Summary

TZIP-017 proposes a standard for a pre-sign or "*Permit*" interface: a lightweight on-chain emulation of `tzn` accounts.

This document provides an overview and rationale for the interface as well as an implementation in Lorentz

Abstract

On-boarding users to token contracts usualy requires a normal Tezos `tz1`, `tz2`, etc. account to submit transactions and pay fees. This requires:

- A pre-existing Tezos account to inject the `reveal` operation, which costs `0.257 Tez`
- A method to pay the holder of the pre-existing account for the reveal

The Permit interface emulates `tzn` accounts on the contract level by implementing their core security features:

- Injected operations are signed using a public key that has the desired key-hash, i.e. `tz1`, `tz2`, etc.
- The submitted parameter (hash) is signed
- To prevent replay attacks, the following are signed:
  - Across chains: the chain ID
  - Across contracts: the target contract's address
  - On the same contract: a strictly-increasing counter
- Signed operations expire, preventing "zombie" or "orphan" approvals
- A relayer network, a la the Ethereum Gas Station, could be used to coordinate fees for permitted transactions

This method supports all contract parameters that can be `PACK`ed, viz. all Michelson types that omit `big_map`.

To use the interface:

- A user crafts a parameter, hashes it, and signs the parameter + context variables (counter, chain Id, etc.) to create a Permit
- Any user can submit the Permit to the target contract
- Any user who knows the hashed parameters can call the target contract on behalf of the signer, consuming the Permit.

Some optimizations implemented:

- Expiry has three levels of granularity so that you rarely need to set expiry for a Permit.
  - Whole-contract default expiry
  - User-specific default expiry
  - Single Permit expiry

Specification

`blake2b_hash`

```
blake2b_hash := bytes
```

`blake2b_hash` is an alias for `bytes` that result from running `PACK` and then `BLAKE2B` on the input.

`seconds`

```
seconds := nat
```

`seconds` is an alias for `nat` that results from `SUB` on two `timestamp`'s.

Abstract storage specification

This is how the contract must behave, but no particular Michelson types or layouts are specified.

There are three "tables":

- `permits`:
  - `key: pair address bytes`
  - `val: timestamp`
- `user_expiries`:
  - `key: address`
  - `val: option seconds`
- `permit_expiries`:
  - `key: pair address bytes`
  - `val: option seconds`

And two top-level fields: The storage also contains two variables:

- `default_expiry: seconds`
- `counter: nat`

Entrypoints

Submission

All contracts implementing TZIP-16 MUST implement BOTH

- An entrypoint to submit Permits:
  - In one-step: submit both the Permits and permitted parameters in one step using a specialized entrypoint type
  - In separate-steps: submit the Permit(s) and permitted parameter(s) in separate steps using an additional entrypoint: `permit`
- `setExpiry`: set your default expiry or the expiry for a particular Permit

One-step Permit Entrypoints

While The `permit` entrypoint allows submitting the Permit and parameters in separate steps without modifying the entrypoint's type, submitting both of them in one-step requires a new entrypoint type.

Given an entrypoint of the form:

```
(entrypoint_type %entrypointName)
```

The one-step Permit entrypoint type MUST BE of ONE of the following forms:

- Batched:

```
list (pair %name entrypoint_type
                 (option %permit (pair key
                                       signature)))
```

- Non-batched, i.e. the batched version without the outermost `list`:

```
pair %name entrypoint_type
           (option %permit (pair key
                                 signature))
```

Note that since the parameter is present, the Blake2B `bytes` hash of the parameter unnecessary and thus omitted from the `permit` field.

See below for naming restrictions when the original entrypoint is also present.

Naming One-step and Separate-step Permits

If both the original entrypoint (`entrypointName`) and the one-step Permit version are implemented in a contract, the entrypoint names must be of the following form:

- `entrypointName`: The original entrypoint's name
- `permitEntrypointName`: The name of the entrypoint with one-step Permits
    - `"permit"` or `"permit_"` MUST BE the prefix.
    - `entrypointName` MUST follow immediately after the prefix.
    - The first character of `entrypointName` MAY BE uppercased.

Additionally, calling `entrypointName` MUST BE equivalent to calling `permitEntrypointName` with `None` in its `permit` field.

Specification

Non-batched

If the non-batched form is implemented, its behavior must be equivalent to the batched form with a singleton list input:

```
entrypointNameUnbatched(x) == entrypointName([x])
```

Batched

We define one-step Permits in terms of the simpler separate-step `permit` entrypoint, which accepts a Permit and then allows anyone to submit the permitted parameters in a separate step.

Given a series of Permits and their respective parameters:

```
P_0, P_1, .., P_N := Permits for X_0, X_1, .., X_N
X_0, X_1, .., X_N := parameters for entrypointName
```

Calling `permit(P_i)` followed by `entrypointName([X_i, None)]` for `i = 0, 1, .., N` within one operation MUST BE equivalent to calling `entrypointName` with the Permits and parameters in one step:

```
permit(P_0)
permitEntrypointName([(X_0, None)])

permit(P_1)
permitEntrypointName([(X_1, None)])
..
permit(P_N)
permitEntrypointName([(X_N, None)])

 ==

permitEntrypointName([ (X_0, Some P_0) ;
                       (X_1, Some P_1) ;
                       ..
                       (X_N, Some P_N)
                     ])
```

In other words, the one-step Permit entrypoint must be equivalent to an implementation that iterates over the list, calling `permit` and then the original entrypoint for each pair. Additionally, the entire batch MUST fail if and only if any element of the list would result in failure.

