·Decurity·

CONTRACT VERIFICATION REPORT

TON CASH

Decurity, 2020

CONTENTS

1 GENERAL INFORMATION

This report contains information about the results of Phase 1 of the TIP-3 Token Contract Verification, conducted by Decurity in the period from 11/24/20 to 12/07/20.

1.1 Introduction

Tasks solved during the work are:

- Review of the TIP-3 Token architecture and security design,
- Development of the high-level description of the token business logic,
- Description of the possible attack vectors and weaknesses of the contracts.

1.2 Scope of Work

The analyzed token contracts are located in the following repository: https://github.com/tonlabs/ton-labs-contracts/tree/master/cpp (commit bad13bbd8223de5e0257f44514e9ca208de6a8b2).

1.3 Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token wallet owner, a contract).

1.4 Weakness Scoring

The findings in this report are scored by an expert evaluation, an impact of each vulnerability is calculated based on its ease of exploitation and severity (for the considered threats).

2 SUMMARY

As a result of this work, we have described the contract interfaces, variables and access model for further verification, along with the list of possible weaknesses and attacks applicable to the token. Also, during the verification of the token contracts several medium impact issues have been identified.

2.1 Discovered Issues

The purpose of this work did not include the security audit or contract code verification. However, during the analysis, we have noticed some errors.

2.1.1 [Low] Incorrect error code in TONTokenWallet transfer

There is the following check on the lines 49-50 of the **tokens-fungible/TONTokenWallet.cpp** file:

```
// Transfer to zero address is not allowed.
require(std::get<addr_std>(dest()).address != 0, error_code::not_enough_balance);
```

The thrown error is wrong, there should be a different error code (like **zero_dest_addr** in **tokens-nonfungible/TONTokenWalletNF.cpp**).

2.1.2 [Low] Inconsistent access control checks

The access control in **tokens-fungible/TONTokenWallet.cpp** file is done using the **check_owner** method but other contracts use inline checks which makes it inconsistent and may lead to an error. An example of such inline check on the lines 42-43 of the **tokens-nonfungible/TONTokenWalletNF.cpp**:

```
  void transfer(lazy<MsgAddressInt> dest, TokenId tokenId, WalletGramsType grams) {
    require(tvm_pubkey() == wallet_public_key_,
error_code::message_sender_is_not_my_owner);
```

2.1.3 [Medium] No address check in transferUTXO

The **transferUTXO** method in **tokens-utxo/TONTokenWalletUTXO.cpp** file does not check that the destination is not zero (unlike transfer methods in other two contracts). Here is the beginning of the method code starting from the line 43:

```
  void transferUTXO(int8 workchain_dest, uint256 pubkey_dest, int8 workchain_rest,
uint256 pubkey_rest,
                    TokensType tokens, WalletGramsType grams_dest) {
    require(tvm_pubkey() == wallet_public_key_,
error_code::message_sender_is_not_my_owner);


    // the function must complete successfully if token balance is less that transfer value.
    if (balance_ < tokens)
      return;


    tvm_accept();
```

2.2 Possible issues

The table below contains sample weaknesses (which can lead to intentional attacks or unintentional bugs) which might be found in the analyzed contracts.

Table 1. Possible issues

Issue	Risk Level	Probability
Memory Corruption	High	Low
Improper Access Control	High	Medium
Incorrect Arithmetics	High	Medium
Incorrect Error Handling	Medium	High
Denial of Service	High	Medium
Incorrect Gas Allocation	Medium	Medium

2.3 Possible attacks

The table below contains sample attacks which might be carried out by malicious attackers.

Table 2. Possible attacks

Attack	Risk Level	Probability
Contract code hijacking *Deploying a malicious wallet contract*	High	High
Financial fraud *A malicious manipulation of the business logic and balances, such as a reentrancy attack*	High	Medium
Unauthorized transactions *Neutralization of the access control leading to unauthorized messages accepted*	High	Medium
Attacks on implementation	High	Medium

Attack	Risk Level	Probability
Exploiting the weaknesses in the compiler or the runtime of the smart contracts		

3 GENERAL RECOMMENDATIONS

This section contains general recommendations how to fix discovered during the testing weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, sorted by the impact (first one is the most critical), technical recommendations for each finding can be found in section 5. Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to higher level.

