Advanced Research & Development Center

Project S014
Automatic Formal Verification of Smart-Contracts

TON Merge Smart-Contracts Audit

Authors
Evgeny Shishkin
evgeny.shishkin@infotecs.ru

Moscow
July, 2021

Contents

1 Introduction **1**

2 TON Merger System **1**

2.1 System Purpose 1
2.2 Terms 2
2.3 System Architecture 3
2.4 Usage Scenario 4
2.5 User Capabilities 6
2.6 Relay Capabilities 6
2.7 DuneRootSwap Owner Capabilities 7
2.8 Implementation 7
2.9 Usage Contexts 8

3 Discovered Defects **8**

3.1 Balance draining using proposeChange 9
3.2 Balance draining using updateRelays 10
3.3 Balance draining using deployUserSwap 10
3.4 Imprecise balance check 11
3.5 Harmful onBounce handler 12
3.6 DuneUserSwap insufficient balance 13
3.7 Incorrect DePool address check 13
3.8 No duneUserSwapCode sanity check 14
3.9 Redundant accept() 15
3.10 Huge OrderId malfunction 15

4 Conclusion **16**

A Swap Order Processing Diagram **16**

1 Introduction

The document contains the program code audit report for the TON Merger project. The report consists of two parts. The first part describes the high-level architecture of the system. The second part containts the list of discovered defects. The list is sorted in the order of defects severity level.

We found several critical vulnerabilities that were able to break the system logic and drain smart-contract funds.

All discovered vulnerabilities were reported to the developers and received their acknowledgement. Some of them were fixed immediately, but some of them still not fixed[1]

The repository with smart-contracts source code is located at:
`https://gitlab.com/dune-network/ton-merge/-/tree/3341333052c303e6e316d5cc8107ea465d4964ac/contracts/free-ton`

2 TON Merger System

In this section, we describe the purpose and the overall architecture of the system, together with roles of participants.

2.1 System Purpose

The TON Merger system is designed to perform account swaps from Dune Network blockchain into FreeTON blockchain, unidirectionally.

During the swap, the coins of a user get blocked in one network and get funded on another network, using some fixed exchange rate.

The more long-standing goal of this project is to propose some typical *scalable* solution for merging exterior blockchains into FreeTON blockchain.

[1]Partially mitigated by the fact that those defects are not that relevant to the ongoing Dune-FreeTON merge process

2.2 Terms

In the document, we use several terms that are defined below.

Term	Description
User	A party that has an account within the blockchain. This party is able to transfer coins to other users, call smart-contract functions by sending messages and observing the blockchain state. Here, we assume, that the user has an account on both, DUNE NETWORK and FreeTON blockchains.
Coins	Native blockchain tokens.
Swap request	User intention to perform the swap of DUN tokens for TON Crystal tokens using the fixed exchanged rate. The intent is issued in a form of blockchain transaction that get sent by a user account into the *Swap* contract of Dune blockchain.
Relay owner	A participant with exclusive rights to manage the relay node.
DuneRootSwap owner	A participant with exclusive rights for *DuneRootSwap* smart-contract management.
Swap period	A time period during which a single swap request has to be executed. If the swap operation is not completed within the stated period, then it may be cancelled.
Merge period	A time period during which all swap requests must be completed. No transfers are possible after this period, except returning remaining funds into the giver smart-contract - the origin of funding coins.
Secret	A data string that is used as a cross-chain authentication mechanism for the user to both acknowledge swap operation on FreeTON side and verify that the operation is completed on Dune Network side.

Destination address	An account identifier within FreeTON blockchain that receives swap funds in case of a successful swap request processing.

2.3 System Architecture

The principal scheme of interaction between different system components is depicted on Fig.1.

