Audit of the DENS-SMV Project

By OCamlPro

August 20, 2021

Table of Major and Critical Issues

Contents

Chapter 1

Overview

1.1 Source code location

The source code is available at `https://github.com/RSquad/dens-smv` at branch master with hash code equal to fbdfe4bca3c372b02cacf9788b4ad37112d0da2c

1.2 Architecture

The project implements a system of voting for proposal with soft-majority, where voting rights are represented as a TIP-3 token (with a root token contract and wallets associated with each users).

The project contains the following contracts:

Demiurge: this contract is in charge of the deployment of `Padawan` contracts and `Proposal` contracts. It acts as a central hub for such interactions.

Proposal: this contract corresponds to a proposal for which users have to vote in a 7 days period, using their voting rights.

Padawan: this contract belongs to a user that is expected to vote for the different proposals. A wallet is associated with this contract to hold the user's voting rights.

Faucet: this contract is used to distribute the voting rights to user and deploy their wallets.

DemiurgeStore: this contract is in charge of storing the code of all deployable contracts and shared addresses. It acts as a central configuration for the infrastructure.

Base: this contract is inherited by all other contracts and contains constant definitions, and modifiers.

***Contract*Resolver:** these contracts are used to store the code of a particular contract, and provide functions to compute the addresses of such contracts and deploy them;

1.3 Message Sequences

1.3.1 Vote

- User sends a `vote` message (from multisig) to his `Padawan` contract to vote for a proposal
- The `Padawan` contract verifies that user has not yet used all his voting rights for this proposal, and sends a `vote` message to the proposal
- On reception of the `vote` message, the `Proposal` contract:
  - checks whether the proposal is still ongoing or not
  - sends a `confirmVote` message or a `rejectVote` message to the `Padawan` contract
  - checks the result of the vote (if it is ended or if enough votes have been issued)

1.3.2 Reclaim Deposits

- User wants to recover some deposits for voting rights that are not currently used in a proposal, user sends a `reclaimDeposit` message to his `Padawan` contract
- On reception of the `reclaimDeposit` message, the `Padawan` contract:
  - Checks whether enough voting rights are "free", and can be sent back
  - Send a `queryStatus` message to all active `Proposal` contracts
- On reception of a `queryStatus` message, a `Proposal` contract sends back an `updateStatus` message to the `Padawan` contract
- On reception of the `updateStatus` message, the `Padawan` contract frees the votes for ended proposals. If enough voting rights are "free", the tokens are sent back by sending a `transfer` message to the wallet contract

1.3.3 Providing Voting Rights to a User

- The administrator (owner of the `Faucet` contract) sends a `changeBalance` message to the `Faucet` contract associating voting rights to a pubkey
- The user sends a `claimTokens` message to the `Faucet` contract, providing the address of the wallet associated with his `Padawan` contract

- If the voting rights of the corresponding pubkey exists, the `Faucet` contract sends a `transfer` message to its wallet to transfer the tokens to the user's wallet
- The user sends a `depositTokens` message to his `Padawan` contract
- The `Padawan` contracts sends a `getBalance_InternalOwner` message to its associated wallet
- The wallet sends back a `onGetBalance` message to the `Padawan` contract with the updated voting rights of the user

Chapter 2

Contract Base

Contents

In file `Base.sol`

2.1 Modifier Definitions

2.1.1 Modifier accept

- Minor issue: this modifier is dangerous in general, although not used in this project, because a function using it is easier to target to drain the balance of the contract. It should be removed.

```
36      modifier accept {
37          tvm.accept();
38          _;
39      }
```

Chapter 3

Contract Demiurge

Contents

In file `Demiurge.sol`

The `Demiurge` contract acts as a central hub to create user contracts and proposal contracts.x

3.1 Modifier Definitions

3.1.1 Modifier checksEmpty

- Minor issue: this modifier is not used. It should be removed.

```
66      modifier checksEmpty() {
67          require(_allCheckPassed(), Errors.NOT_ALL_CHECKS_PASSED);
68          tvm.accept();
69          _;
70      }
```

3.2 Constructor Definitions

3.2.1 Constructor

- **Critical issue: Administrative Take-over in `Demiurge.constructor`**
  No test is performed to verify the sender in the case `msg.sender != address(0)`. An attacker could use it to deploy the contract himself for another user, providing its own `addrStore`, i.e. with his own code for most contracts.

