Dune Network - Free TON merger smart contract analysis

Prepared by Pruvendo

07/13/21

Executive Summary

In the course of this project, pursuant to the intention of the merger of Dune Network with Free TON, we shall perform an audit of the Solidity smart contracts developed for the Dune Network.

Source data

The source code of the smart contracts is available on Gitlab at:
https://gitlab.com/dune-network/ton-merge/-/tree/993377f5df07b4b42811ec995874bc9b986cea49/contracts/free-ton
However, at the moment of submission, new commits have been made.

Motivation

During the Dune Network -> Free TON merge, a supply of 9M TONs will be swapped in exchange for DUN tokens. Given this amount, the security of the smart contracts used in the infrastructure is essential. The goal of this contest is to identify vulnerabilities in the code that could lead to a leak of tokens or a non-working infrastructure.

Business level scenarios

The Dune to Free TON merger smart contract system contains two central contracts: RootSwap and UserSwap.

RootSwap manages the configuration of the merge (in particular, the list of users (relays / oracles) that can confirm transactions that replicate Dune orders on the Free TON side. It also deploys UserSwap contracts upon eligible relays' requests, accepts and sends tokens to UserSwaps.

RootSwap is deployed by the owner of a merge.

UserSwap handles requests from a user, to confirmed by a minimum number of relays set in the RootSwap configuration. It contains the sum that the user wants to receive from RootSwap, or a proposal to change RootSwap configuration. When a request is confirmed, tokens are sent to UserSwap, and then the user can request funds transfer to some other contract or a DePool, or the configuration of RootSwap is changed.

UserSwap is deployed by RootSwap upon request from an eligible user.

Scenario I: The Main Scenario

1. Deploy UserSwap (`deployUserSwap`)
2. Initialize UserSwap (`confirmOrder`, setting i.a. the secret)
3. Wait until a sufficient number of relays has confirmed this UserSwap
4. Check whether RootSwap has enough tokens (`requestCredit`)
5. RootSwap has enough tokens
6. Tokens are sent from RootSwap to UserSwap
7. The secret is checked (`revealOrderSecret`)

8. The secret check is successful
9. Tokens are sent to another contract or to a DePool

Scenario II: Not enough tokens on RootSwap

1. Deploy UserSwap (`deployUserSwap`)
2. Initialize UserSwap (`confirmOrder`, setting i.a. the secret)
3. Wait until a sufficient number of relays has confirmed this UserSwap
4. Check whether RootSwap has enough tokens (`requestCredit`)
5. RootSwap does not have enough tokens (`creditDenied`)
6. Check whether RootSwap has enough tokens
7. RootSwap does now have enough tokens
8. Tokens are sent from RootSwap to UserSwap
9. The secret is checked (`revealOrderSecret`)
10. The secret check is successful
11. Tokens are sent to another contract or to a DePool

Scenario III: DePool rejected the transfer

1. Deploy UserSwap (`deployUserSwap`)
2. Initialize UserSwap (`confirmOrder`, setting i.a. the secret)
3. Wait until a sufficient number of relays has confirmed this UserSwap
4. Check whether RootSwap has enough tokens (`requestCredit`)
5. RootSwap does not have enough tokens (`creditDenied`)
6. Check whether RootSwap has enough tokens
7. RootSwap does now have enough tokens
8. Tokens are sent from RootSwap to UserSwap
9. The secret is checked (`revealOrderSecret`)
10. The secret check is successful
11. Tokens are sent to a DePool
12. The DePool rejects the transfer
13. The transfer order is repeated
14. The DePool accepts the transfer

Scenario IV: We want to send received tokens back to RootSwap

1. Deploy UserSwap (`deployUserSwap`)
2. Initialize UserSwap (`confirmOrder`, setting i.a. the secret)

3. Wait until a sufficient number of relays has confirmed this UserSwap
4. Check whether RootSwap has enough tokens
5. RootSwap does not have enough tokens (`creditDenied`)
6. Check whether RootSwap has enough tokens
7. RootSwap does now have enough tokens
8. Tokens are sent from RootSwap to UserSwap
9. We send them back (`cancelOrder` or `closeSwap`)
10. Tokens are sent back to RootSwap