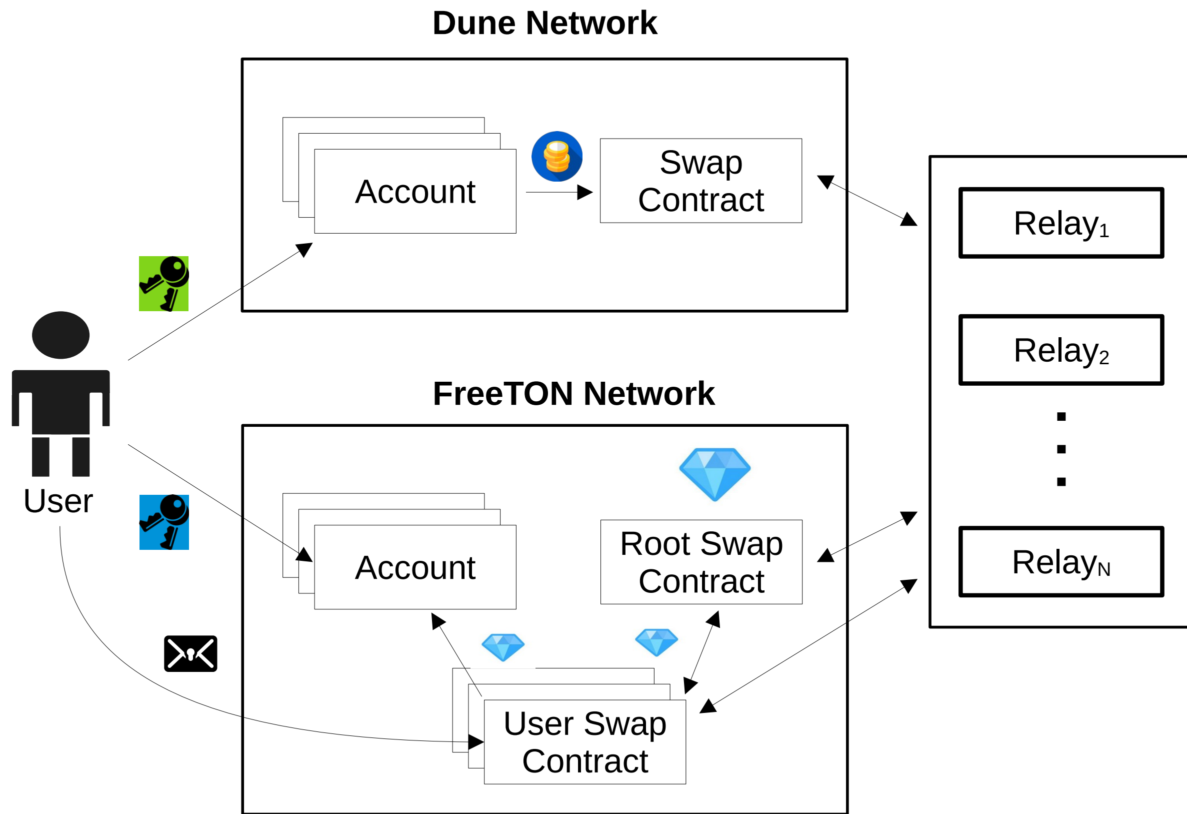


Figure 1: TON Merger System Architecture

We distinguish the following system components:

- **User**. Initiates a swap request, putting the destination address, tokens amount and secret hash inside the request among other things.
- **Account**. A blockchain account that is used either to initiate a swap request on Dune side, or to receive swap coins on FreeTON side.

- **Swap Contract.** The smart-contract in Dune Network responsible for receiving swap requests from Dune users. Requests have to contain enough coins for the swap operation to complete successfully. The sent coins will be blocked forever in the contract once the swap operation is successfully completed.
- **Relay.** A network node that acts as a retranslator between two blockchains. In particular, it tracks user swap requests on Dune side and translates them into swap request in FreeTON blockchain, sending a message into the *DuneRootSwap* contract.
- **Root Swap Contract.** FreeTON smart-contract that processes user swap requests from the relay nodes. It is the main storage and manager of the swap coins. A single swap request is issued in a form of *DuneUserSwap* smart-contract originating from *DuneRootSwap* contract.
- **User Swap Contract.** A smart-contract responsible for managing user swap request. After a successful confirmed deployment, it receives coins from the *DuneRootSwap* contract, and eventually sends them to the destination address.
- **Secret.** A data string that is used as a cross-chain authentication mechanism. User approves the swap operation by revealing it to the FreeTON smart-contract. This secret value get translated back into the Dune Network, proving the fact that the user accepted the swap on FreeTON network.

2.4 Usage Scenario

Here, we describe the main usage scenario.