- **Major issue: No initialization check performed in `Demiurge.constructor`**
  The `_createChecks` function gives the false feeling the checks are performed for initialization of the Padawan and Proposal codes. However, the checks are not performed in the functions where they would be required. No attempt is done to perform the same checks for addresses.

- Minor issue (readability): a number is used as an error, a constant should be defined instead.

- Minor issue (duplicate code): the check `addrStore != address(0)` is performed twice, the second one is useless.

```
82      constructor(address addrStore) public {
83          if (msg.sender == address(0)) {
84              require(msg.pubkey() == tvm.pubkey(), 101);
85          }
86          require(addrStore != address(0), Errors.
                STORE_SHOULD_BE_NOT_NULL);
87          tvm.accept();
88
89          if (addrStore != address(0)) {
90              _addrStore = addrStore;
91              DemiurgeStore(_addrStore).queryCode{value: 0.2 ton,
                    bounce: true}(ContractType.Proposal);
92              DemiurgeStore(_addrStore).queryCode{value: 0.2 ton,
                    bounce: true}(ContractType.Padawan);
93              DemiurgeStore(_addrStore).queryAddr{value: 0.2 ton,
                    bounce: true}(ContractAddr.DensRoot);
94              DemiurgeStore(_addrStore).queryAddr{value: 0.2 ton,
                    bounce: true}(ContractAddr.TokenRoot);
95              DemiurgeStore(_addrStore).queryAddr{value: 0.2 ton,
                    bounce: true}(ContractAddr.Faucet);
96          }
97
98          _createChecks();
99      }
```

3.3 Public Method Definitions

3.3.1 Function deployPadawan

- Minor issue: the function should check that the code of the Padawan contract was correctly initialized.

```
103     function deployPadawan(address owner) external onlyContract {
104         require(msg.value >= DEPLOY_FEE + 2 ton);
105         require(owner != address(0));
106         TvmCell state = _buildPadawanState(owner);
107         new Padawan{stateInit: state, value: START_BALANCE + 2 ton
                }(_addrTokenRoot);
108     }
```

3.3.2 Function deployReserveProposal

- Minor issue: this function should check that `_codePadawan` and `_codeProposal` have been correctly initialized
- Minor issue: there is no need to store `_codePadawan` in the proposal struct as it is already a global variable.

```
112     function deployReserveProposal(
113         string title,
114         ReserveProposalSpecific specific
115     ) external onlyContract {
116         require(msg.value >= DEPLOY_PROPOSAL_FEE);
117         TvmBuilder b;
118         b.store(specific);
119         TvmCell cellSpecific = b.toCell();
120 
121         NewProposal _newProposal = NewProposal(
122             0,
123             _addrDensRoot,
124             ProposalType.Reserve,
125             cellSpecific,
126             _codePadawan,
127             _buildProposalState(title)
128         );
129         _newProposals.push(_newProposal);
130 
131         _beforeProposalDeploy(uint8(_newProposals.length - 1));
132     }
```

3.3.3 Function getTotalDistributedCb

Critical issue: No permission check in `Demiurge.getTotalDistributedCb`
Anybody can send this message. An attacker could use it to force the deployment of all proposals with a wrong number of total votes.

