Transaction Encryption

Transaction encryption unlike contract state encryption has two parties who need data access. The scheme therefore makes use of the DH-key exchange as described in the previous section to generate a shared encryption key. This symmetrictx_encryption_key is unique for every transaction and can be used by both the network and the user to verify the completed transactions.

1. Generation of shared secret - user side

Using the Eliptic-Curve Diffie Hellman key exchange (ECDH) the user generates a shared secret fromconsensus_io_exchange_pubkey andtx_sender_wallet_privkey .

```

Copy tx_encryption_ikm=ecdh({ privkey:tx_sender_wallet_privkey, pubkey:consensus_io_exchange_pubkey,// from genesis.json });// 256 bits

```

1. Generatetx_encryption_key
2. user side

The user then generates a sharedtx_encryption_key using HKDF-SHA256 and thetx_encryption_ikm generated in step 1. The pseudo-random HDKF is used to ensure deterministic consensus across all nodes.

The random component comes from a 256-bit nonce so that each transaction has its own encryption key, An AES-256-GCM encryption key is never used twice.

```

Copy nonce = true_random({ bytes: 32 });

tx_encryption_key = hkdf({ salt: hkdf_salt, ikm: concat(tx_encryption_ikm, nonce), }); // 256 bits

```

1. Encrypt transaction - user side

After initiating a transaction the user encrypts the input data with the shared transaction encryption key, using an AES-256-GCM authenticated encryption scheme.

The input (msg) to the contract is always prepended with the sha256 hash of the contract's code. This is meant to prevent replaying an encrypted input of a legitimate contract to a malicious contract, and asking the malicious contract to decrypt the input.

In this attack example the output will still be encrypted with atx_encryption_key that only the original sender knows, but the malicious contract can be written to save the decrypted input to its state, and then via a getter with no access control retrieve the encrypted input.

```

Copy ad = concat(nonce, tx_sender_wallet_pubkey);

codeHash = toHexString(sha256(contract_code));

encrypted_msg = aes_128_siv_encrypt({ key: tx_encryption_key, data: concat(codeHash, msg), ad: ad, });

tx_input = concat(ad, encrypted_msg);

```

1. Generationtx_ecryption_key
2. network side

The enclave uses ECDH to derive the sametx_encryption_ikm from thetx_sender_wallet_pubkey and theconsensus_io_exchange_privkey . The network then derives thetx_encryption_key from the publicly signednonce and this shared secret using HDKF.

Within the trusted component the transaction input is decrypted to plaintext.

```
```

Copy nonce=tx_input.slice(0,32);// 32 bytes tx_sender_wallet_pubkey=tx_input.slice(32,32);// 32 bytes, compressed curve25519 public key encrypted_msg=tx_input.slice(64);

tx_encryption_ikm=ecdh({ privkey:consensus_io_exchange_privkey, pubkey:tx_sender_wallet_pubkey, });// 256 bits

tx_encryption_key=hkdf({ salt:hkdf_salt, ikm:concat(tx_encryption_ikm,nonce), });// 256 bits

codeHashAndMsg=aes_128_siv_decrypt({ key:tx_encryption_key, data:encrypted_msg, });

codeHash=codeHashAndMsg.slice(0,64); assert(codeHash==toHexString(sha256(contract_code)));

msg=codeHashAndMsg.slice(64);

```

BREAK - Data output formatting

The output must be a valid JSON object, as it is passed to multiple mechanisms for final processing:

- Logs are treated as Tendermint events
- Messages can be callbacks to another contract call or contract init
- Messages can also instruct sending funds from the contract's wallet
- A data section which is free-form bytes to be interpreted by the client (or dApp)
- An error section
- 

Here is an example output for an execution:

```

Copy { "ok": { "messages":[{ "type":"Send", "to":"...", "amount":"..." }, { "wasm":{ "execute":{ "msg":"{\"banana\":1,\"papaya\":2}",// need to encrypt this value "contract_addr":"aaa", "callback_code_hash":"bbb", "send":{"amount":100,"denom":"uscrt"} } } }, { "wasm":{ "instantiate":{ "msg":"{\"water\":1,\"fire\":2}",// need to encrypt this value "code_id":"123", "callback_code_hash":"ccc", "send":{"amount":0,"denom":"uscrt"} } } }], "log":[{ "key":"action",// need to encrypt this value "value":"transfer"// need to encrypt this value }, { "key":"sender",// need to encrypt this value "value":"secret1v9tna8rkemndl7cd4ahru9t7ewa7kdq87c02m2"// need to encrypt this value }, { "key":"recipient",// need to encrypt this value "value":"secret1f395p0gg67mmfd5zcqvpnp9cxnu0hg6rjep44t"// need to encrypt this value }], "data":"bla bla"// need to encrypt this value } }

```

Please Note!

- on aContract
- message, themsg
- value should be the samemsg
- as in ourtx_input
- , so we need to prepend thenonce
- andtx_sender_wallet_pubkey
- just like we did on the tx sender above
- On aContract
- message, we also send acallback_signature
- , so we can verify the parameters sent to the enclave (read more here: ......)
- 

