ULCDocumentsCore

Sep 02, 2019

Contents

ULCDocuments

The Blockchain-based signature app

Important: The documentation is not complete yet. It may change in near future.

The main goal of this documentation is to explain the mechanism behind ULCDocuments, on the Ethereum blockchain. This will allow anyone to develop their own WebApp or software to interact with ULCDocuments.

This documentation's source code is available here

Note: For the beta version, full source code of *smart contracts* are not avaiable.

CHAPTER 1

What is ULCDocuments ?

ULCDocuments is an Free and Open-Source software using the Ethereum blockchain. The goal to make it easy for anyone to sign any type of document permanantly on the blockchain.

ULC is the acronym for **U**ltra **L**ow **C**ost. We aim to provide a very cheap way to certify documents: you only need to pay transactions fees determined by the blockchain when sending signatures.

ULCDocuments is composed of 2 Ethereum smart contracts. They should be deployed by different actors to respect the decentralized aspect of the blockchain.

CHAPTER 2

Why use ULCDocuments ?

Using ULCDocuments instead of simply sending a transaction with your hash as data gives developers the possibility to **create tools or other smart contracts which can then interact easily with your signatures**.

We aim to create an open source project, used by many other projects, in order to **create a powerfull tool allowing interaction between people's signatures**. For example, anyone could create **extensions for PDF**, **Office/LibreOffice plugin**, **native applications** working with ULCDocuments!

Feel free to create your own project using ULCDocuments verification in order to trust signature of a third party. You can also use and/or create many JSON standards like Open Badges with ULCDocuments!

To showcase this technology, and to give an example of an application using ULCDocuments, we have created a WebApp implementing ULCDocumentsCore at https://ulcdocuments.blockchain-elite.fr/

CHAPTER 3

Who is behind the project ?

ULCDocuments is a project developped by Blockchain Elite Labs . The Labs section from BLockchain Elite aims to create some experimental Open-Source tools based on the Blockchain technology, and to promote Blockchain Elite know-how.

3.1 ULCDocKernel

3.1.1 Summary

Each organisation or individual has to publish its own **ULCDocKernel** smart contract. The kernel is used to explicitly publish signatures to the blockchain. This way, the owner of the published ULCDocKernel smart contract has a complete and independent way to certify documents.

The kernel has 3 main features :

- Multi-Owner ability
- Native timestamp tracking
- *Undestroyable* signatures

3.1.2 Multi-Owner and Multi-Operator ability

For most organization, one person cannot do dangerous actions alone, such as signing official documents. As such, the kernel has the ability to wait for multiple **confirmations** before doing anything, like publishing a signature, or changing critical configuration parameters.

Roles

ULCDocKernel has 2 levels of administrative rights:

- `operators` can only push, confirm and revoke signatures.

- `owners` are administrators of the Smart Contract. In addition to having operator's rights, they can also change sensible parameters.

In order to do something on the Smart Contract, like **signing** or **changing parameters**, you need to call a **requester**. This requester will record you request and only process it if it reaches `operatorsForChange` (respectively `ownersForChange`) request count.

Warning: An account can't be `owner` and `operator` at the same time.

By default, ULCDocKernel is configurated to work with only one owner account (and no operators). If you use the kernel with only one owner, all your requests will be done immediately (as `ownersForChange` will be set to 1);.

Requester

When you want to do an action, requesters will create a `keccak256` hash of the request and store it inside a mapping.

```
mapping(bytes32 => address[]) public doOwnerRequest;
mapping(bytes32 => address[]) public doOperatorRequest;
```

Then, each time an address requests the same action, it will be added into an array. The action will be processed only when the array's length reaches the value `operatorsForChange` or `ownersForChange`.

Constructor

By default, the creator of the smart contract is the owner, and everything is configurated to work with only one confirmation, as explained above.

```
constructor() public {
    owners[msg.sender] = true;
    ownerCount = 1;
    ownersForChange = 1;
    operatorsForChange = 1;
}
```

Variables

Bellow are the variables used by the smart contract for operator management:

```
uint256 public ownerCount;
uint256 public ownersForChange;

uint256 public operatorCount;
uint256 public operatorsForChange;

mapping (address => bool) public owners;
mapping(bytes32 => address[]) public doOwnerRequest;

mapping (address => bool) public operators;
mapping(bytes32 => address[]) public doOperatorRequest;

event OwnershipNewInteractor(address indexed newOwner);

```

(continues on next page)

(continued from previous page)

```
event OwnershipLossInteractor(address indexed oldOwner);

event OwnershipTransferred(
    address indexed previousOwner,
    address indexed newOwner
);

event OperatorNewInteractor(address indexed newOperator);

event OperatorLossInteractor(address indexed oldOperator);

event OperatorshipTransferred(
    address indexed previousOperator,
    address indexed newOperator
);
```