- **Critical issue: No value check in `Demiurge.getTotalDistributedCb`**
  This function is in charge of deploying all pending proposals. It should check that the sender gave enough value to perform these deployments before the end of the action phase. Otherwise, the action phase may succeed, all proposal will be removed from the array of proposals, but the deployments will fail by lack of gas.
- Minor issue: this function should send back the remaining gas not consumed to its called, especially if the caller gave a lot of gas to account for the deployments of multiple proposals.

```
148     function getTotalDistributedCb(
149         uint128 totalDistributed
150     ) public override {
151         _totalVotes = totalDistributed;
152         _getBalancePendings -= 1;
153         _deployProposals();
154     }
```

3.3.4 Function updateAddr

- Minor issue: add `_passCheck` for addresses too.

```
174     function updateAddr(ContractAddr kind, address addr) external
            override onlyStore {
175         require(addr != address(0));
176         if (kind == ContractAddr.DensRoot) {
177             _addrDensRoot = addr;
178         } else if (kind == ContractAddr.TokenRoot) {
179             _addrTokenRoot = addr;
180         } else if (kind == ContractAddr.Faucet) {
181             _addrFaucet = addr;
182         }
183     }
```

Chapter 4

Contract DemiurgeStore

Contents

4.1 Overview

In file `DemiurgeStore.sol`

This contract is used to store "global" values for the whole infrastructure, such as the code of the contracts to be deployed and the addresses of some contracts.

4.2 General Minor-level Remarks

In general, the infrastructure would be safer if this contract would be implemented in two phases:

- In the Initialization phase, the contract is waiting for all the `setXXX` methods to be called to initialize all the fields. A bitmap can be used to keep the current initialization state. Any attempt to user a `getXXX` method should fail.
- In the Post-Initalization phase, the contract accepts to reply to `getXXX` methods, but `setXXX` methods are disabled.

There is also an inconsistency between the getters and setters: getters are generic (they take a `kind` as argument), whereas setters are specific (there is a different one for every kind).

4.3 Public Functions

4.3.1 Function queryAddr

- Minor issue: a `require` could be added to fail if `kind` is not a well-known kind.

```
43      function queryAddr(ContractAddr kind) public view {
44          address addr = _addrs[uint8(kind)];
45          IDemiurgeStoreCb(msg.sender).updateAddr{value: 0, flag: 64,
                bounce: false}(kind, addr);
46      }
```

4.3.2 Function queryCode

- Minor issue: a `require` could be added to fail if `kind` is not a well-known kind.

```
38      function queryCode(ContractType kind) public view {
39          TvmCell code = _codes[uint8(kind)];
40          IDemiurgeStoreCb(msg.sender).updateCode{value: 0, flag: 64,
                bounce: false}(kind, code);
41      }
```

4.3.3 Function setPadawanCode

- Minor issue: the infrastructure would probably be safer if the expected code hash is hardcoded in the source code, and check through a `require`

```
14      function setPadawanCode(TvmCell code) public signed {
15          _codes[uint8(ContractType.Padawan)] = code;
16      }
```

4.3.4 Function setProposalCode

- Minor issue: the infrastructure would probably be safer if the expected code hash is hardcoded in the source code, and check through a `require`

```
17      function setProposalCode(TvmCell code) public signed {
18          _codes[uint8(ContractType.Proposal)] = code;
19      }
```

Chapter 5

Contract Padawan

Contents

5.1 Overview

In file `Padawan.sol`

This contract is used by a user to collect his voting rights (within a token wallet), and vote for proposals. Voting rights can be added, and reclaimed if not currently used.

5.2 Variable Definitions

- Minor issue: there is no function to clean `_activeProposals`, i.e. to remove proposals that are ended. Currently, it is possible to use `reclaimDeposit` with argument 0 to do that. It would be better to introduce a `cleanProposals` function for that purpose.

```
21    address _addrTokenRoot;

23    TipAccount _tipAccount;
```

```
24      address _returnTo;

26      mapping(address => uint32) _activeProposals;

28      uint32 _requestedVotes;

29      uint32 _totalVotes;

30      uint32 _lockedVotes;
```