Scenario V: Proposal to change UserSwap properties

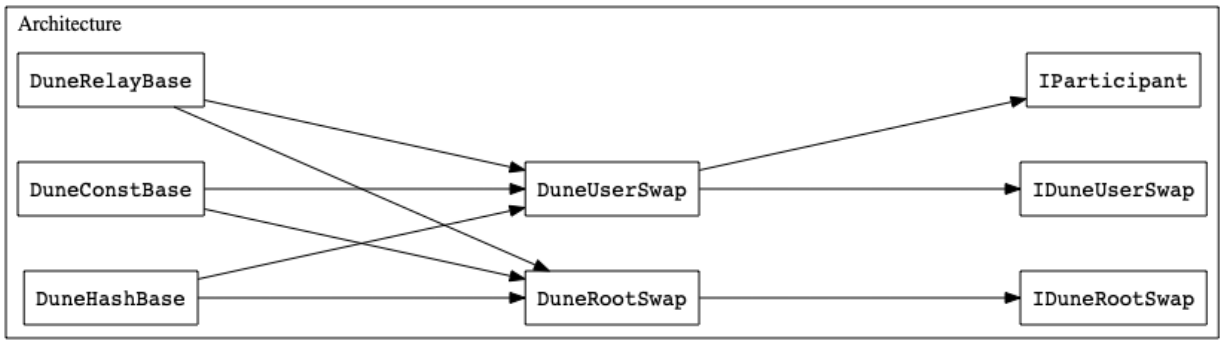
1. Deploy UserSwap (`deployUserSwap`)
2. Initialize UserSwap (`confirmOrder`, setting i.a. the secret)
3. Create and send a proposal (`proposeChange`) for one or more RootSwap parameters:
    a. Setting a new amount of votes required for confirmation (`nreqs`)
    b. Adding new relays (`add_relays`)
    c. Deleting existing relays (`del_relays`)
    d. Changing merge expiration date (`merge_expiration_date`)
    e. Changing the maximum allowed amount of unconfirmed deployments (`max_unconfirmed_deployments`)
4. Wait until a sufficient number of relays has voted for the proposal (`acceptChange` or `rejectChange`) and the outcome is known:
    a. If more or equal than `nreqs` relays have voted for the proposal, the proposal is accepted and the parameters of UserSwap are updated
    b. If more or equal than `nreqs` relays have voted against the proposal, the proposal is rejected and no changes to the parameters of UserSwap occur

The contracts have a sufficient number of getters to implement the scenarios described above.

