Realio Security audit

Author: Nguyen Duy Khanh, Nguyen Tien Dung, Ruslan Akhtariev

06 / 21 / 2023

contact@notional.ventures

notional.ventures

github.com/notional-labs

Table of Content

1. Introduction

Objectives of this audit

For this audit, we analyzed the codebase in these two repositories:

- https://github.com/realiotech/realio-network (commit: 96dc193b53cf0dfaeff60a1b71cd5e7b5a94cd53)
- https://github.com/realiotech/cosmos-sdk (commit: 863ad92d2e67f156ccddf02ac8e1a6884657d3f6)

The goal was to:

- Find security issues that could be exploited to steal funds or to seriously disrupt the services dependent on the codebase.
- Find any potential bugs that lead to unexpected behavior causing the system to function incorrectly.
- Find any violations against the codebase's spec.
- Find any violations against the common best practices usually followed by codebases related to the cosmos-sdk.
- Make recommendations to improve code readability and performance.

It is important to note that we can't cover all the possible issues that could be found within the codebase.

2. Guidance on reading this

This report classifies the issues found into the following severity categories:

=

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

3. Content

No	Description	Severity	Status
1	**MsgBeginRedelegate** can be exploited to change the token of a delegation	Critical	**Resolved**
2	**MsgCreateToken** allows minting token whose denom start with “a”	Critical	**Resolved**
3	**MsgTransferToken** can break staking invariant	Minor	**Resolved**
4	**ModuleAccountInvariants** only checks the sum of the denoms, not the equality of coins	Minor	**Resolved**
5	**InitGenesis** only checks the sum of the denoms, not the equality of coins	Minor	**Resolved**
6	Logic for bank **sendRestriction** is unnecessary complicated	Informational	**Pending**
7	Inefficient logic for checking address authorization	Informational	**Pending**
8	IBC transfer for authorization required token is blocked and must be enable manually	Informational	**Pending**

LIST OF ISSUES FOUND

Critical

1. MsgBeginRedelegate can be exploited to change the token of a delegation
The sdk fork is mainly about re-designing the staking module so that there could be two bonding denoms; each validator must choose only one of the two bonding denoms to operate on; a validator can only be delegated to using its correct bonding denom.

In the original sdk, the logic for msgServer.Delegate, msgServer.Undelegate and msgServer.BeginRedelegation all include one check to see if the token specified in the msg is indeed the bond denom. In the sdk fork, changes were made to those checks so that they instead check if the token specified in the msg equals the related validators' bonding denom. However, for the case of msgServer.BeginRedegation a vulnerability was discovered due to comparing only the dst validator's bonding denom instead of comparing both src validator's denom and dst validator's denom. As a result, an attacker could exploit this to change the token of his delegation by sending a MsgBeginRedelegate to redelegate to a different validator with a different bonding denom. The malicious MsgBeginRedelegate will be executed successfully if the specified unbonding token has the same denom as the destination validator's bonding denom because the logic for msgServer.BeginRedelegation, as mentioned above, only checks equality to the dst validator's denom.

2. MsgCreateToken allows minting token whose denom start with "a"
The token creation utility in x/asset allows users to create a token with their chosen denom that will be prefixed with "a" (as long as tokens of the same denom do not exist). There is a potential risk here where a user can mint any native token (token not created by x/asset) whose denom starts with "a". If the chain hosts native tokens whose denom starts with "a", a malicious actor could steal that token supply.

Minor

1. MsgTransferToken can break the staking invariant

The logic for executing MsgTransferToken doesn't block users from transferring tokens to module accounts. A malicious actor could exploit this to send tokens to BondedPool account or NotBondedPool account, staking invariants will be broken.

2. ModuleAccountInvariants only checks the sum of the denoms, not the equality of coins

When checking the bonded pool's tokens against bonded validators' tokens, the check only ensures that the total amount of bonded pool's tokens is equal to the total amounts of bonded validators' tokens. It does not take into account the equality of each of the specific coins held by the two accounts.