```

Copy callback_signature = sha256(consensus_callback_secret | calling_contract_addr | encrypted_msg | funds_to_send)

```

- For the rest of the encrypted outputs we only need to send the ciphertext, as the tx sender can getconsensus_io_exchange_pubkey
- fromgenesis.json
- andnonce
- from thetx_input
- that is attached to thetx_output
- with this info only they can decrypt the transaction details.
- Here is an example output with an error:
- 
```

```

Copy { "err":"{\"watermelon\":6,\"coffee\":5}"// need to encrypt this value }

```

- An example output for a query:
-

```

Copy { "ok":"{\"answer\":42}"// need to encrypt this value }

```

1. Writing output - network side

The output of the computation is encrypted using thetx_encryption_key

```

Copy // already have from tx_input: // - tx_encryption_key // - nonce

if(typeofoutput["err"]=="string") {

encrypted_err=aes_128_siv_encrypt({ key:tx_encryption_key, data:output["err"], });

output["err"]=base64_encode(encrypted_err);// needs to be a JSON string }

elseif(typeofoutput["ok"]=="string") {

// query // output["ok"] is handled the same way as output["err"]...

encrypted_query_result=aes_128_siv_encrypt({ key:tx_encryption_key, data:output["ok"], });

output["ok"]=base64_encode(encrypted_query_result);// needs to be a JSON string }

elseif(typeofoutput["ok"]=="object") {

// init or execute // external query is the same, but happens mid-run and not as an output

for(minoutput["ok"]["messages"]) { if(m["type"]=="Instantiate"||m["type"]=="Execute") {

encrypted_msg=aes_128_siv_encrypt({ key:tx_encryption_key, data:concat(m["callback_code_hash"],m["msg"]), });

// base64_encode because needs to be a string // also turns into a tx_input so we also need to prepend nonce and
tx_sender_wallet_pubkey

m["msg"]=base64_encode( concat(nonce,tx_sender_wallet_pubkey,encrypted_msg) ); } }

for(linoutput["ok"]["log"]) { // l["key"] is handled the same way as output["err"]...

encrypted_log_key_name=aes_128_siv_encrypt({ key:tx_encryption_key, data:l["key"], });

l["key"]=base64_encode(encrypted_log_key_name);// needs to be a JSON string

// l["value"] is handled the same way as output["err"]...

encrypted_log_value=aes_128_siv_encrypt({ key:tx_encryption_key, data:l["value"], });

l["value"]=base64_encode(encrypted_log_value);// needs to be a JSON string }

// output["ok"]["data"] is handled the same way as output["err"]...

encrypted_output_data=aes_128_siv_encrypt({ key:tx_encryption_key, data:output["ok"]["data"], });

output["ok"]["data"]=base64_encode(encrypted_output_data);// needs to be a JSON string }

returnoutput;

```

1. Receiving output - user side

The transaction output is written to the chain and only the wallet with the righttx_sender_wallet_privkey can derivetx_encryption_key . To everyone else but the tx signer the transaction data will be private.

Every encrypted value can be decrypted by the user following:

```

Copy // output["err"] // output["ok"]["data"] // output["ok"]["log"][i]["key"] // output["ok"]["log"][i]["value"] // output["ok"] if input is a query

encrypted_bytes=base64_encode(encrypted_output);

aes_128_siv_decrypt({ key:tx_encryption_key, data:encrypted_bytes, });

```

- Foroutput["ok"]["messages"][i]["type"] == "Contract"
- ,output["ok"]["messages"][i]["msg"]
- will be decrypted in by the consensus layer when it handles the contract callback
-

Last updated1 year ago On this page *1. Generation of shared secret - user side* 2. Generate tx_encryption_key - user side * 3. Encrypt transaction - user side* 4. Generation tx_ecryption_key - network side* BREAK - Data output formatting* Please Note! * 5. Writing output - network side* 6. Receiving output - user side

Was this helpful? Edit on GitHub Export as PDF