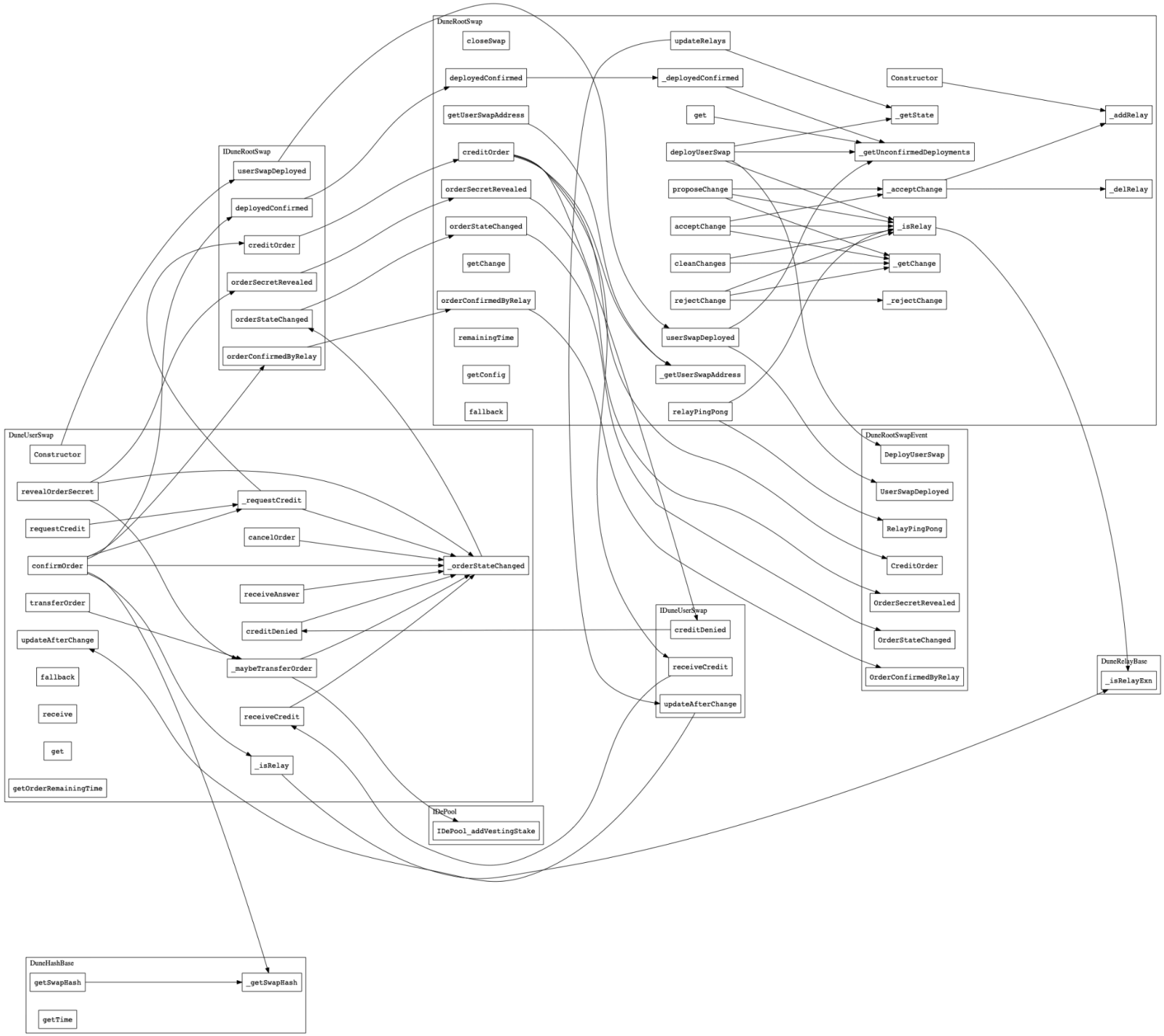
Program structure

This section contains the description of the contract hierarchy, the call graph, and the analysis of the contract public functions in a tabular form.

Description scheme below presents the contracts and interfaces of the system. *Base contracts are ancestors of the two main contracts and I* are interfaces which are for external usage. IParticipant interface is used to communicate with DePools.


Architecture
DuneRelayBase
DuneConstBase
DuneHashBase
DuneUserSwap
DuneRootSwap
IParticipant
IDuneUserSwap
IDuneRootSwap

