Audit of True-NFT Smart Contracts

By OCamlPro

September 22, 2021

Table of Minor, Major and Critical Issues

Contents

Chapter 1

Introduction

This informal audit details the functionning of the TrueNFT smart contracts of the FreeTon repository. The `true-nft-core` directory contains a simplified version of a more complex NFT project; it only contains a high level version, synthesizing the storage of the different contracts, so as to retrieve them with DeBots (the DeBots code were not part of the audited contracts). This report contains a full review of the code with the different issues, from minor to critical, that were found and fixes are proposed in some cases.

We found 2 issues that we qualified as Critical, and 5 issues that we qualified as Major. Some of the issues were sent to the authors through private exchanges on Telegram.

This document is the submission to the 17th contest of the ForMet subgovernance, which is accessible here.

Chapter 2

Overview

The implementation of NFT studied in this document is a Proof of Concept of how NFT primitives should be built on FreeTON. It does not correspond to the actual implementation of an NFT that would be directly deployed. Instead, these contracts act as templates of how NFT contracts should be implemented to provide the same interface as other NFTs on FreeTON. As such, the different contracts of TrueNFT-core should be modified, or at minimum considered as just a source of inspiration. Considering this special aspect of these contracts, serving as a *specification written in Solidity*, we focused this audit on two aspects. First, we looked for issues in the NFT-Core logic, i.e. how the different contracts and data interactions were implemented with respect to what an NFT should be. Second, as a usual audit, we searched for actual issues in the code. While this second kind of issues may not be relevant (as the project is a POC that should be modified), we assumed that if this repository is expected to be a reference, it should be perfect in every regard.

2.1 Specification

Non Fungible Tokens are deployed from a **NftRoot** contract which will be the basis of all minted tokens. This contract has two purposes.

- The deployment of a **Data** contract, representing a fraction of the digitalized asset (i.e. the NFT itself). So as to easily retrieve the information of the asset from outside the blockchain, the **Data** contract deploys two **Index** contracts, each pointing to the **Data** address: one retrievable from the NftRoot address and the owner's address (allowing users to list all NFTs derived from this root and owned by the same owner), and one from only the owner's address (allowing users to list all NFTs owner by the same owner).

- The deployment of an **IndexBasis** contract with the Data code hash.

These four contracts represent the whole NFT core implementation. The DeBot SDK provides a primitive to list all contracts with a given code-hash. To use this primitive to list NFTs, the contracts make use of *salted* codes: for example, **Index** contracts are deployed with the same code, but salted with the owner's address, and either the **NftRoot** address or zero, so as to create different code-hashes (with actually the same code!). As a consequence, the DeBot primitive can be used to list all NFTs for a given owner address, within or not a specific NFT root.

From our audit, we think that this mechanism is safe, and works as expected.

2.2 Generic issues

Before reading in detail the source code, several issues (mostly coding habits) affected the project as a whole. We list them in this section.

Major issue: Funds accessibility and bounced messages

Unless a contract is destroyed (which is not the case for all contracts), funds are not accessible. As there is no error handling, especially for bounced messages, funds may accumulate on the contracts. However, there are no provided functions to recover such funds to the user.

Minor issue: Naming convention

Static variables should start with a prefix like ”s_” and globals should start with a prefix like ”g_” or ”m_” and internal/private functions should start with ”_”. Following such rules would make these contracts much easier to read and audit.

Chapter 3

Contract Data

Contents

3.1 Overview

Major issue: No way to get funds back

Tokens sent to the contract are locked forever. Such tokens are received when IndexBasis are destroyed, or when these contracts are deployed. Such tokens could be used to fund the long-term storage of the contract, but, if it is the purpose, it should be specified.

3.2 Contract Inheritance

IData	
IndexResolver	

3.3 Static Variable Definitions

uint256	_id	

```
    uint256 static _id;
```

3.4 Variable Definitions

address	_addrRoot	
		used in @7.Data.transferOwnership
		used in @7.Data.getInfo
		used in @7.Data.deployIndex
		used in @7.Data.deployIndex
		used in @7.Data.deployIndex
		assigned in @7.Data.:constructor
		used in @7.Data.:constructor
address	_addrOwner	
		assigned in @7.Data.transferOwnership
		used in @7.Data.transferOwnership
		used in @7.Data.transferOwnership
		used in @7.Data.transferOwnership
		used in @7.Data.transferOwnership
		used in @7.Data.getOwner
		used in @7.Data.getInfo
		assigned in @7.Data.:constructor
		used in @7.Data.:constructor
address	_addrAuthor	
		assigned in @7.Data.:constructor
		used in @7.Data.:constructor

```
    address _addrRoot;
```

```
    address _addrOwner;
```

```
    address _addrAuthor;
```