Ex: bonded pool has 100ario and 200arst, and bonded validators have (in sum) 200ario and 100arst. The sum of the bonded pool tokens is equal to bonded validators' tokens, but in reality, something went terribly wrong.

3. InitGenesis only checks the sum of the denoms, not the equality of coins

When checking bonded pool's tokens against bonded validators' tokens, the check only ensures that the total amount of bonded pool's tokens is equal to the total amounts of bonded validators' tokens. It does not take into account the equality of each of the specific coins held by the two accounts.

Informational

1. Logic for bank sendRestriction is unnecessary complicated

In bank.keeper, there's a field BaseSendKeeper.sendRestriction which contains a function for handling send restrictions. If we want to add any new logic for a restriction, we have to "combine" that logic into the function inside BaseSendKeeper.sendRestriction via SendRestriction.Append. The code for SendRestriction.Append is an unnecessary complication and could be optimized with a list of functions inside BaseSendKeeper.sendRestriction instead of just a function; removing the need to "combine" every time a new function is added as a Restriction.

2. Inefficient logic for checking address authorization

Authorization info contains information about whitelisted addresses of a token managed by asset module. We store authorization info in

the forms of an array of TokenAuthorization where TokenAuthorization is a go struct defined with 2 fields Address (string) and Authorized (bool). To check if an address is authorized to send/receive a token, we loop through this array of TokenAuthorization and check if there exists a TokenAuthorization with identical Address and Authorized = true. An optimization here could be to store a key for each whitelisted address in the kv store, that way, when checking address authorization, we only need to check if the key exists instead of looping through the whole lists of TokenAuthorization.

3. IBC transfer for authorization required token is blocked and must be enabled manually

The logic for ibc transfering involves sending tokens to an escrow address which is generated based on port id and channel id for each transfer channel. Therefore, the token manager must authorize this escrow address in order for the token to be transferable through the respective channel.

Here we have the security reports of the following modules:

1. Asset Module
2. Mint Module
3. Stake Module

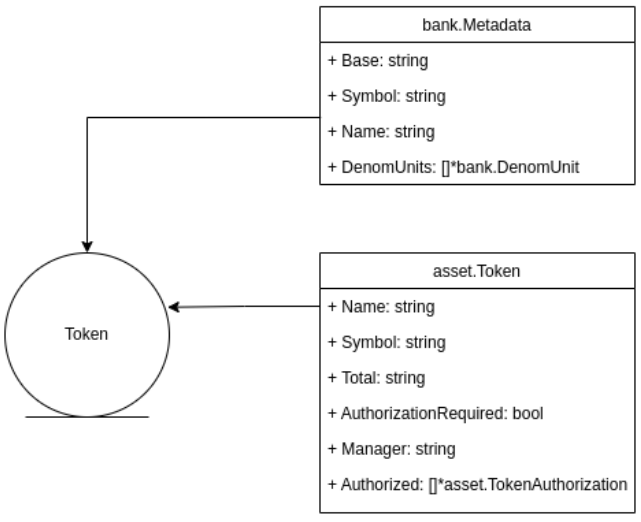
Asset Module

State

The data of x/asset is all about keeping track of its tokens which involves managing the state of the following objects:

1. Tokens created by the x/asset module, this data is stored in this key-value format: "Token/Value" | lowercase(Token.Symbol) -> TokenMetaData
2. Module x/bank's token metadata, this data is stored in this key-value format: 0x1 | byte(denom) -> ProtocolBuffer(Metadata)

A token created by x/asset is referenced by a bank.Metadata and an asset.Token. The bank.Metadata is mostly used together with asset.Token for SendRestriction logic while asset.Token is used in all logic involving handling the tokens. asset.Token contains info about the name, symbol, total amount and whitelisted addresses.



Relationship between Metadata and Token

- Metadata.Base = Metadata.Symbol + "a" = token denom
- Metadata.Symbol = Token.Symbol
- Metadata.Name = Metadata.Name

Since the token denom is always prefixed with “a”, there could be potential vulnerability in counterfeiting native tokens whose denom starts with “a”. It is up to the logic of creating/handling tokens to negate this vulnerability.