Call graph for contracts' inter-communication



Functions							
Names	Access	Requires/sendErrors	Exceptions	Gas	Transfers	Crosscontract	assessment
constructor (DuneRootSwap)	**OwnerSet AuthOwner**	EXN_BAD_NUMBER_OF_RELAYS EXN_BAD_NUMBER_OF_CONFIRMATIONS _addRelay contains require		**tvm.accept**			**check balance before accept**
closeSwap	**AuthOwner**	EXN_SWAP_NOT_EXPIRED		**tvm.accept**			**check balance before accept**
_getState							
deployUserSwap	**_isRelay**	EXN_ALREADY_DEPLOYED EXN_SWAP_EXPIRED EXN_MAX_UNCONFIRMED_DEPLOYMENTS		**tvm.accept**	2 ton	**calls DuneUserSwap.constructor**	**check balance before accept**
relayPingPong	**_isRelay**			**tvm.accept**			**check balance before accept**
deployedConfirmed	**AuthUser**			**+ ROOT_MSG_FEE tvm.accept**		**called by DuneUserSwap.confirmOrder**	
creditOrder	**RootSet AuthUser**	EXN_SWAP_EXPIRED EXN_AUTH_FAILED		+ ROOT_MSG_FEE	ton_amount + (1 ton)	**called by DuneUserSwap._requestCredit calls DuneUserSwap.receiveCredit calls DuneUserSwap.creditDenied**	
userSwapDeployed	**AuthUser**			**+ ROOT_MSG_FEE tvm.accept**		**called by DuneUserSwap.constructor**	
orderStateChanged	**AuthUser**			**+ ROOT_MSG_FEE tvm.accept**		**called by DuneUserSwap._orderStateChanged**	

orderSecretRevealed	**AuthUser**			**+ ROOT_MSG_FEE tvm.accept**		**called by DuneUserSwap.revealOrderSecret**	
orderConfirmedByRelay	**AuthUser**			**ROOT_MSG_FEE**		**called by DuneUserSwap.confirmOrder**	
cleanChanges	**_isRelay**	EXN_CHANGE_NOT_EXPIRED CHANGE_EXPIRATION_TIME		**tvm.accept**			**check balance before accept**
proposeChange	**_isRelay**	EXN_BAD_NUMBER_OF_CONFIRMATIONS EXN_BAD_NUMBER_OF_CONFIRMATIONS EXN_BAD_NUMBER_OF_RELAYS EXN_SWAP_EXPIRED EXN_CHANGE_ALREADY_PROPOSED		**tvm.accept**			**check balance before accept**
acceptChange	**_isRelay**	EXN_WRONG_CHANGE_ID EXN_ALREADY_VOTED_FOR_CHANGE _addRelay contains require	**uncaught exception opt_change.get()**	**tvm.accept**			**check balance before accept**
rejectChange	**_isRelay**	EXN_WRONG_CHANGE_ID EXN_ALREADY_VOTED_FOR_CHANGE	**uncaught exception opt_change.get()**	**tvm.accept**			**check balance before accept**
updateRelays	**AuthOwner**	EXN_CHANGE_ALREADY_PROPOSED		**tvm.accept**		**calls DuneUserSwap.updateAfterChange**	**check balance before accept**
getChange			**uncaught exception opt_change.get()**				
getUserSwapAddress							
remainingTime							

getConfig							
get (DuneRootSwap)							
constructor (DuneUserSwap)	**AuthRoot**			2 ton	ROOT_MSG_FEE	**called by DuneRootSwap.constructor calls DuneRootSwap.userSwapDeployed**	
confirmOrder	**_isRelay**	EXN_SWAP_EXPIRED EXN_ALREADY_CONFIRMED EXN_ALREADY_CONFIRMED		**tvm.accept**	ROOT_MSG_FEE x3	**calls DuneRootSwap.orderConfirmedByRelay calls DuneRootSwap.deployedConfirmed calls DuneRootSwap.creditOrder**	**check balance before accept**
requestCredit	**AuthOwnerOrRelay**	EXN_UNKNOWN_ORDER EXN_SWAP_EXPIRED		**tvm.accept**	ROOT_MSG_FEE	**calls DuneRootSwap.creditOrder**	**check balance before accept**
revealOrderSecret	**AuthOwnerOrRelay**	EXN_NOT_YET_CREDITED EXN_MAX_FAILED_REVELATIONS		**tvm.accept**	g_ton_amount + DEPOOL_FEE ROOT_MSG_FEE	**calls DuneRootSwap.orderSecretRevealed calls IDePool.addVestingStake**	**check balance before accept**
transferOrder	**AuthOwnerOrRelay**	EXN_NOT_YET_REVEALED EXN_BALANCE_TOO_LOW		**tvm.accept**		calls IDePool.addVestingStake	**check balance before accept**
cancelOrder	**AuthOwnerOrRelay**	EXN_NOT_CANCELLABLE EXN_BALANCE_TOO_LOW		**tvm.accept**	g_ton_amount		**check balance before accept**
receiveCredit	**AuthRoot**			ton_amount + (1 ton)		**called by DuneRootSwap.creditOrder**	
creditDenied	**AuthRoot**			**tvm.accept**		**called by DuneRootSwap.creditOrder**	**check balance before accept**
closeSwap	**AuthOwnerOrRelay**	EXN_SWAP_NOT_EXPIRED		**tvm.accept**			**check balance before accept**
receiveAnswer	**depool**	EXN_AUTH_FAILED					