3.5 Constructor Definitions

3.5.1 Constructor

Minor issue: Constants

Value ”101” should be defined as a constant (would improve readability)

Major issue: addrOwner may be null

The constructor allows `addrOwner` to be null, making the contract useless and untransferable. A `require` should check non-null `addrOwner` before `tvm.accept`.

```
    constructor(address addrOwner, TvmCell codeIndex) public {
        optional(TvmCell) optSalt = tvm.codeSalt(tvm.code());
        require(optSalt.hasValue(), 101);
        (address addrRoot) = optSalt.get().toSlice().decode(address);
        require(msg.sender == addrRoot);
        require(msg.value >= Constants.MIN_FOR_DEPLOY);
        tvm.accept();
        _addrRoot = addrRoot;
        _addrOwner = addrOwner;
        _addrAuthor = addrOwner;
        _codeIndex = codeIndex;

        deployIndex(addrOwner);
    }
```

3.6 Public Method Definitions

3.6.1 Function getInfo

```
    function getInfo() public view override returns (
        address addrRoot,
        address addrOwner,
        address addrData
    ) {
        addrRoot = _addrRoot;
        addrOwner = _addrOwner;
        addrData = address(this);
    }
```

3.6.2 Function getOwner

```
    function getOwner() public view override returns(address addrOwner) {
        addrOwner = _addrOwner;
    }
```

3.6.3 Function transferOwnership

Major issue: New owner may be null

The contract does not check that the new owner is not null. As a consequence, a user may lose complete ownership of the contract by mistake, with no new owner for the contract. A `require` should check that `addrTo.value` is not zero.

Minor issue: Sending `destruct` should specify sent value

It is a good practice to specify the value sent within messages, especially here where the `destruct` function may have a higher cost than expected in derived implementations. Fix: add a value field associated to a constant that can be easily modified by derived implementations.

Minor issue: New owner may be equal to the old one

The new owner of the contract may be equal to the old one, hence destructing and rebuilding identical contracts. A `require` should check that `addrTo` is not equal to `_addrOwner`.

```
    function transferOwnership(address addrTo) public override {
        require(msg.sender == _addrOwner);
        require(msg.value >= Constants.MIN_FOR_DEPLOY);

        address oldIndexOwner = resolveIndex(_addrRoot, address(
            this), _addrOwner);
        IIndex(oldIndexOwner).destruct();
        address oldIndexOwnerRoot = resolveIndex(address(0),
            address(this), _addrOwner);
        IIndex(oldIndexOwnerRoot).destruct();

        _addrOwner = addrTo;

        deployIndex(addrTo);
    }
```

3.7 Internal Method Definitions

3.7.1 Function deployIndex

Minor issue: Constants

Value ”0.4 ton” should be defined as a constant (would improve readability).

```
    function deployIndex(address owner) private {
        TvmCell codeIndexOwner = _buildIndexCode(_addrRoot, owner);
        TvmCell stateIndexOwner = _buildIndexState(codeIndexOwner,
            address(this));
        new Index{stateInit: stateIndexOwner, value: 0.4 ton}(
            _addrRoot);

        TvmCell codeIndexOwnerRoot = _buildIndexCode(address(0),
            owner);
        TvmCell stateIndexOwnerRoot = _buildIndexState(
            codeIndexOwnerRoot, address(this));
        new Index{stateInit: stateIndexOwnerRoot, value: 0.4 ton}(
            _addrRoot);
    }
```

Chapter 4

Contract DataResolver

Contents

4.1 Overview

In file `DataResolver.sol`

4.2 Variable Definitions

TvmCell	_codeData	
		assigned in @1.NftRoot.:constructor
		used in @1.NftRoot.:constructor
		used in @5.DataResolver._buildDataCode

```
    TvmCell _codeData;
```

4.3 Public Method Definitions

4.3.1 Function resolveCodeHashData

```
    function resolveCodeHashData() public view returns (uint256 codeHashData) {
        return tvm.hash(_buildDataCode(address(this)));
    }
```

4.3.2 Function resolveData

```
    function resolveData(
        address addrRoot,
        uint256 id
    ) public view returns (address addrData) {
        TvmCell code = _buildDataCode(addrRoot);
        TvmCell state = _buildDataState(code, id);
        uint256 hashState = tvm.hash(state);
        addrData = address.makeAddrStd(0, hashState);
    }
```

4.4 Internal Method Definitions

4.4.1 Function _buildDataCode

```
    function _buildDataCode(address addrRoot) internal virtual view returns (TvmCell) {
        TvmBuilder salt;
        salt.store(addrRoot);
        return tvm.setCodeSalt(_codeData, salt.toCell());
    }
```