5.3 Public Method Definitions

5.3.1 Function confirmVote

- Minor issue: there is no real reason to call `_updateLockedVotes` here, as it could be called in `reclaimDeposit` instead. Indeed, `_lockedVotes` is only used when the deposit is reclaimed, so it will save the cost of the recomputation if the user votes for many proposals without reclaiming his tokens.

```
74      function confirmVote(uint32 votesCount) external onlyContract {
75          // TODO: better to check is it proposal or not
76          optional(uint32) optActiveProposal = _activeProposals.fetch
                (msg.sender);
77          require(optActiveProposal.hasValue());
78
79          _activeProposals[msg.sender] += votesCount;
80
81          _updateLockedVotes();
82
83          _owner.transfer(0, false, 64);
84      }
```

5.3.2 Function reclaimDeposit

- Minor issue: the user might want to use `votes=0` to cancel a withdrawal. In this case, this function should skip sending all `queryStatus` messages, unless the goal is to clean the `_activeProposals` mapping (we advise to create a function for that purpose).
- Minor issue: there is no reason to send `queryStatus` messages if the `_unlockDeposit` function was called, i.e. if the reclaim was already successful

```
103     function reclaimDeposit(uint32 votes, address returnTo)
            external onlyOwner {
104         require(msg.value >= 3 ton, Errors.MSG_VALUE_TOO_LOW);
105         require(votes <= _totalVotes, Errors.NOT_ENOUGH_VOTES);
106         require(returnTo != address(0));
```

```
107         _returnTo = returnTo;
108         _requestedVotes = votes;
109
110         if (_requestedVotes <= _totalVotes - _lockedVotes) {
111             _unlockDeposit();
112         } else {
113             _requestedVotes = 0;
114         }
115
116         optional(address, uint32) optActiveProposal =
                _activeProposals.min();
117         while (optActiveProposal.hasValue()) {
118             (address addrActiveProposal,) = optActiveProposal.get()
                    ;
119             IProposal(addrActiveProposal).queryStatus
120                 {value: QUERY_STATUS_FEE, bounce: true, flag: 1}
121                 ();
122             optActiveProposal = _activeProposals.next(
                    addrActiveProposal);
123         }
124     }
```

5.3.3 Function vote

- **Critical issue: Unlimited voting rights in `Padawan.vote`**
  An attacker can call this method several times in the same round and in consecutive rounds to vote several times for the same proposal, until the `Padawan.confirmVote` message is received. Fix: voting rights should be immediately decreased instead of waiting for `confirmVote`.

- **Major issue: Infinite locking of deposits in `Padawan.vote`**
  An attacker could send a faked proposal address to a user to make him vote for a non-existing proposal. It can generate a little increase in storage, but if the fix of the critical issue above is done, it could also lock the deposits forever, as the corresponding contract will never end and unlock the deposits. Fix: this method should take the title of the proposal in argument, computes the address of the proposal, and the contract should correctly deal with bounced messages.

```
55     function vote(address proposal, bool choice, uint32 votes)
           external onlyOwner {
56         require(msg.value >= VOTE_FEE, Errors.MSG_VALUE_TOO_LOW);
57         optional(uint32) optActiveProposal = _activeProposals.fetch
               (proposal);
58
59         uint32 activeProposalVotes = optActiveProposal.hasValue() ?
                optActiveProposal.get() : 0;
60         uint32 availableVotes = _totalVotes - activeProposalVotes;
61         require(votes <= availableVotes, Errors.NOT_ENOUGH_VOTES);
62
63         // TODO: better to remove
64         if (activeProposalVotes == 0) {
65             _activeProposals[proposal] = 0;
```

```
66          }
67
68          IProposal(proposal).vote
69              {value: 0, flag: 64, bounce: true}
70              (_owner, choice, votes);
71      }
```

5.4 Internal Method Definitions

5.4.1 Function _unlockDeposit

- Minor issue: this function should skip sending a message if `_requestedVotes` is 0.

```
146     function _unlockDeposit() private {
147         ITokenWallet(_tipAccount.addr).transfer
148             {value: 0.1 ton + 0.1 ton}
149             (_returnTo, _requestedVotes, 0.1 ton);
150         _totalVotes -= _requestedVotes;
151         _requestedVotes = 0;
152         _returnTo = address(0);
153     }
```

Chapter 6

Contract PadawanResolver

Contents

6.1 Overview

In file `PadawanResolver.sol`

This contract is inherited by contracts that need to deploy `Padawan` contract and verify that an address belongs to a deployed `Padawan` contract.