3.1 Current weaknesses remediation

- Fix the issues described in the section 2.1 .
- Perform review and verification of all the business scenarios of the contracts,
- Make the interfaces of the contracts consistent:
    - The error codes should have the same values and names between different contract types,
    - The access checks should be done in the same way.
- Perform security audit of the whole architecture and the implementation.

3.2 Security process improvement

To build mature security process and avoid the losses, the contracts have to be thoroughly audited. The code review, dynamic and static security analysis must be performed.

Furthermore, we recommend launching a public bug bounty campaign.

4 DETAILS

4.1 Description

The analyzed smart contracts implement an alternative token standard (as opposed to e.g. ERC20) which is suitable for the TON blockchain.

There are also 2 alternative versions of the token: the non-fungible token (RootTokenContractNF and TONTokenWalletNF contracts respectively) and the UTXO extension of the fungible token (RootTokenContractUTXO and TONTokenWalletUTXO contracts).

4.1.1 Fungible Tokens

Each token wallet is a separate contract (TONTokenWallet instance) created by the root token (RootTokenContract instance). Token wallets store the respective balances in their own storage, and the transactions between them are verified.

Upon creation RootTokenContract instance stores the TONTokenWallet code, the total token supply, the total number of granted tokens (initially 0, additional minting available later).

After creation, the RootTokenContract can deploy the TONTokenWallet contracts and optionally grant them the tokens.

The flowchart below shows a brief algorithm for deploying an empty wallet. The blue blocks are called by the user.