In the beginning, interested parties, that is main stakeholders of both blockchains, do the following:

1. Deploy *Swap* contract on Dune Network.
2. Deploy *DuneRootSwap* contract on FreeTON blockchain. The contract is initially deployed with enough coins to cover all expected swap operations and user swap contract's management.

3. Deploy several relay nodes that will retranslate events between two blockchains. Relay nodes play the crucial role in the system, so, for greater reliability, it is expected to have several nodes working in parallel.

After the initial steps completed, the process works as follows:

1. The user issues a swap request in Dune Network by sending the transaction into the *Swap* contract. Among other things, they put destination address, swap coins amount, hash value of the secret into the request data.

2. Relay node tracks the issued swap request on the Dune Network and translates it into FreeTON: it sends the *deployUserSwap* transaction into the *DuneRootSwap* contract. As a result, new *DuneUserSwap* smart-contract get created. While working in ensemble with *DuneRootSwap*, it is responsible for managing the swap request.

3. Other Relay nodes also track the request. As far as they are not the first to discover the request, they have to acknowledge it instead of creating new swap request. It is done by sending *confirmOrder* transaction into the newly created *DuneUserSwap* contract.

4. After receiving enough confirmations from other relays[2], the DuneUserSwap requests funds from *DuneRootSwap* to cover the swap operation expenses.

5. The *DuneRootSwap* sends coins into the *DuneUserSwap* for the requested amount plus some coins to cover operational expenses.

6. The user, i.e. the swap request issuer, checks that the transfer amount matches their expected amount. If everything is correct, they reveal their secret value by sending *revealOrderSecret* into *DuneUserSwap* contract. The contract compares the hash value of the secret with the actual hash that was originally supplied in the request. If they match, the contract transfer coins to the destination address.

[2] This is a parameter of the system

7. Relay node discovers the secret revelation event on FreeTON blockchain and translates it back into Dune Network, into the *Swap* contract. As a result, the initial DUN coins get blocked forever in the *Swap* contract, preventing double spending.

The result of this whole operation is that DUN tokens get swapped for TON Crystals. The swap in other direction is not possible.

The diagram of this process is partially depicted in Appendix A.

2.5 User Capabilities

Users of the system may perform the following actions:

- Issue swap requests to exchange DUN tokens for TON Crystal tokens using the fixed exchange rate. The entry point is the *Swap* smart-contract on Dune Network.
- Acknowledge the swap request after its issuance on the Free-TON side by revealing the secret. As a result, swap coins get sent to the destination address of the swap request.
- Cancel the swap operation in case the operation is not completed within the swap period interval. In this case, the DUN coins are returned to the user.

2.6 Relay Capabilities

It is expected that relay nodes act according to some algorithm. However, if a manual intervention is required, the relay node owner could perform the following actions on behalf of its node:

- Confirm the swap request issued on Dune Network, on Free-TON side.
- Cancel the issued swap request with expired swap period.
- Initiate the swap coins transfer from the *DuneUserSwap* to the destination address.

- Reveal the swap request secret. By doing that, the swap request get the final approval. In this case, funds get transferred to the destination address. This scenario assume that the user gives their secret to the relay node owner.
- Inform the *DuneRootSwap* contract about the status of the node.
- Issue a Swap operation options change request. New options may contain: identifiers of new relay nodes, identifiers of relay node for removal, number of expected confirmations from relay nodes, the swap period and the merge period, etc.
- Vote for or against the proposed options change.
- Cancel the change request if the request was not processed in a timely manner.

2.7 DuneRootSwap Owner Capabilities

DuneRootSwap owner is capable in doing the following:

- Delete the *DuneRootSwap* contract, transferring all the remaining funds to the giver address, if the merge period is expired.
- Update swap options in the *DuneUserSwap* contracts in case they were changed recently.

2.8 Implementation

The whole system consists of the following components:

- *Swap* smart-contract, residing in Dune Network
- Relay node. It has several sub-modules we do not consider here.
- *DuneUserSwap* and *DuneRootSwap* smart-contracts of FreeTON blockchain.

2.9 Usage Contexts

The TON Merge system was designed as a typical solution for merging exterior blockchains into FreeTON.

In the proposed architecture, relay nodes play the crucial role of oracles between two blockchains. In case those nodes act maliciously, the whole merge process may be compromised.