6.2 Internal Method Definitions

6.2.1 Function _buildPadawanState

- Minor issue: the state built in this function uses `address(this)` as one of the static variables for the contract. Yet, this contract is bound to be inherited by different contracts (here, at least `Demiurge` and `Proposal`), i.e. computed addresses will be different for different contracts. Instead, the value of the `_deployer` variable should be made explicit to the caller, by passing it as an argument of the function.
- Minor issue: this function should fail (`require`) if the `_codePadawan` variable has not yet been initialized. A global boolean could be used for that, set in an internal function initializing both global variables.

```
16      function _buildPadawanState(address owner) internal virtual
            view returns (TvmCell) {
17          return tvm.buildStateInit({
```

```
18              contr: Padawan,
19              varInit: {_deployer: address(this), _owner: owner},
20              code: _codePadawan
21          });
22      }
```

Chapter 7

Contract Proposal

Contents

7.1 Overview

In file `Proposal.sol`

This contract is used to collect the votes for a particular proposal. Votes are sent by `Padawan` contracts.

7.2 Constructor Definitions

7.2.1 Constructor

- Minor issue: there is a limitation to 16 kB for deploy messages. For this constructor, the deploy message contains the code of `Proposal`, the title and the code of `Padawan`. Thus, it might become a problem in the future.

There is already a mechanism in the infrastructure to download codes from the `DemiurgeStore`, this contract should take advantage of it.

- Minor issue: the `_voteCountModel` variable is initialized to `SoftMajority` in this constructor, but it is not used anywhere. Consider removing it if no future use.

```
25      constructor(
26          uint128 totalVotes,
27          address addrClient,
28          ProposalType proposalType,
29          TvmCell specific,
30          TvmCell codePadawan
31      ) public {
32          require(_deployer == msg.sender);
33
34          _addrClient = addrClient;
35
36          _proposalInfo.title = _title;
37          _proposalInfo.start = uint32(now);
38          _proposalInfo.end = uint32(now + 60 * 60 * 24 * 7);
39          _proposalInfo.proposalType = proposalType;
40          _proposalInfo.specific = specific;
41          _proposalInfo.state = ProposalState.New;
42          _proposalInfo.totalVotes = totalVotes;
43
44          _codePadawan = codePadawan;
45
46          _voteCountModel = VoteCountModel.SoftMajority;
47      }
```

7.3 Public Method Definitions

7.3.1 Function queryStatus

- Minor issue: a `require` should check that the message contains enough value to send the message.

```
162     function queryStatus() external override {
163         IPadawan(msg.sender).updateStatus(_proposalInfo.state);
164     }
```

7.3.2 Function vote

- Minor issue: a `require` should check that the message contains enough value to send back the reply;
- Minor issue: given that the constructor initializes `_proposalInfo.start` to `now`, it is impossible for this function to return the `VOTING_NOT_STARTED` error.

- Minor issue: the transaction could be aborted if a `onProposalPassed` message is sent by `_finalize` (in `_wrapUp`), together with `rejectVote` or `confirmVote` messages, because of the flag 64. Need to test what happens if two messages are sent by the same transaction, with one of them containing the flag 64.

```
55      function vote(address addrPadawanOwner, bool choice, uint32
            votesCount) external override {
56          address addrPadawan = resolvePadawan(addrPadawanOwner);
57          uint16 errorCode = 0;
58
59          if (addrPadawan != msg.sender) {
60              errorCode = Errors.NOT_AUTHORIZED_CONTRACT;
61          } else if (now < _proposalInfo.start) {
62              errorCode = Errors.VOTING_NOT_STARTED;
63          } else if (now > _proposalInfo.end) {
64              errorCode = Errors.VOTING_HAS_ENDED;
65          }
66
67          if (errorCode > 0) {
68              IPadawan(msg.sender).rejectVote{value: 0, flag: 64,
                    bounce: true}(votesCount, errorCode);
69          } else {
70              IPadawan(msg.sender).confirmVote{value: 0, flag: 64,
                    bounce: true}(votesCount);
71              if (choice) {
72                  _proposalInfo.votesFor += votesCount;
73              } else {
74                  _proposalInfo.votesAgainst += votesCount;
75              }
76          }
77
78          _wrapUp();
79      }
```