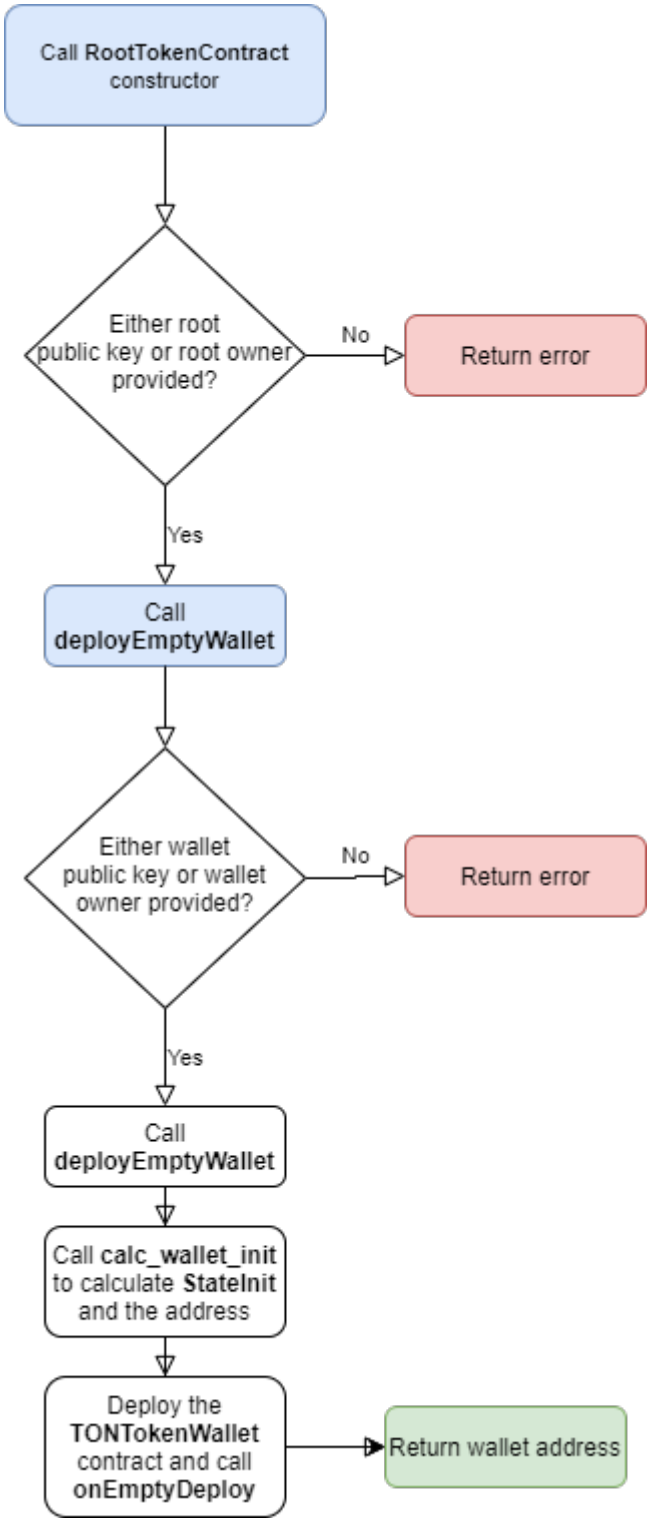

Image 1. The process of deploying an empty wallet

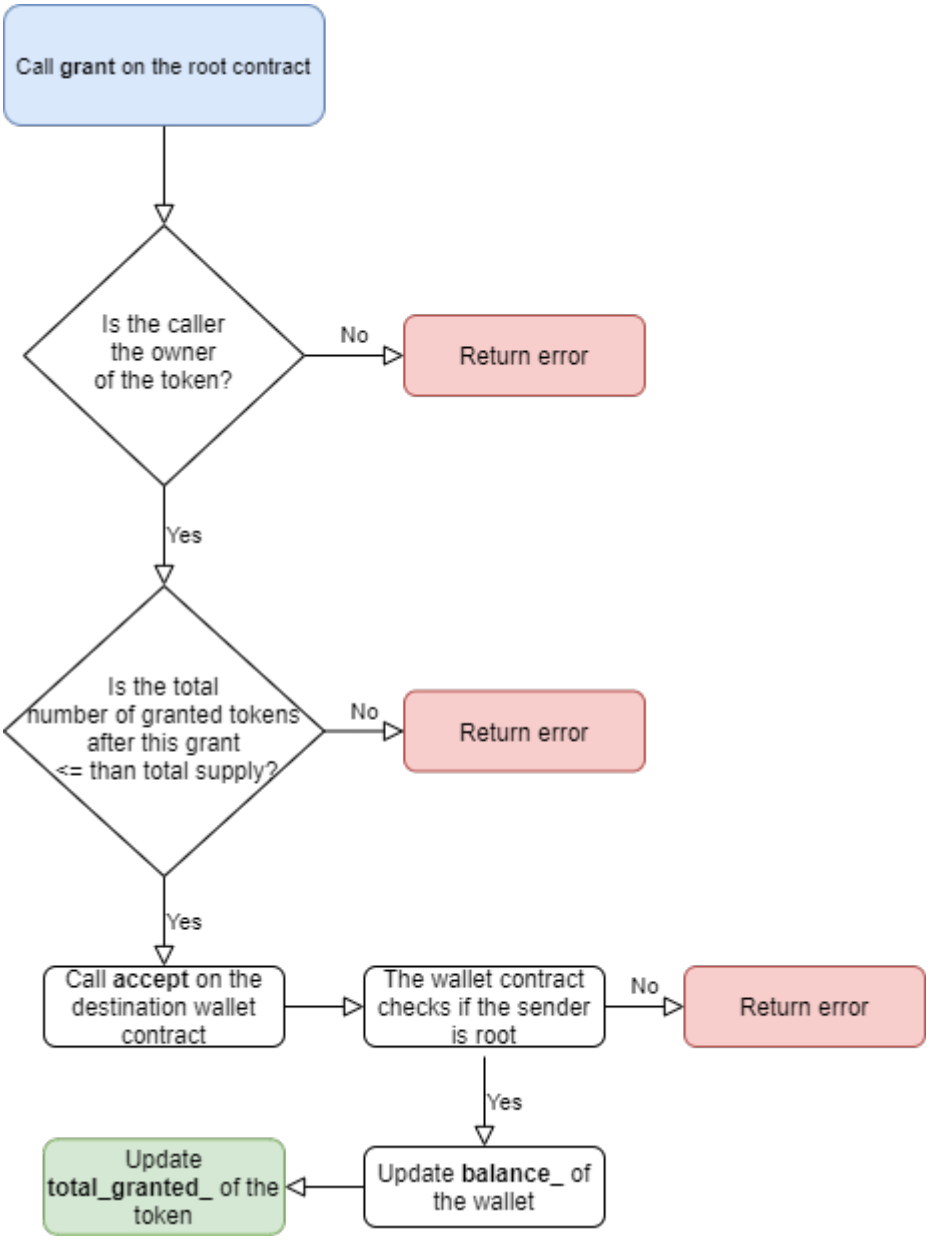

Image 2. The process of granting tokens to a wallet

The TONTokenWallet contract provides the functions to transfer tokens to other wallets. The external transfer calls can be done only by the wallet owner. The contract sends the internal transfer message to the destination wallet contract.

This internal message is handled by the internalTransfer method which checks that the message sender is a wallet by reconstructing the StateInit structure and calculating the expected wallet contract address.