Note that submitting a batch of Permits/parameters across multiple operations may not be equivalent to submitting the entire batch in one step:

- One of the entrypoint calls could fail, which would cancel the entire batch if it were submitted in a single operaton, but would fail independently when submitting the batch in separate steps
- One of the Permits could expire during the batch's submission
- The behavior of `entrypointName` could change across multiple operations, e.g. if it's implemented using the `NOW` or `LEVEL` instructions

Separate-step Permit

With separate-step Permits, the original entrypoint `entrypointName` retains its original type and the Permit must be submitted to the `permit` entrypoint in a separate step, i.e. in an additional contract call.

Specification

```
list (pair %permit key
               (pair signature
                    bytes))
```

- `key` is the signer's public key
- `bytes` is the `Blake2B` hash of the `pack`ed parameter
    - For example, if the parameter is `42`, we might calculate it using: `PUSH nat 42; PACK; BLAKE2B`
- `signature` is by the given `key` and signs the `bytes` with
    - The chain ID
    - The target contract address
    - The current counter

To make a Permit, a user:

- Chooses the Michelson parameters of the target contract that they want to submit
- Applies the Michelson instructions `PACK`, followed by `BLAKE2B` to get the hash
- Signs the hash as if it were the `lambda` in the Generic Multisig
    - It's the same method as is used for the Specialized Multisig: See the Tutorial or below for more details.
- Any user may then submit a list of one or more `Pair USER_PUBLIC_KEY (Pair PARAMETER_HASH_SIGNATURE PARAMETER_HASH)` to the contract
- Any user may then call the permitted parameters, once they're revealed by the signer

Storage updates

When a Permit is created:

- The creation time is saved. See the `SetExpiry` entrypoint for more detail.
- The `counter: nat` is incremented by one.

Duplicate Permit

If a duplicate Permit is submitted, i.e. the hash is in `permits` for the user and has not expired, the contract must `FAILWITH` either:

- The `string`: `"DUP_PERMIT"`
- The `pair string t`: `Pair "DUP_PERMIT" x`, for any `x` of type `t`

Missigned Permit

If a permit is missigned, the `permit` entrypoint must fail with (`FAILWITH`) a pair composed of the string `"MISSIGNED"` and the `bytes` whose signature was invalid:

```
Pair "MISSIGNED" missigned_bytes
```

For example,

```
Pair "missigned"

0x05070707070a000000049caecab90a0000001601dcf1431e9fa9c4fc3b0859e3ea91bbfecfbb7252
00070700000a000000200f0db0ce6f057a8835adb6a2c617fd8a136b8028fac90aab7b4766def688ea
0c
```

See here for a more detailed explanation of this example.

Including the `bytes` to sign in a predictible way in the error allows users to find them by submitting a missigned permit using a `dry-run`.

Packing the `Blake2B` hash for signing

When the contract validates a Permit's `signature`, it combines the given `Blake2B` with:

- The chain ID
- Its own contract address
- The current counter

To do so, it runs code equivalent to the following `lambda (pair nat bytes) bytes` on the `pair` of `(nat :counter)` and `(bytes :given_parameter_Blake2B)`:

```
PACK_FOR_SIGNING := {
  SELF;
  ADDRESS;
  CHAIN_ID;
  PAIR;
  PAIR;
  PACK
}
```

SetExpiry

```
pair address (pair seconds (option bytes))
```

- `address` is user's `key` hash
- `seconds` is the new user-default expiry

- `Some bytes` for a particular Permit
  - `bytes` is the `Blake2B` hash of the `pack`ed parameter
  - The Permit *must* exist and not be revoked
- `None` for a user-default, the effective expiry of any Permit whose specific expiry is unset
  - If the user-default is unset, the effective expiry of any Permit whose specific expiry is unset is the global default.

Users may only set their own (default and Permits') expiries.

If the difference between the stored timestamp and `NOW` is at least the effective expiry, the Permit is revoked. A revoked Permit can't be used with the `Permit` or `SetExpiry` entrypoints. See `Cleaning up Permits` for more detail.

Individual permits may be revoked by setting the expiry for that Permit to `0 seconds`

TZIP-016

This contract MUST implement TZIP-016.

The following off-chain-views are required and MUST be provided with Michelson implementations:

- `GetDefaultExpiry: unit → nat`
  - Access the contract's default expiry in seconds
- `GetCounter: unit → nat`
  - Access the current counter

Implementation

Cleaning up Permits

Cleaning up expired Permits is left to the implementor:

- Keeping all Permits in contract storage is possible, but would use a lot of storage
  - For example, it takes `67 bytes` for a user to submit their first Permit Since storage costs are only incurred for increases in storage, consider the difference in cost for `100` Permits in series, by a single user:
    - `6700 bytes * cost/byte` to store all permits simultaneously
    - `67 bytes * cost/byte` to store only one permit at a time
- Permits can be deleted lazily
  - For example, this [stablecoin contract]((https://github.com/tqtezos/stablecoin) finds and deletes all of a user's expired Permits whenever they submit new Permit
- Expired permits could be cleared out using an explicit entrypoint

Limiting `seconds` in `SetExpiry`

If users are allowed to call the `SetExpiry` with arbitrarily large `nat`'s, some or all of the contract could be locked (i.e. it costs more than the gas limit to call an entrypoint) if an expiry is set to a large enough value.

Implementors should consider putting an upper bound on this value as a safety measure.

Implementations

- A [stablecoin contract]((https://github.com/tqtezos/stablecoin) implementing Permit
- A partial implementation of Permit in Lorentz may be found here

Copyright