7.4 Internal Method Definitions

7.4.1 Function _buildPadawanState

- Minor issue (code repetition): instead of defining this function, the same function in `PadawanResolver` should take the `deployer` in argument.

```
154     function _buildPadawanState(address owner) internal view
            override returns (TvmCell) {
155         return tvm.buildStateInit({
156             contr: Padawan,
157             varInit: {_deployer: _deployer, _owner: owner},
158             code: _codePadawan
159         });
160     }
```

7.4.2 Function _finalize

- Minor issue: a `require` should check that the message contains enough value to send the `onProposalPassed` message. This check could be moved earlier in methods calling `_finalize`

```
81      function _finalize(bool passed) private {
82          _results = ProposalResults(
83              uint32(0),
84              passed,
85              _proposalInfo.votesFor,
86              _proposalInfo.votesAgainst,
87              _proposalInfo.totalVotes,
88              _voteCountModel,
89              uint32(now)
90          );
91
92          ProposalState state = passed ? ProposalState.Passed :
                ProposalState.NotPassed;
93
94          _changeState(state);
95
96          IClient(address(_addrClient)).onProposalPassed{value: 1 ton
                } (_proposalInfo);
97
98          emit ProposalFinalized(_results);
99      }
```

7.4.3 Function _softMajority

- **Critical issue: Division by 0 in `Proposal._softMajority`**
  If `totalVotes=1`, this function fails with division by 0. Fix: the function should check that `totalVotes>1`, and add special cases for `totalVotes=1` and `totalVotes=0`
- Minor issue (readability): use `returns (bool passed)` to avoid the need to define a temporary variable and to return it.

```
141     function _softMajority(
142         uint32 yes,
143         uint32 no
144     ) private view returns (bool) {
145         bool passed = false;
146         passed = yes >= 1 + (_proposalInfo.totalVotes / 10) + (no *
                ((_proposalInfo.totalVotes / 2) - (_proposalInfo.
                totalVotes / 10))) / (_proposalInfo.totalVotes / 2);
147         return passed;
148     }
```

7.4.4 Function _tryEarlyComplete

- **Major issue: Overflow in `Proposal._tryEarlyComplete`**
  If vote counts are expected to be in the full `uint32` range, `yes*2` and `no*2` can overflow. Fix: use `uint64` for parameters.

- Minor issue (readability): use `returns (bool completed, bool passed)` to avoid the need to define temporary variables and to return them.

```
101     function _tryEarlyComplete(
102         uint32 yes,
103         uint32 no
104     ) private view returns (bool, bool) {
105         (bool completed, bool passed) = (false, false);
106         if (yes * 2 > _proposalInfo.totalVotes) {
107             completed = true;
108             passed = true;
109         } else if(no * 2 >= _proposalInfo.totalVotes) {
110             completed = true;
111             passed = false;
112         }
113         return (completed, passed);
114     }
```

7.4.5 Function _wrapUp

- Minor issue: the function could immediately check if the state is above `Ended` to avoid recomputing again when the state cannot change anymore;