4.4.2 Function _buildDataState

```
    function _buildDataState(
        TvmCell code,
        uint256 id
    ) internal virtual pure returns (TvmCell) {
        return tvm.buildStateInit({
            contr: Data,
            varInit: {_id: id},
            code: code
        });
    }
```

Chapter 5

Contract Index

Contents

5.1 Overview

In file `Index.sol`

5.2 Contract Inheritance

IIndex	

5.3 Static Variable Definitions

address	_addrData	
		used in @8.Index.getInfo
		used in @8.Index.destruct
		used in @8.Index.destruct
		used in @8.Index.:constructor

```
    address static _addrData;
```

5.4 Variable Definitions

address	_addrRoot	
		used in @8.Index.getInfo
		assigned in @8.Index.:constructor
		used in @8.Index.:constructor
		assigned in @8.Index.:constructor
		used in @8.Index.:constructor
address	_addrOwner	
		used in @8.Index.getInfo
		assigned in @8.Index.:constructor
		used in @8.Index.:constructor

```
    address _addrRoot;
```

```
    address _addrOwner;
```

5.5 Constructor Definitions

5.5.1 Constructor

Minor issue: Constants

Value "101" should be defined as a constant (would improve readability)

Minor issue: Double initialization of `_addrRoot`

`_addrRoot` is initialized twice if `addrRoot` = 0.

```
    constructor(address root) public {
        optional(TvmCell) optSalt = tvm.codeSalt(tvm.code());
        require(optSalt.hasValue(), 101);
        (address addrRoot, address addrOwner) = optSalt
            .get()
            .toSlice()
            .decode(address, address);
        require(msg.sender == _addrData);
        tvm.accept();
        _addrRoot = addrRoot;
        _addrOwner = addrOwner;
        if(addrRoot == address(0)) {
            _addrRoot = root;
        }
    }
```

5.6 Public Method Definitions

5.6.1 Function destruct

```
    function destruct() public override {
        require(msg.sender == _addrData);
        selfdestruct(_addrData);
    }
```

5.6.2 Function getInfo

```
    function getInfo() public view override returns (
        address addrRoot,
        address addrOwner,
        address addrData
    ) {
        addrRoot = _addrRoot;
        addrOwner = _addrOwner;
        addrData = _addrData;
    }
```

Chapter 6

Contract IndexBasis

Contents

6.1 Overview

In file `IndexBasis.sol`

6.2 Static Variable Definitions

address	_addrRoot	
		used in @2.IndexBasis.getInfo
		used in @2.IndexBasis.destruct
uint256	_codeHashData	
		used in @2.IndexBasis.getInfo

```
    address static _addrRoot;
```

```
    uint256 static _codeHashData;
```

6.3 Modifier Definitions

6.3.1 Modifier onlyRoot

Minor issue: Modifiers

Modifiers are often source of bugs ; using them should be avoided, especially when containing calls to `tvm.accept()` that would happen before later `require` that would be added in derived implementations.

Minor issue: Constants

Value ”100” should be defined as a constant (would improve readability)

```
    modifier onlyRoot() {
        require(msg.sender == _addrRoot, 100);
        tvm.accept();
        _;
    }
```

6.4 Constructor Definitions

6.4.1 Constructor

```
    constructor() public onlyRoot {}
```

6.5 Public Method Definitions

6.5.1 Function destruct

```
    function destruct() public onlyRoot {
        selfdestruct(_addrRoot);
    }
```

6.5.2 Function getInfo

```
    function getInfo() public view returns (address addrRoot,
        uint256 codeHashData) {
        addrRoot = _addrRoot;
        codeHashData = _codeHashData;
    }
```

Chapter 7

Contract IndexResolver

Contents

7.1 Overview

In file `IndexResolver.sol`

7.2 Variable Definitions

TvmCell	_codeIndex	
		used in @1.NftRoot.mintNft
		assigned in @1.NftRoot.:constructor
		used in @1.NftRoot.:constructor
		assigned in @7.Data.:constructor
		used in @7.Data.:constructor
		used in @6.IndexResolver._buildIndexCode

```
    TvmCell _codeIndex;
```

7.3 Public Method Definitions

7.3.1 Function resolveCodeHashIndex

```
    function resolveCodeHashIndex(
        address addrRoot,
        address addrOwner
    ) public view returns (uint256 codeHashIndex) {
        return tvm.hash(_buildIndexCode(addrRoot, addrOwner));
    }
```

7.3.2 Function resolveIndex

```
    function resolveIndex(
        address addrRoot,
        address addrData,
        address addrOwner
    ) public view returns (address addrIndex) {
        TvmCell code = _buildIndexCode(addrRoot, addrOwner);
        TvmCell state = _buildIndexState(code, addrData);
        uint256 hashState = tvm.hash(state);
        addrIndex = address.makeAddrStd(0, hashState);
    }
```