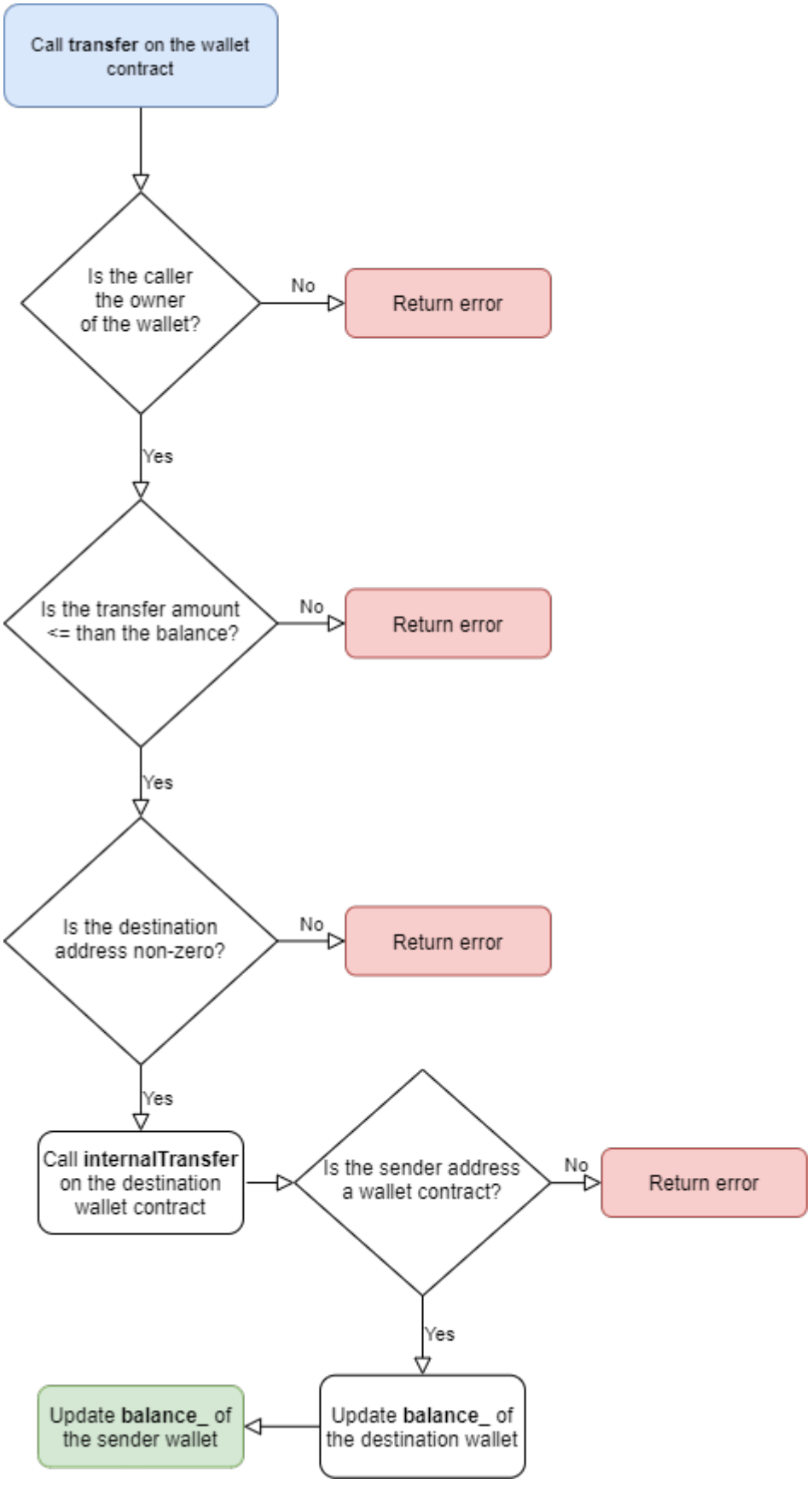

Image 3. The process of transferring tokens between wallets

The overall process of interaction between different parties (root token owner, wallet owner, root contract, wallet contracts) can be described by the following diagram:

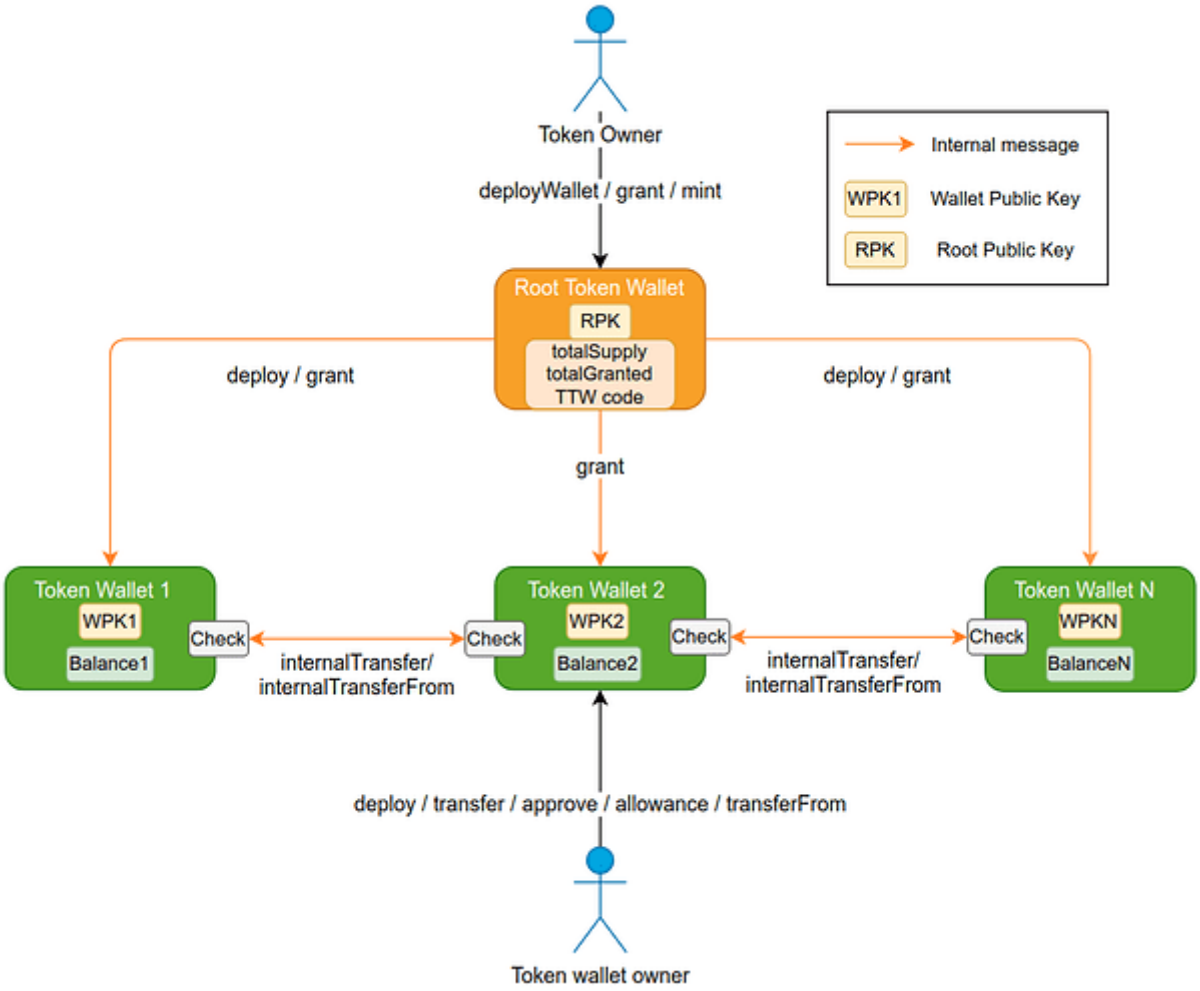

Image 4. The interaction of users and contracts

4.1.2 Non-Fungible Tokens

The non-fungible tokens work essentially the same as fungible except each token is assigned a unique identifier, and instead of updating the balances transferring the tokens actually moves them from one wallet to another (to the tokens_ array).

4.1.3 UTXO Tokens

The UTXO token contract is an extension of the fungible token contract. UTXO-based wallet allows to identify tokens as coins. The set of all UTXOs represents total token supply. Each coin has an owner and number of tokens it represents. So, one coin is one wallet. This model allows, for example, to process transactions with coins in parallel.

There is no allowance interface for UTXO wallet, and a TONTokenWallet contract cannot transfer tokens to an existing wallet. It can only receive tokens once, during the deploy message processing.

Every token transfer results in creation of at least 2 new wallets. The first holds the transferred number of tokens and the second holds the remaining token balance. After the transfer the original wallet must have a 0 token balance. Zero-balance destination wallets should not be created.

UTXO Root contract doesn't have grant method, it can create wallets using deployWallet only. Zero tokens in deployWallet is not accepted (zero_tokens_not_allowed error thrown).

To transfer tokens, a TONTokenWallet contract must perform the following steps within a single transaction:

1. Calculate the address of a new TONTokenWallet #1 using the public key provided by the destination wallet owner,

2. Deploy the new TONTokenWallet #1 at the address calculated at step 1,
3. Send some tokens to it in an internal deploy (internalTransfer) message,
4. Calculate the address of a new TONTokenWallet #2 using the new public key given by current wallet owner,
5. Deploy the new TONTokenWallet #2 to the address calculated at step 4 and send the remaining tokens to it in a deploy (internalTransfer) message.

Initial data of a UTXO wallet contains an additional utxo_received_ boolean flag set to false. When a wallet receives tokens, it is switched to true.