Functions

The ULCDocKernel smart contract lets you use different functions to interact with it, detailled bellow:

Setters (Requesters)

These functions allow you to manage owners and operators for this kernel. Keep in mind these behave like requesters, and as such need enough confirmations to work.

```
// Set the number of operators needed for confirmation
function setOperatorsForChange(uint256 _nb) external onlyOwner {}

// Set the number of owners needed for confirmation
function setOwnersForChange(uint256 _nb) external onlyOwner {}

// Add a new kernel owner
function requestAddOwner(address _newOwner) external onlyOwner{}

// Add a new kernel operator
function requestAddOperator(address _newOperator) external onlyOwner {}

// Transfer kernel's ownership
function requestChangeOwner(address _oldOwner, address _newOwner) external onlyOwner{}

// Remove a kernel owner
function requestRemoveOwner(address _removableOwner) external onlyOwner{}

// Remove a kernel operator
function requestRemoveOperator(address _removableOperator) external onlyOwner{}
```

Getters

These functions allow you to get information on the kernel. As these are not requesters, you do not need to have enough confirmations to use them.

```
//Returns all adresses who approved the keccak256 operator request
function getOperatorRequest(bytes32 _theKey) external view returns(address[] memory)
↪{}

//Returns numbers of operators who confirmed the keccak256 request.
function getOperatorRequestLength(bytes32 _theKey) external view returns(uint256) {}

//Returns all adresses who approved the keccak256 owner request
function getOwnerRequest(bytes32 _theKey) external view returns(address[] memory) {}

//Returns the number of owners who confirmed the keccak256 request.
function getOwnerRequestLength(bytes32 _theKey) external view returns(uint256) {}
```

3.1.3 Signature Storing

Structure

ULCDocKernel has a `struct` called `Document` which has all information about a signed document.

```
struct Document {
      bool initialized;
      bool signed;
      bool revoked;

      uint256 signed_date;
      uint256 revoked_date;
      uint16 document_family;
      uint8 signature_version;

      string revoked_reason;
      string source;
      string extra_data;
}
```

By default, the EVM makes all var set to `false`, `0`, or `""`.

- `initialized` is set to `true` as soon as someone starts to try signing a document. When it is activated, you cannot push an another version of the document. This is a security measure to prevent **deleting** or **cheating** about the fact that you sign some extra data, document family and so on.
- `signed` is set to `true` as soon as the document has enough confirmations. **It is the only field you need to trust to know if something has been signed or not.**
- `revoked` is set to `true` as soon as the document has enough revoke requests from operators.
- `signed_date` is the UNIX timestamp (result of `block.timestamp` in solidity) of the block where signature has been officially sent.
- `revoked_date` is the UNIX timestamp (result of `block.timestamp` in solidity) of the block where revoked state has been officially declared.
- `document_family` is the index of the array where is stored the string value describing the family.
- `signature_version` is the version of the Kernel which hosts signatures (for migration purposes)
- `revoked_reason` is defined by operators when they push a revoke statement.
- `source` is defined by operators when they push document. It is the location where we can find the document if it is public.

- extra_data is defined by operators when they push document. It is a key-value field (param:value, param2:value2). It can be used to add extra information for automatic process, or to be compatible with newer Kernel Version.

Finding a Signature

To find a signature, you need its bytes32 code. To obtain it, just check the HASH_ALGORITHM string. By default, ULCDocKernel uses the document's **SHA3-256** hash.

```
mapping(bytes32 => Document) public SIGNATURES_BOOK;
```

Note: Because mapping hashes its keys with a *32 bytes* format, it is useless to use hash algorithms with more than *32 bytes*, like for example ‘‘SHA3-512‘‘*.

Constructor

```
constructor() public {
    HASH_ALGORITHM = "SHA3-256";
}
```

Variables

Bellow are the variables used by the smart contract for signature management:

```
uint256 public DOCUMENTS_COUNTER; // stat purposes only
string public HASH_ALGORITHM; //Essential Information to know how to hash files

string[] public DOC_FAMILY_LIST = ["Undefined",
"Diploma",
"Bill",
"Order",
"Letter",
"Publication",
"Code",
"Image",
"Audio",
"Video",
"Binary",
"Text"];

// Stringified version to get all array in one .call()
string public DOC_FAMILY_STRINGIFIED = "Undefined,Diploma,Bill,Order,Letter,
↪Publication,Code,Image,Audio,Video,Binary,Text";

mapping(bytes32 => Document) public SIGNATURES_BOOK;
```