		EXN_UNEXPECTED_ANSWER_FROM_DEPOOL					
updateAfterChange	**AuthRoot**			?		**called by DuneRootSwap.updateRelays**	
get (DuneUserSwap)							
getOrderRemainingTime							

Issues / Error list

The contracts have been analyzed on logical mistakes, each function - on readability, adequate variables' names and functional encapsulation. Also, duplicate code was searched to prevent DRY[1] principle violation. In some cases when a certain algorithm is implemented, common sense has been used to realize the correspondence of the final algorithm and potential goal. However, when common sense contradicts with contract code, we cannot assert that as a bug, we mark that place for more attention from the developers side.

The majority of functions are good readable, variables' names are self-explanatory, the contract logic can be retrieved from its code with common knowledge of solidity programming language and the subject of contract usage area.

The next section contains the issues found with explanation and remarks of what kind of issue we treat it. The final decision of the level of severity is left on after-contest period, when all potential issues are revealed by all teams participating in the audit process. The section is divided into 2 parts corresponding to one of the two contracts presented: `DuneRootSwap` and `DuneUserSwap`. Sometimes we use abbreviation *root* contract for the `DuneRootSwap` and *user* contract for `DuneUserSwap`. Some statements below correspond to a particular function, some can be applied for several functions, some - for functions communication, some - for inter-contract communication. We are trying to maximally identify the issue and point to it, however it is not possible in every case, and in that case we describe the potential issue to be realizable without contract investigation for external readers.

`DuneUserSwap` is intended for use with a specific root contract, we do not investigate the contract behaviour with a different root.

[1] https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

DuneRootSwap		
Function or member	Issues	Cumulative severity
constructor	• `swap_expiration_time` is not required to be more than some reasonable positive constant, which would make it business explained • `merge_expiration_date` is not required to be more than `now`, allowing it to be in the past which seems counter-productive • duneUserSwapCode can be arbitrary, which allows to run root contract with malicious user code image • `freeton_giver` address is not required to be non-zero which would make it usable • `freeton_giver` is used only for `selfdestruct` operation, which makes the name unclear	minor
_addRelay	• `g_relay_counter` is global relays counter, preventing having more than `MAX_RELAYS` (64) relays during all contract lifetime	Not an issue, just a note important for further discovery

deployUserSwap	• prevents from double run with the code
	`require( g_prev_user_key != pubkey ||`
	`g_prev_swap_hash != swap_hash,`
	`EXN_ALREADY_DEPLOYED );`
	however it doesn't prevent from incorrect value of
	`count = _getUnconfirmedDeployments( relay_index )`
		major

	when called **twice in a row** with the same `pubkey` and different `swap_hash` - as the count increment is performed only in callback function `userUserDeployed` ● when called twice but not in row with the same arguments the second event emission `DeployUserSwap( pubkey, swap_hash, newSwap )` may be confusing ● In a case of malicious relay (or just misset) it can call the function many times with correctly specified arguments (to pass requires) and as the function is in accept mode it will use the contract balance for useless work; we suggest to perform this deployment for the caller funds	