7.4 Internal Method Definitions

7.4.1 Function _buildIndexCode

```
    function _buildIndexCode(
        address addrRoot,
        address addrOwner
    ) internal virtual view returns (TvmCell) {
        TvmBuilder salt;
        salt.store(addrRoot);
        salt.store(addrOwner);
        return tvm.setCodeSalt(_codeIndex, salt.toCell());
    }
```

7.4.2 Function _buildIndexState

```
    function _buildIndexState(
        TvmCell code,
        address addrData
    ) internal virtual pure returns (TvmCell) {
        return tvm.buildStateInit({
            contr: Index,
            varInit: {_addrData: addrData},
            code: code
        });
    }
```

Chapter 8

Contract NftRoot

Contents

8.1 Overview

Major issue: No way to get funds back

Tokens sent to the contract are locked forever. They are sent when IndexBasis are destroyed and when contracts are deployed. The contract should provide a function to recover accumulated funds, or specify that these funds are used for long-term storage.

8.2 Contract Inheritance

DataResolver	
IndexResolver	

8.3 Variable Definitions

uint256	_totalMinted	
		assigned in @1.NftRoot.mintNft
		used in @1.NftRoot.mintNft
		used in @1.NftRoot.mintNft
address	_addrBasis	
		used in @1.NftRoot.destructBasis
		assigned in @1.NftRoot.deployBasis
		used in @1.NftRoot.deployBasis

```
    uint256 _totalMinted;
```

```
    address _addrBasis;
```

8.4 Constructor Definitions

8.4.1 Constructor

Minor issue: Variable initialization

The globals _totalMinted and _addrBasis are not initialized.

Minor issue: Code initialization

Anyone can build a NftRoot contract with a fake _codeData and _codeIndex ; consider checking the contract hashes.

Minor issue: Code initialization

It is usually a bad practice to initialize variables containing code cells in constructors, as deployment messages are limited to 16kB.

```
    constructor(TvmCell codeIndex, TvmCell codeData) public {
        tvm.accept();
        _codeIndex = codeIndex;
        _codeData = codeData;
    }
```

8.5 Public Method Definitions

8.5.1 Function deployBasis

Critical issue: Multiple calls may lead to leakage of `IndexBasis` contracts

`_addrBasis` is updated after every call to `deployBasis`, hence a call to this function forbids the deletion of the previously deployed `IndexBasis`. If only one `IndexBasis` contract should be created, the function should `require` that `_addrBasis` is null before deploying a new contract. Otherwise, `destructBasis` should receive `codeIndexBasis` as argument too, to be able to recompute the corresponding contract address to destruct.

Minor issue: Constants

Values ”0.5 ton”, ”0.4 ton” and ”104” should be defined as constants (would improve readability)

Minor issue: Variable name typo

Variable `codeHasData` should be named `codeHashData`.

```
    function deployBasis(TvmCell codeIndexBasis) public {
        require(msg.value > 0.5 ton, 104);
        uint256 codeHasData = resolveCodeHashData();
        TvmCell state = tvm.buildStateInit({
            contr: IndexBasis,
            varInit: {
                _codeHashData: codeHasData,
                _addrRoot: address(this)
            },
            code: codeIndexBasis
        });
        _addrBasis = new IndexBasis{stateInit: state, value: 0.4
            ton}();
    }
```

8.5.2 Function destructBasis

Critical issue: Function should not be public

This function can be called by anyone (no check on sender), so that anybody can destroy `IndexBasis` contracts; the authentification of destruct in IndexBasis is useless.

Minor issue: Check _addrBasis is not zero

Before destroying the contract pointed to by `_addrBasis`, the contract should check that the variable is not zero, or fail.

Minor issue: Set _addrBasis to zero

After destroying a contract, the function should set the `_addrBasis` variable to zero.

```
    function destructBasis() public view {
        IIndexBasis(_addrBasis).destruct();
    }
```

8.5.3 Function mintNft

Minor issue: Constants

Value ”1.1 ton” should be defined as a constant.

Minor issue: No check of `msg.value`

The contract should check that enough balance is carried within `msg.value` to deploy the contract. Otherwise, the `_totalMinted` value may be increased, but the contract will still fail to deploy the corresponding contract.

```
    function mintNft() public {
        TvmCell codeData = _buildDataCode(address(this));
        TvmCell stateData = _buildDataState(codeData, _totalMinted);
        new Data{stateInit: stateData, value: 1.1 ton}(msg.sender, _codeIndex);

        _totalMinted++;
    }
```