Functions

Push Requests

Pushing a document is the first step to signing it with data. Then, other operators will only need to confirm it to make the signature effective. This is the same when revoking a document. If you do not want to add data to your signature, use confirm requests instead, as these are cheaper.

```
//Request to sign a document, and add data to the signature.
function pushDocument(bytes32 _SignatureHash, string memory _source, uint16 _
↪indexDocumentFamily, string memory _extra_data) public atLeastOperator␣
↪whenNotPaused notUpgraded{}

//Request to add a "revoked" statement on the signature, and add a reason for that␣
↪(can be then displayed to clients).
function pushRevokeDocument(bytes32 _SignatureHash, string calldata _reason) external␣
↪atLeastOperator whenNotPaused {}
```

Note: When you use a push request on your Kernel (to sign it or revoke it), you automatically confirm it. So, if you use a simple signature Kernel (only one owner/operator needed to confirm the request), your request will accepted in only one transaction.

Confirm Requests

When dealing with kernels using multiple operators, only the first one needs to use a push request. Every other operator will only need to confirm that request by using one of the functions below (to sign or revoke a document). These functions can also be used by the first operator instead of the push functions. It won't be possible to use the field `extra_data`, but it will result in cheaper transaction.

```
//Request to confirm a signature.
function confirmDocument(bytes32 _SignatureHash) public atLeastOperator whenNotPaused␣
↪notUpgraded{}

//Request to confirm a revoke statement.
function confirmRevokeDocument(bytes32 _SignatureHash) external atLeastOperator␣
↪whenNotPaused {}
```

Multiple Documents Requests

It is also possible to request multiple signatures with only one transaction. The only drawback is that you won't be able to push documents with data, because we can't use string arrays with standard ABI (but you will be able to set the document family as it is stored as an integer).

```
// Confirm mulitple documents
function confirmDocumentList(bytes32[] calldata _allKeys) external atLeastOperator␣
↪whenNotPaused notUpgraded {}

// Push mulitple documents that have only a documentFamily set.
function lightPushDocumentList(bytes32[] calldata _allKeys, uint16[] calldata _
↪allDocumentFamily) external atLeastOperator whenNotPaused notUpgraded {}
```

Utility

This function can be useful when building an application, but is not needed when signing documents.

```
//Return the size of the DOC_FAMILY_LIST array
function getDocFamilySize() public view returns(uint256) {}
```

3.1.4 Security about signatures

Requirements

ULCDocument is designed to resist **minority key stealing** attacks, as long as you meet those requirements:

- Attacker must have **less than** `ownersForChange` **and** `operatorsForChange` keys stolen
- You must have at least `ownersForChange` keys available

Warning: If you are an organisation, do not give all your *owner accounts* to the same person!

Recovery

Clean Compromised Accounts

First remove all compromised `owners` and `operators` using associated functions (see MultiRoles section).

Then, you need to have at least `operatorsForChange` available accounts.

Clean Requests

If an attacker started to sign documents (but can't confirm it, as he does't have enough accounts), then you need to clear it by using the `clearDocument` function below:

```
function clearDocument(bytes32 _SignatureHash) external atLeastOperator notUpgraded␣
↪whenNotPaused {}
```

Note: You can't clear a document if it has already been signed or revoked.

If an attacker started to revoke documents you can also clear the request (and delete possible `revoked_reason` added with it) using the function below:

```
function clearRevokeProcess(bytes32 _SignatureHash) external atLeastOperator␣
↪notUpgraded whenNotPaused {}
```

Note: You can't clear a document revoke request if it has already revoked.

If an attacker started to request a selfdestruct, you can clear the requester counter using the function below:

```
//note : only a compromised owner account can request kill().
function clearKill() external onlyOwner {}
```

Contest Other Operations

Owners can also use a *lower level* of cleaning by reseting every request that they want.

```
/**
* @dev Delete the request queue for that action. Used to save gaz if there is a lot␣
↪of owners or
for security purposes, to reset the action agreed counter.
*/
function requestOwnerContest(bytes32 _doRequestContest) external onlyOwner {}

function requestOperatorContest(bytes32 _doRequestContest) external atLeastOperator {}
```

Note: To reset a counter, you first need to know which request has been created by the attacker. Then you need to calculate the `bytes32` hash of the request and call *contester functions*

3.2 ULCDocMod

3.2.1 Summary

The aim of the *ULCDocMod* smart contract is to be an authority that certifies *ULCDocKernel* address. It has to guarantee the integrity of organisation's *ULCDocKernel* code and the identity of the owner. We can slightly compare it to a *Certificate Authority* for https protocol.

As everyone can publish a *ULCDocMod* smart contract, the acknowledgement is only based on the trust the user give. For example, Blockchain-Elite, the developper of ULCDocuments, publishes it's own ULCDocMod smart contract.

3.2.2 More in depth