creditOrder	● modifier `RootSet (require( g_root_address != address(0), EXN_NOT_INITIALIZED );)` seems to be redundant as `g_root_address = address( this )` initialized in constructor and never modified
	● Lines
	`constructor (DuneRootSwap) "OwnerSet AuthOwner" "EXN_BAD_NUMBER_OF_RELAYS EXN_BAD_NUMBER_OF_CONFIRMATIONS`
	`_addRelay require after tvm.accept" tvm.accept address computed_addr = _getUserSwapAddress( pubkey, swap_hash );`
	`require( msg.sender == computed_addr, EXN_AUTH_FAILED );`
	repeat the logic of AuthUser modifier and violate DRY principle
	● the balance is required to be more than ton_amount however ton_amount+1ton is sent
	● the function can empty the contract balance which make it unusable, we
		major

	suggest to MINIMUM_BALANCE constant to be introduced and used to check the balance before transfers	
orderStateChanged, orderSecretRevealed, orderConfirmedByRelay	• very similar functions, however first two work in accept mode, the third - not, we cannot find the reason for such difference	minor

acceptChange
rejectChange
_acceptChange
_rejectChange
	• if relay votes for expired change, the contract first cleans it by `_getChange( changeId )` call and then falls with uncaught exception at `opt_change.get()` and state would not be saved which forces user to call `cleanChanges` in advance which is counter-intuitive
	• Functions can be perceived as pure (as they have `change` as an argument) however don't save the state, as they modify the g_changes member
	• The voting mechanism seems imperfect:
	  ○ those part of relays who vote faster - win, not waiting for others (g_nreqs is not required to be more than a half)
	  ○ one relay can vote both for and against
	• delete g_changes[change.id]; (lines `633` and `650`) are written twice
	• `change.merge_expiration_date` is not required to be more than now, which can make contract unusable
	• `change.merge_expiration_date` has similar name to `merge_expiration_date` constructor argument however corresponds by meaning to the sum of `merge_expiration_date + swap_expiration_time` which can be confusing
	• uint64 bit = uint64(1) << index ; will cause integer overflow if index = 64:
	  ○ index is defined as `uint8 index = _isRelay( msg.pubkey() ) ;`
	  ○ then at _isRelayExn: `optional(uint8) opt_index = g_relays.fetch ( key );`
	  ○ and before at _addRelay: `require( g_relay_counter <`
		critical

	`MAX_RELAYS, EXN_TOO_MANY_RELAYS );`
	`g_relays[key] = g_relay_counter++ ;`
	○ MAX_RELAYS = 64, so index can be equal to 64 (at max value)
	○ The 1st right (lowest) bit of masks change.ackMask and change.rejMask is not used as index starts from 1	

g_changes member	● Effectively contains one value (or none), however is typed as `mapping(uint64 => Change)`. That however allows them to vote more consciously passing the change id to. ● All the expired or applied changes are forgotten	not an issue
updateRelays/ DuneUserSwap.updateAfterChange	● to update user swap contracts the updateRelays function must be called explicitly for each(!) user swap address, which allows to have a time gap between state of the root contract and user contracts or even remains some of the user contracts un-updated, which can make them unoperational as for example root contract changes the expiration time and stops serving ● the message to DuneUserSwap.updateAfterChange doesn't contain value, and the function itself is not in accept mode, so there is no explicitly mentioned gas payer	moderate
`DuneUserSwap`		
confirmOrder	● Some ambiguity in variables names: meaning of swap_date and g_swap_expiration_date member is not fully clear: ○ we found no reason to have g_swap_expiration_date more than g_merge_expiration_date root member as root contract will not serve us after ● uint64 bit = uint64(1) << index ; may cause integer overflow by the reasons mentioned above ● Double requirement of g_state == STATE_WAITING_FOR_CONFIRMATION on lines 235 and 275	critical