To enhance the overall system reliability, the current TON Merger architecture supports up to 64 relay nodes working in parallel. At the same time, it is assumed that some number of those nodes may act maliciously, but the majority act honestly and correctly.

Regarding the Dune Network and FreeTON merge, the setting is more relaxed. There will be only 3 relay nodes and all of them are considered trusted.

We especially emphasize this fact, because some of founded defects may manifest itself only in the presence of malicious relays, and, hence, not that harmful to the ongoing Dune Network-FreeTON *merge process.*

3 Discovered Defects

We limit our audit work only with *DuneUserSwap* and *DuneRootSwap* contracts, leaving Dune Swap code and Relay nodes code aside.

The following severity ranking for defects is used:

1. **Critical**. The coins of smart-contracts could be stolen/spent in unexpected way. The whole merge process may be compromised.
2. **Medium**. Moderate coins loss and system malfunction, but with possibility of relatively quick recovery.
3. **Low**. Does not manifest itself in any way, but may lead to users or developers confusion.

Please note that in this ranking, we do not rely on probabilities of events, only on severity of their expected outcomes.

Below we list the discovered defects sorted in the order of their severity.

3.1 Balance draining using proposeChange

In *DuneRootSwap.sol*, within the *proposeChange* function, there is the following code:

```
function proposeChange(
  uint8 nreqs,
  uint256[] add_relays,
  uint256[] del_relays, ...) public returns (uint64)
{
  uint8 index = _isRelay( msg.pubkey() ) ;
  require( uint64(now) < g_merge_expiration_date,
           EXN_SWAP_EXPIRED );

  tvm.accept();

  optional(Change) opt_prev = _getChange( g_change_counter )
      ;
  require ( !opt_prev.hasValue(),
     EXN_CHANGE_ALREADY_PROPOSED );
  // ...
}
```

Severity:Critical.

Problem: The balance of the *DuneRootSwap* could be quickly drained in case of consecutive calls of this function together with passing long lists into *add_relays* and *del_relays*.

Explanation: The main vulnerability here is in the possible exception through require() occurring after the tvm.accept() call.
The *proposeChange* method is executed by an external message, and, hence, is paid from the balance of the contract.

If such call aborts with an error, the block validator will not put the call into the block. However, validator will withdraw the coins from the balance for the performed execution. Other validators will do the same thing after receiving this message: external messages processing is not coordinated in any way in FreeTON. This results in balance draining.

In case of small amount of computation steps and data cells, the loss is not that considerable. But if we pass large lists into the *add_relays* and *del_relays*, the balance draining tempo will be considerable: ≈ 5

TON Crystal for each call. If we let at least 1 malicious send requests into the *DuneRootContract*, its funds could be drained dramatically.

Status: Fixed. ✓

3.2 Balance draining using updateRelays

In *DuneRootSwap.sol*, there is the *updateRelays* function with the following code:

```
1 function updateRelays( address addr )
2   public view AuthOwnerOrRelay() {
3   tvm.accept();
4 
5   (uint8 nreqs, uint256[] relays, uint8[] indexes) =
      _getState();
6   IDuneUserSwap( addr ).updateAfterChange
7     (relays, indexes, nreqs, g_merge_expiration_date);
8   }
```

Severity:Critical.

Problem: The balance of *DuneRootSwap* could be drained if the function *updateRelays* get called with an empty address.

Explanation: The nature of this vulnerability is the same as in 3.1, i.e. exception after the contract accepts the message.
However, in this particular case, the situation is even worse because the *getState* call consumes a lot of gas and hence the speed of balance draining is even higher.

Status: Fixed. ✓

3.3 Balance draining using deployUserSwap

```
1 function deployUserSwap( uint256 pubkey, uint256 swap_hash )
    public returns ( address newSwap )
2  {
3   require( g_prev_user_key != pubkey ||
4            g_prev_swap_hash != swap_hash,
5            EXN_ALREADY_DEPLOYED );
6   uint8 relay_index = _isRelay( msg.pubkey() ) ;
7   require(uint64(now) < g_merge_expiration_date,
     EXN_SWAP_EXPIRED);
```

```
8     tvm.accept();
9       // ...
10  }
```

Severity:Critical.