- Minor issue: there is no need to call `_changeState` before calling `_finalize`, as `_finalize` always calls `_changeState` and will thus override the state written in this function;

```
116     function _wrapUp() private {
117         (bool completed, bool passed) = (false, false);
118 
119         if (now > _proposalInfo.end) {
120             completed = true;
121             passed = _calculateVotes(_proposalInfo.votesFor,
                    _proposalInfo.votesAgainst);
122         } else {
123             (completed, passed) = _tryEarlyComplete(_proposalInfo.
                    votesFor, _proposalInfo.votesAgainst);
124         }
125 
126         if (completed) {
127             _changeState(ProposalState.Ended);
128             _finalize(passed);
129         }
130     }
```

Chapter 8

Contract ProposalResolver

Contents

8.1 Overview

In file `ProposalResolver.sol`

This contract is inherited by contracts that need to deploy `Proposal` contract and verify that an address belongs to a deployed `Proposal` contract.

8.2 Internal Method Definitions

8.2.1 Function _buildProposalState

- Minor issue: the state built in this function uses `address(this)` as one of the static variables for the contract. Yet, this contract is bound to be inherited by different contracts (although here, onlye `Demiurge` uses it), i.e. computed addresses will be different for different contracts. Instead, the value of the `_deployer` variable should be made explicit to the caller, by passing it as an argument of the function.
- Minor issue: this function should fail (`require`) if the `_codeProposal` variable has not yet been initialized. A global boolean could be used for that, set in an internal function initializing both global variables.

```
14      function _buildProposalState(string title) internal view
            returns (TvmCell) {
```

```
15          return tvm.buildStateInit({
16              contr: Proposal,
17              varInit: {_deployer: address(this), _title: title},
18              code: _codeProposal
19          });
20      }
```

Chapter 9

Contract Faucet

Contents

9.1 Overview

In file `Faucet.sol`

This contract is used to create the initial voting rights of all users. Voting rights are stored in TIP-3 token wallets created by a root token contract.

9.1.1 Constructor

- **Major issue: No permission checks in `Faucet.constructor`**
  The constructor is called with no check on the caller. As a consequence, an attacker could deploy the contract for another user (i.e. on the address for the pubkey of the other user), initializing the contract with his own token wallet address. This attack is still limited, but it might take some time for the user to really understand what is happening, and the user will have to restart the deployment on another pubkey. Fix: check that the pubkey signed the constructor message
- Minor issue: it would be safer to use the `DemiurgeStore` to recover the address of the token root contract.

```
22     constructor(address addrTokenRoot, address addrTokenWallet)
           public {
```

```
23          tvm.accept();
24          _addrTokenRoot = addrTokenRoot;
25          _addrTokenWallet = addrTokenWallet;
26      }
```

9.2 Public Method Definitions

9.2.1 Function claimTokens

- Minor issue: the contract should implement "on-bounced" callbacks on its token wallet to recover from sending tokens to not-yet-deployed token wallets.

```
28      function claimTokens(address addrTokenWallet) external override
            {
29          require(_balances[msg.pubkey()] != 0, Errors.INVALID_CALLER
                );
30          tvm.accept();
31
32          _totalDistributed = _balances[msg.pubkey()];
33          ITokenWallet(_addrTokenWallet).transfer(addrTokenWallet,
                _balances[msg.pubkey()], 0.1 ton);
34
35          delete _balances[msg.pubkey()];
36      }
```

9.2.2 Function deployWallet

- **Critical issue: No limitation on `Faucet.deployWallet`**
  A malicious user owning a balance in the contract can drain the contract balance by sending many `deployWallet` messages to the contract. Every message spends 0.5 ton of the balance. The owner of the contract has no way to block the attack, as the attack remains possible as long as the user does not use his balance with `claimTokens`. Fix: the contract could remember that a deployment request was already done by this user.

```
49      function deployWallet() external override {
50          require(_balances[msg.pubkey()] != 0, Errors.INVALID_CALLER
                );
51          tvm.accept();
52
53          ITokenRoot(_addrTokenRoot).deployEmptyWallet
54              {value: 0.5 ton, flag: 1, bounce: true}
55              (0, 0, msg.pubkey(), 0, 0.25 ton);
56      }
```