_requestCredit	• Checks the balance and therefore can not always normally completed, when is called from confirmOrder and not completed (because of balance shortage) contract will come to the deadlock:
	  ○ contract will be in the STATE_FULLY_CONFIRMED state, but the call of requestCredit can be performed only at STATE_CREDIT_DENIED state
	  ○ we cannot find a way to escape from this situation
		critical

revealOrderSecret	• Can be used for attack from the malicious relays to decrease the contract balance passing incorrect arguments and forcing the contracts to check them in accept mode • Moreover - after a series of unsuccessful revelation contract will close the way to do it, making it unusable - the attack direction for malicious party • has string return type however with status meaning - better to use enum or integer type here • function checks the balance to be greater than g_ton_amount + DEPOOL_FEE + STANDARD_MSG_FEE + ROOT_MSG_FEE, however it sends depool message or(xor) standard transfer and one message for root, so this check is redundant and requires more balance than actually needed	major
transferOrder	• if g_depool address corresponds to “bad” depool (wrong minStake, vesting is not confirmed and never will, closed) there is no way to change the mean of transfer - the user swap can only be closed after expiration • the same redundant check for balance as for revealOrderSecret	critical
`STATE*` `REVEAL_STATE*`	• Better to use enums	not an issue

constants		
cancelOrder closeSwap	• both functions do almost the same, however closeSwap can be called in more situations • however cancelOrder will not make the contract more operational as it doesn't clean it and it becomes unusable • closeSwap - when called on unconfirmed contract will not cause the decreasing of unconfirmed counter in root contract	moderate

updateAfterChange	• The comparison of merge_expiration_date with 0 is redundant as the contract is passed with g_merge_expiration_date which cannot be 0:
	  ○ it cannot be set to 0 in change:
	  ■ `if( change.merge_expiration_date != 0 ){ g_merge_expiration_date = change.merge_expiration_date; }`
	  ○ it cannot be 0 before change:
	  ■ proposeChange: `require( uint64(now) < g_merge_expiration_date, EXN_SWAP_EXPIRED );`
	• if the relay is removed from the relay set during the change, it cannot perform confirmOrder and initiate the swap, however that can be done by any other active relay even not revealing the secret - only its hash
	• if the g_nreqs member is decreased during the change and the current number of confirmations becomes enough for us, we can come to the deadlock if there are no any active relays which can confirm our swap, for example if all of them already have confirmed us:
	  ○ Say there was 100 relays and 50 for nreqs, we decrease the number of relays to 50 from the same set, and nreqs to 25. Let us have 49 confirmations from those 50 which already signed us. Now we have 49/25 confirmations which seems to be enough however we cannot do anything, as contract is in
		critical

	STATE_WAITING_FOR_CONFIRMATION state	
General suggestions	● There is no way to return own tons put to the contract, only passing it to root in closeSwap - however they can be potentially sent there to support contract operationality ● There is big power given for g_nreqs relays - they can manipulate date parameters and after that close the already credited contract ○ they can also exclude all other relays and distribute funds on their own will ● Many functions contain ‘require’ after tvm.accept which highly undesirable coding style allowing validator attack, almost all of them are fixed to the moment of contest end, so we omit them in the list	moderate/major

Conclusion

Pro:

1. Contracts are well structured
2. Functions are readable and short enough to realize
3. Variables' names are self-explanatory in most cases
4. Basic logic and contracts inter-communication with callbacks has been realized
5. Events are sent and satisfactory
6. Getters are complete to fulfil contract interaction
7. We suggest no need to redesign architecture

Cons:

1. Functions are to more carefully check the balance after tvm.accept and before transfers
2. Contracts access should be tuned to prevent eligible but malicious parties
3. Potentially uncaught exceptions are found
4. Some unnamed constants are found, which slightly prevents readability
5. DePool interaction is to be tuned
6. Voting mechanism for changes is imperfect and is suggested to be redesigned
7. Some issues are found: we treat functions with them as critical (5), major (4), moderate (2), and minor (2).

Pro-cons:

1. Some of the issues (but not all) are fixed at the moment of this document submission in the next commits

Finally, we propose that a workaround is needed and do not recommend the contracts to be used in production.