Luksathon Developer Manual

November, 2021

—

António Pedro
Dropps.io
Lukso Blockchain

1. Key SmartContract Design Patterns

I. Universal Profiles

Universal Profiles (UPs) are smart contract accounts that should be used instead of EOA (Externally Owned Addresses). This carries several advantages.

In order to understand all a Universal Profile encompasses, it is easier to see it as a set of components working together.

Component 0: ERC173

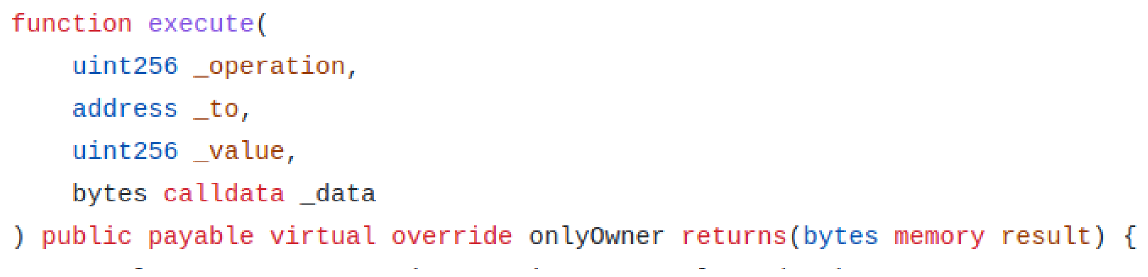
A Universal Profile is an ownable contract. This means that access to certain features is restricted to the owner address at that time. For the first time, the security of a blockchain account is upgradeable. Users no longer depend on saving their private key since the moment their account was generated or forever losing access to their assets in case the key gets lost.

The owner of their account can even be another smart contract. And this smart contract can implement any security mechanism. From a multi-sig wallet where your family and close friends can step in, to any off-chain authentication method managed by a centralized or decentralized institution.

Component 1: ERC725X

The ERC725X is a component that gives a smart contract address the ability to forward calls to any other contract. Just like a proxy would. This is an essential part of the Universal Profile system because it is what allows the UP to interact with any other contract on the blockchain. All a developer has to do is pass the UP the calldata for the function call. For example, instead of directly calling the Dropps Token Contract to transfer tokens, a developer passes the calldata to the UP which will run ***execute()*** to forward the call to the Dropps Token Contract and do the token transfer.

The key function is the *execute()* function:



```
function execute(
    uint256 _operation,
    address _to,
    uint256 _value,
    bytes calldata _data
) public payable virtual override onlyOwner returns(bytes memory result) {
```

This function can do a:

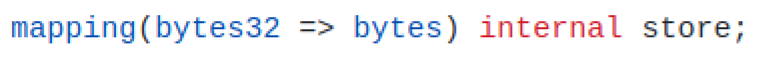
- Read-only call with operator STATICCALL.
- Read/write call with operator CALL.
- Storage delegating call with operator DELEGATECALL.
- Contract creation with operator CREATE and CREATE2

All you have to do is specify the correspondent number on the _*operation* parameter.

It is interesting that we can have a smart contract that does the calls for us, instead of them coming directly from our EOA. But why? Because this component can be owned. And only the owner can run execute(). If the owner is our regular EOA, then we've effectively put our address behind a proxy contract that can now execute all our calls. Interesting...if only I could store more information in this proxy contract, my Identity would be much more than an address...I could have metadata there like my username, the addresses of my digital assets...enter the second component.

Component 2: ERC725Y

Is a simple key/value data store. It features functions for getting and setting data and the storage is a simple mapping:



```
mapping(bytes32 => bytes) internal store;
```

The key is expected to be the keccak256 hash of a type name. e.g. *keccak256('ERCXXXMyNewKeyType')*.

This component is also ownable, just like the ERC725X. So now a user, organization or smartcontract has a way of storing identity information (725Y) and a way of signing operations with that identity (725X).

Component 3: LSP-1-UniversalReceiver

First (before Solidity 0.6.0), there was only *fallback()*. This function executed:

1. When a contract was called without any data, for example via *.send()* or *.transfer()* functions. Popular use cases for it were to reject the transfer of ether, emit events or forward the ether to another address.

2. When no function in the contract matched the function identifier in the call data. The main use case being proxies, which placed a DELEGATECALL inside the *fallback()*; the magic that allowed the proxy contract to call any contract without knowing its interface.

Solidity core devs decided to split the *fallback()* function in 0.6.0. Making two separate functions for the two separate use cases. We got:

- *receive() external payable* — for empty calldata (and any value)
- *fallback() external payable* — when no other function matches (not even the receive function). Optionally payable.

The receive function became the one used to handle value transfers to the contract. Internally, value transfers are calls without calldata and with an arbitrary value in the value field. This means that **only ether transfers can be handled by receive().**

Hence, the need for a way to handle the transfer of other types of blockchain assets remained. The UniversalReceiver (LSP-1) is the Lukso version of standards that tackle this issue (ERC223, ERC777).

One might wonder why it is important to pass control to a receiving contract when some asset (like a token) is transferred to it.

The first use case is when tokens get transferred by mistake. Tokens should only be transferred to a receiver that is expecting or at least can handle them. By passing control to the receiver during the transfer, we give the receiver a chance to reject unexpected token transfers or at least forward them to a contract where they can be retrieved in case of a mistake.

The second use case is saving a record of owned assets on-chain, in the state of an account. Ownership lives with the respective contracts, but a pointer to those assets can be saved in the Universal Profile key/value store. This makes it so much easier to track the assets in the state of an account.

The third use case is reacting to received assets, for example, by emitting events or calling other smart contracts.

II. NFT 2.0