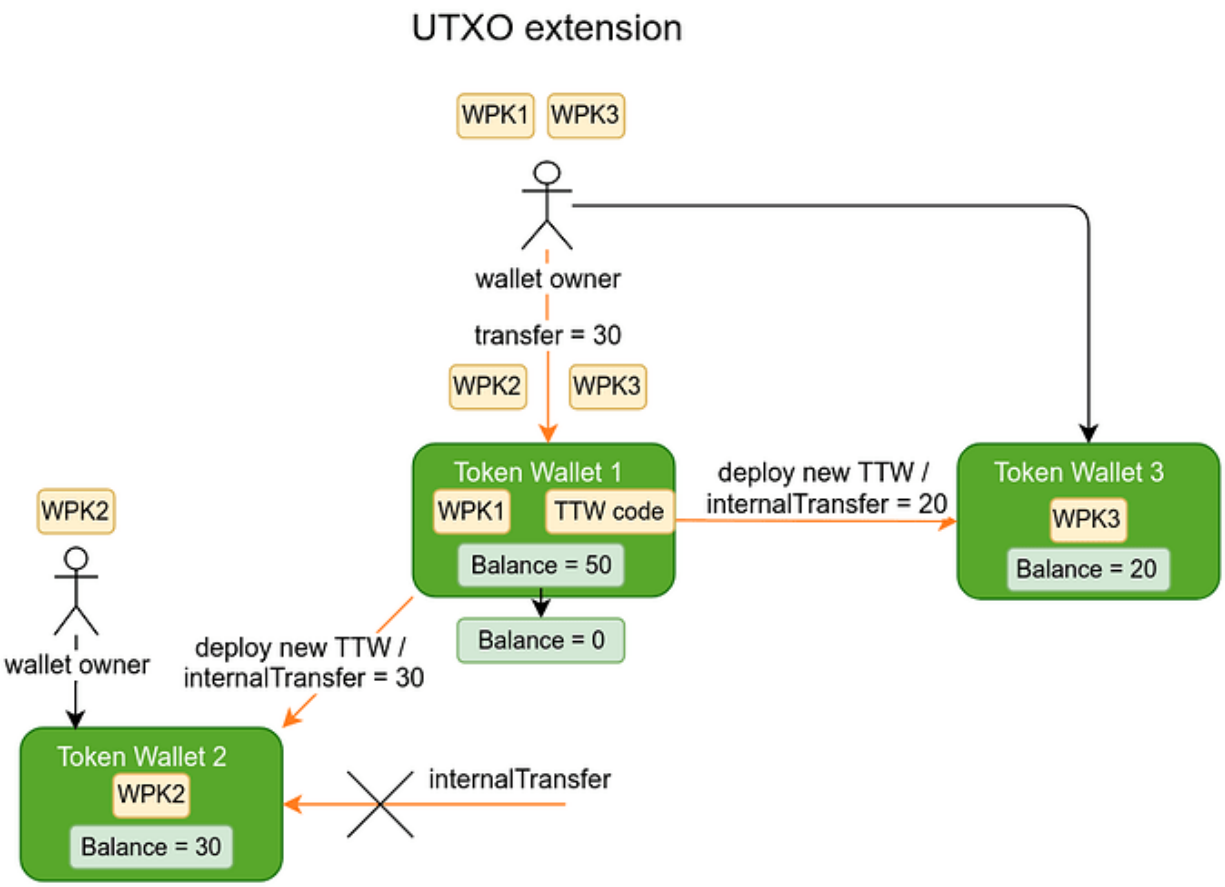

Image 5. The process of transferring tokens between UTXO wallets

4.2 Roles

The following roles have been assigned based on the logic of the contracts:

Name	Description
Anyone	Anyone on the blockchain
Token internal owner	Owner of the root token contract defined by owner_address_
Token external owner	Owner of the root token contract defined by root_public_key_
Token owner	Owner (internal or external) of the root token contract
Wallet internal owner	Owner of the root token contract defined by owner_address_
Wallet external owner	Owner of the root token contract defined by wallet_public_key_
Root contract	Root token contract (not owner)
Wallet contract	Wallet contract (not owner)
Allowance spender	An address with approved allowance