Problem: The *DuneRootSwap* balance may be drained if the function *deployUserSwap* get called with the same parameters in an interleaving fashion, like:

$$(deploy(pk_1, sh_1), deploy(pk_2, sh_2), deploy(pk_1, sh_1))^{*}...$$

Explanation: The function *deployUserSwap* is responsible for issuing new user swap requests. For this purpose, it creates new instance of *DuneUserSwap* with the passed parameters each time it is called.

To create a single instance of a contract, you need to spend a relatively big amount of coins. Besides check on line 3, there is no mechanism in the code that prevents from redeploying *DuneUserSwap* several times. In case of second deployment, the coins attached to a constructor message will be returned to the *DuneRootSwap* in the bounce message. However, the gas spent on constructing such message will not be returned. This gives the ability to drain the balance by consecutive calls emitted in such a way to bypass the check on line 3.

Status: Not fixed. ✗

3.4 Imprecise balance check

```
1  function creditOrder( ... ) public override RootSet()
2   {
3       // ...
4       tvm.accept();
5       // ...
6       if( address(this).balance >= ton_amount ) {
7           IDuneUserSwap( computed_addr ).receiveCredit
8               {value: ton_amount + ( 1 ton ),
9                bounce: true,
10               flag: 0 } ();
11          credited = true ;
12      } else {
13        IDuneUserSwap( msg.sender ).creditDenied();
14      }
```

```
15      emit CreditOrder ( order_id , credited , ton_amount );
16    }
```

Severity:Critical.

Problem: In case of *DuneUserSwap* insufficient balance, the exception may be thrown after *accept()*, leading to balance draining.

Explanation: The check on line 6 does not count the scenario, when the following inequality holds

$$ton_amount + 1\,TON > balance > ton_amount$$

In this case, the exception will be thrown: the contract tries to send more than it possesses. It is thrown after *accept()*, so the draining scenario takes place.

Status: Fixed. ✓

3.5 Harmful onBounce handler

```
1  onBounce(TvmSlice slice) external {
2    uint32 functionId = slice.decode(uint32);
3    if(functionId == tvm.functionId(DuneUserSwap)){
4      uint8 relay_index = slice.decode(uint8);
5      _deployedConfirmed( relay_index );
6    }
7  }
```

Severity:Critical.

Problem: The unintentionally inserted *onBounce* handler distorts the swap order acknowledgement logic, potentially leading to draining scenario.

Explanation: Originally, there was the *onBounce* handler in *DuneRootSwap*, that was later commented out. In the process of fixing found defects, the author mistakenly uncommented the handler. The presence of this handler distorts the swaps acknowledgement logic: a malicious relay may decrease the number of unconfirmed deploys, and, by doing many swap requests, drain the *DuneRootSwap* balance.

Status: Fixed. ✓

3.6 DuneUserSwap insufficient balance

```
function confirmOrder( ... ) public
{
 // ...
   IDuneRootSwap( s_root_address )
      .orderConfirmedByRelay{ value: ROOT_MSG_FEE }
    ( tvm.pubkey(), s_swap_hash, order_id, msg.pubkey() );
 // ...
```

Severity:Medium.

Problem: Due to insufficient balance of the newly created *DuneUserSwap* contract, the swap process may stop until the manual user intervention.

Explanation: The *DuneUserSwap* contract get created with **1 TON** balance. After each confirmation from relay node, the contract sends acknowledgement message into *DuneRootSwap* attaching **0.1 TON**s value to it.
Now, you need only 9 confirmations to drain the balance of the contract. After that, it will stop working. The system is designed to support up to 64 relays, thus, the provided balance is insufficient and may lead to unintended stop of operation.

Status: Not fixed. ✗

3.7 Incorrect DePool address check

```
function _maybeTransferOrder() private {
 if ( g_depool == address(0) ){
   _orderStateChanged( STATE_TRANSFERRED ) ;
    g_dest.transfer({value:g_ton_amount, bounce:false, flag
       :1});
 } else { /* ... */ }
 // ...
}
```

Severity:Medium.

Problem: In the code, there is a check that tries to distinguish empty address value. However, the check is not complete. This may lead to distortion of coins transfer process that is hard to recover.

Explanation: At the end of swap operation, the *DuneUserSwap* contract transfers the expected amount of coins to the destination address. At the same time, if the user specified the DePool address, the funds will be tranfered into the DePool instead.

