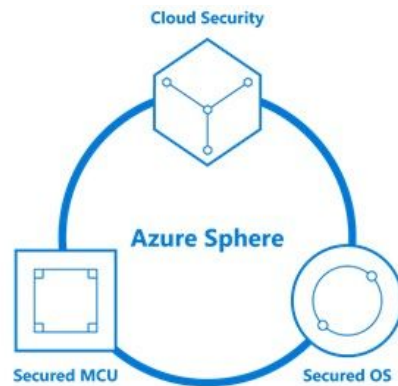
- Uses vendor library or Ockam-developed library to make external calls via a hardware communication protocol (I2C, SPI, UART, USB, etc.)
- Keys stored in external hardware device
- Specifically designed hardware for crypto operations, no overhead on microcontroller other than making API calls to device.
- A hardware vault may any of the following:
  - Trusted Execution Environment
  - Trusted Platform Module
  - Secure Enclaves
  - Hardware Security Modules
  - Crypto Modules



# Types of Vaults

Other platforms we're keeping an eye on:

- RISC-V
  - Crypto Extension
  - Key Storage
- Azure Sphere
  - Pluton Security Subsystem
- OpenTitan
  - Open-source guidelines for silicon root-of-trust



Q & A