4.3 Contract interfaces

4.3.1 Fungible Tokens

4.3.1.1 RootTokenContract

The following variables are defined:

Name	Description
name_	Token name
symbol_	Token symbol
decimals_	Number of digits after the decimal
root_public_key	Public key of the contract creator
total_supply_	Total current number of supplied tokens (initial + minted)
total_granted_	Total current number of granted tokens (deployed to the wallets)
wallet_code_	The code of the TONTokenWallet contract
owner_address_	(Optional) Root contract owner address

The following action methods are implemented:

Name	Description	Required role
constructor	Create the root token contract	Anyone
deployWallet	Deploy the TONTokenContract	Token owner

deployEmptyWallet	Deploy the TONTokenContract with initial balance 0	Anyone
grant	Grant tokens to the TONTokenWallet instance	Token owner
mint	Increase the total supply	Token owner

The following getters are implemented:

Name	Description
getName	Return name_
getSymbol	Return symbol_
getDecimals	Return decimals_
getRootKey	Return root_public_key
getTotalSupply	Return total_supply_
getTotalGranted	Return total_granted_
getWalletCode	Return wallet_code_
getWalletAddress	Return the wallet address for a given public key

The following miscellaneous and support functions are implemented:

Name	Description
_on_bounced	Process the bounced message
_fallback	Process the unknown message

calc_wallet_init	Calculate the StateInit structure and the wallet address for a given public key
is_internal_owner	Check if the owner_address_ is defined
check_internal_owner	Check if the sender address equals owner_address_
check_external_owner	Check if the sender contract's public key equals root_public_key_
check_owner	Run check_internal_owner or check_external_owner
prepare_root_state_init_and_addr	Reconstruct the StateInit structure and calculate the expected root contract address

The following error codes are defined (variable names are self-explanatory):

- message_sender_is_not_my_owner
- not_enough_balance
- wrong_bounced_header
- wrong_bounced_args
- internal_owner_enabled
- internal_owner_disabled
- define_pubkey_or_internal_owner

4.3.1.2 TONTokenWallet

The following action methods are implemented:

Name	Description	Required role
constructor	Create the root token contract	Anyone
transfer	Transfer tokens	Wallet owner
accept	Accept tokens	Root contract
internalTransfer	Receive tokens	Wallet contract
approve	Approve allowance	Wallet owner
transferFrom	Receive using allowance	Wallet owner
internalTransferFrom	Transfer using allowance	Allowance spender
disapprove	Disapprove allowance	Wallet owner

The following getters are implemented:

Name	Description
getBalance_InternalOwner	Return balance_ (only for Wallet internal owner which is currently not set)
getName	Return name_
getSymbol	Return symbol_
getDecimals	Return decimals_
getWalletKey	Return wallet_public_key
getRootAddress	Return root_address_
getOwnerAddress	Return owner_address_ (or 0)
allowance	Return allowance_ (or 0)

The following miscellaneous and support functions are implemented:

Name	Description
_on_bounced	Process the bounced message
onEmptyDeploy	Set balance to 0
_fallback	Process the unknown message
expected_sender_address	Calculate the StateInit structure and the wallet address for a given public key
is_internal_owner	Check if the owner_address_ is defined
check_internal_owner	Check if the sender address equals owner_address_
check_external_owner	Check if the sender contract's public key equals root_public_key_
check_owner	Run check_internal_owner or check_external_owner
prepare_wallet_state_init_and_addr	Reconstruct the StateInit structure and calculate the expected wallet contract address

The following error codes are defined (variable names are self-explanatory):

- message_sender_is_not_my_owner
- not_enough_balance

- message_sender_is_not_my_root
- message_sender_is_not_good_wallet
- wrong_bounced_header
- wrong_bounced_args
- non_zero_remaining
- no_allowance_set
- wrong_spender
- not_enough_allowance
- internal_owner_enabled
- internal_owner_disabled

5 APPENDIX

5.1 About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained an expertise in blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.