The problem arises at line 2 - the check for empty DePool address . In *TON Solidity* programming language, the address may be equal to *address(0)*, but also it could be undefined, and equal to **addr_none** value in this case. [3] The check ignores this fact. Now, if you put empty *DePool* address in the swap parameters like {depool_addr:“”}, then the check will not be able to distinguish it, and will try to send coins to the empty address, leading to exception.
The funds will not leave the balance of the contract, however, it may be challenging to fix this situation in a running system.

The correct check is done like this:

```
1 if (g_depool == address(0) || g_depool == address.
    makeAddrNone()) { ... }
```

Status: Not fixed. ✗

3.8 No duneUserSwapCode sanity check

```
1  constructor(
2                  address freeton_giver,
3                  uint256[] relays,
4                  uint8 nreqs,
5                  TvmCell duneUserSwapCode,
6                  uint64 merge_expiration_date,
7                  uint64 swap_expiration_time,
8                  bool testing
9                  ) OwnerSet() AuthOwner() public
10     {
11      // ...
12     }
```

Severity:Low.

Problem: The *DuneRootSwap* may be deployed with incorrect *duneUserSwapCode* field passed. In this case, this may not be obvious why the system does not work as expected.

[3]In contrast to Ethereum Solidity

Explanation: Due to absence of *duneUserSwapCode* sanity check, there is a scenario, when the *DuneRootSwap* contract deploys *DuneUserSwap* contract, but with different, unexpected code.

In our case, we were struggling to found out what is wrong with the system while it was deploying empty *DuneUserSwap* contracts. All calls were successful up to some point, but messages were not get transferred. A lot of time passed until we discovered the cause. This situation could have been avoided, if the sanity check was in place.

Status: Not fixed. ✗

3.9 Redundant accept()

```
1 function userSwapDeployed( ... ) public override
2     AuthUser(pubkey, swap_hash)
3 {
4     tvm.accept();
5     // ...
6 }
```

Severity:**Low**.

Problem: There are several places in the code with redundant accept() call: it is not needed there, and leads to confusion.

Explanation: Several methods in *DuneRootSwap* could be called only by internal message from the *DuneUserSwap* contract. However, in some of those functions, there is the *tvm.accept()* present. The effect of this call is achieved only for external messages, thus, it is redundant, and leads to confusion of developers.

Redundant calls are to be found in **DuneRootSwap.sol**, on lines **358, 412, 432, 450**, in **DuneUserSwap.sol** on line **436**.

Status: Not fixed. ✗

3.10 Huge OrderId malfunction

```
1 function confirmOrder(string orderId, ...) public {
2  // ...
3 }
```

Severity:**Low**.

Problem: When huge OrderId string is used as a swap identifier, the system *silently* stops working.

Explanation: To distinguish orders on the Dune side, each swap request is numbered with the unique *orderId* identifier. In the *DuneRootSwap* and the *DuneUserSwap* contracts, the OrderId identifier has a type **string**. It allows to write huge strings into this field. If you try to use huge identifiers (10Kb), the system stops working without any visible reasons. For example, the *confirmOrder* call successfully computes, but no messages are sent between contracts due to some reason.

We had no chance to investigate this issue further, however, we established the mere fact of a malfunction.

This whole issue could have been avoided if the *OrderId* had the type **uint**.

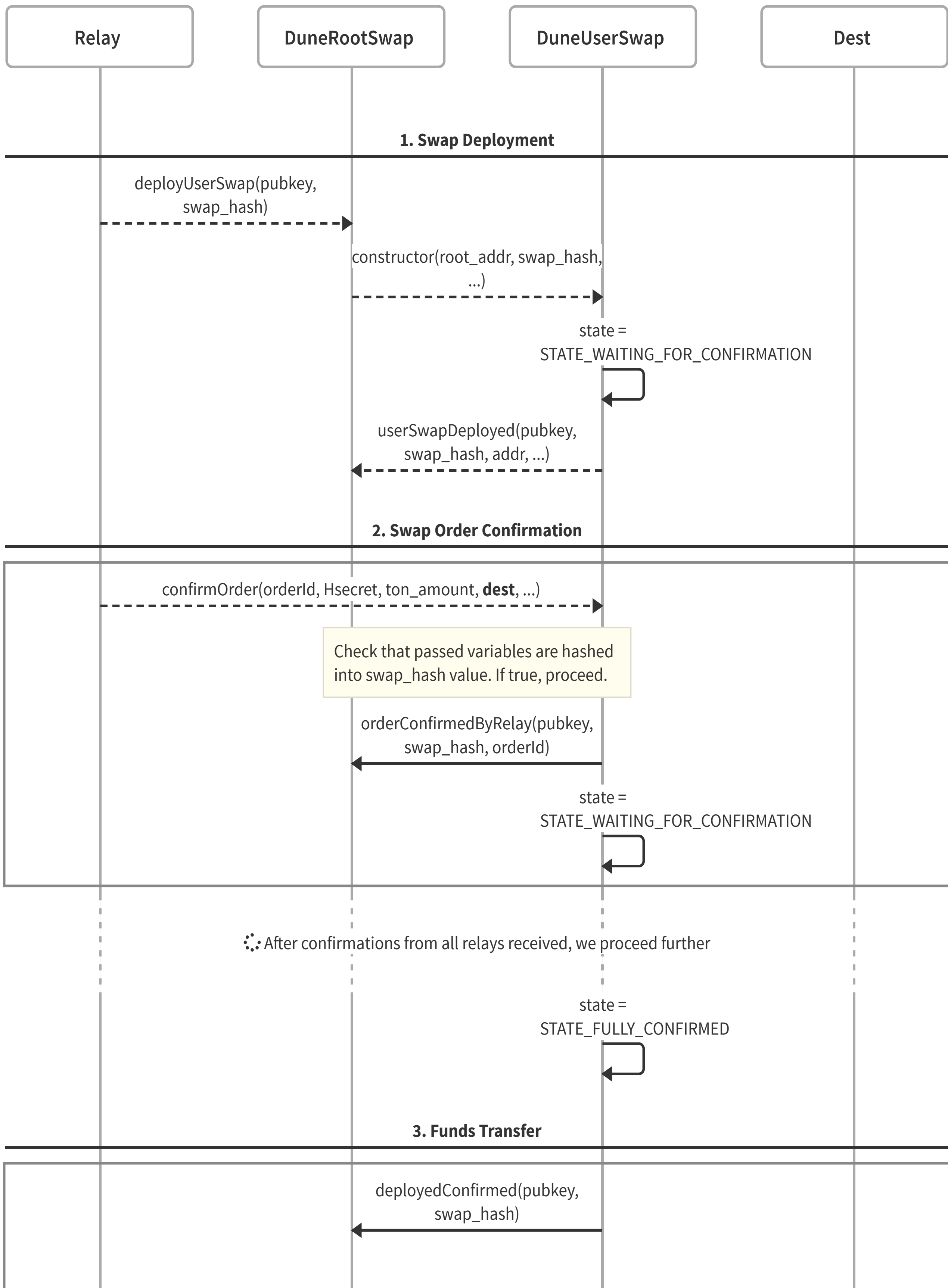
Status: Not fixed. ✗

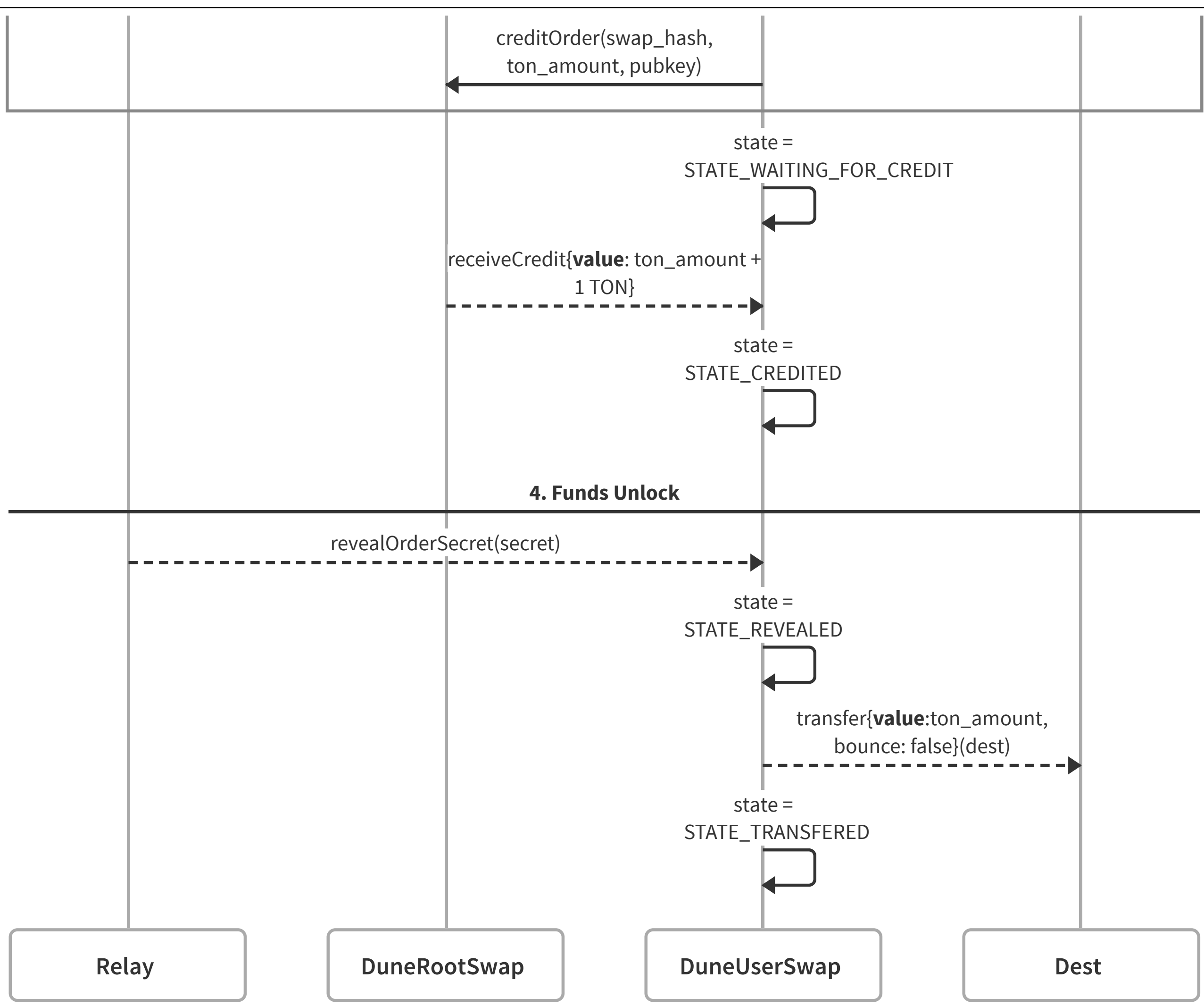
4 Conclusion

During the audit, we discovered **5 critical**, 2 medium, 3 - low severity vulnerabilities. All those issues were discussed with corresponding authors and received their acknowledgement.

The defects we were able to find were seen for the first time by the authors. This allows us to conclude that we were the first who discovered it among other audit attempts, if any.

A Swap Order Processing Diagram


Dune Merge Process
Relay
DuneRootSwap
DuneUserSwap
Dest
1. Swap Deployment
deployUserSwap(pubkey, swap_hash)
constructor(root_addr, swap_hash, ...)
state = STATE_WAITING_FOR_CONFIRMATION
userSwapDeployed(pubkey, swap_hash, addr, ...)
2. Swap Order Confirmation
confirmOrder(orderId, Hsecret, ton_amount, dest, ...)
Check that passed variables are hashed into swap_hash value. If true, proceed.
orderConfirmedByRelay(pubkey, swap_hash, orderId)
state = STATE_WAITING_FOR_CONFIRMATION
After confirmations from all relays received, we proceed further
state = STATE_FULLY_CONFIRMED
3. Funds Transfer
deployedConfirmed(pubkey, swap_hash)



creditOrder(swap_hash,
ton_amount, pubkey)
state =
STATE_WAITING_FOR_CREDIT
receiveCredit{value: ton_amount +
1 TON}
state =
STATE_CREDITED
4. Funds Unlock
revealOrderSecret(secret)
state =
STATE_REVEALED
transfer{value:ton_amount,
bounce: false}(dest)
state =
STATE_TRANSFERED
Relay
DuneRootSwap
DuneUserSwap
Dest
MADE WITH swimlanes.io