Logic

Keeper method

Core methods that interface directly with the state

- Setter:
  SetToken()
- Getter:
  GetToken()
  GetAllToken()
  IsAddressAuthorizedToSend()

Important notes :

1. GetToken(), SetToken() and IsAddressAuthorizedToSend() is frequently used by the module logic flow while GetAllToken() is only used for query service.
2. SetToken() lowercases the token symbol before setting it to the store. This means that GetToken() also has to lowercase token symbols before getting to store, which it did. The two methods have to be in accordance so that there’s no exploitation point regarding mismatched Get/Set.
3. SetToken() is the only core method that can modify the state of x/asset. It is also used only by MsgServer method which means that the only actor that can maliciously affect the module state is x/asset’s messages sent by an attacker. Therefore we will carefully inspect SetToken() together with its usage in MsgServer methods.

MsgServer methods

- CreateToken
  CreateToken takes in MsgCreateToken and creates a new token. MsgCreateToken contains Manager, Name, Symbol, Total and AuthorizationRequired. In the process, CreateToken instantiates an asset.Token and a bank.Metadata and sets them to store. It then mints the specified amount of the token whose denom is lowercase(Symbol) + "a" to the manager account. Since CreateToken has the capability to mint tokens on users’ request, it must check to see if the denom of minted tokens exists yet, or

else an attacker could use this to counterfeit a token that exists on chain.

- AuthorizeAddress
  AuthorizeAddress takes in MsgAuthorizeAddress and whitelists an address to be able to send the specified token. We should make sure that only the token manager can run this message. Other than that, there's no potential issue related to this method.
- UnAuthorizeAddress
  UnAuthorizeAddress takes in MsgUnAuthorizeAddress and removes an address out of the whitelist. We should make sure that only the token manager can run this message. Other than that, there are no potential issues related to this method.
- UpdateToken
  UpdateToken takes in MsgUpdateToken and enables whitelist functionality for the token. We should make sure that only the token manager can run this message. Other than that, there are no potential issues related to this method.

The 4 methods mentioned above all use SetToken(). There seems to be no security issue related to their usage of SetToken(). Therefore, it is safe to conclude that the module state will not be maliciously modified by any agents.

- TransferToken
  TransferToken takes in MsgTransferToken and sends tokens from an address to another, similar to bank.MsgSend but this method is specifically for sending x/asset tokens. In order for the whitelist functionality to work, we should make sure that we run through the SendRestriction logic and that SendRestriction works as expected. Furthermore we should make sure that TransferToken does include all the restrictions that bank.MsgSend imposes to users like not allowing sending tokens to module accounts.

SendRestriction

- AssetSendRestriction
  AssetSendRestriction handles the logic for restricting sending tokens to un-whitelisted addresses. Since this function affects both bank.SendCoins which is used in many core sdk logic and x/asset.TransferToken, we have inspected the code to make sure that:
  1. There's no way to bypass this restriction
  2. There's no way an attacker can exploit this restriction to block the sending of tokens they don't manage.

3. There's no way an attacker can exploit this to block the logic flow of other modules causing chain halt or DDOS.

Mint Module

Context

In Realio, the logic for inflation was changed so that the token minted token decreases over time instead of basing on a bonded ratio like the upstream inflation model.
In order to enforce the new inflation model, the Realio team has made changes to the mint module. Here we document the list of changes made to the mint module.

Modifications

- The amount of minted coins each block is calculated with this function:
  amount minted = (max supply - current supply) * inflation rate / blocks per year
  This function is calculated in this line in BeginBlocker. We've looked at the code and written tests to confirm that the code implements its function correctly.
- Adding maximum supply which doesn't exist upstream:

```
// Supply cap 1 RIO : 10^18 aRIO (attoRio)
var rioSupplyCap, _ =
math.NewIntFromString("750000000000000000000000000")
```

- Changes to module's param:
  Remove InflationRateChange, InflationMax, InflationMin and GoalBonded from module's params while adding InflationRate to module's params.
  Since we no longer use the upstream inflation model, which is based on incentivizing token holders to stake, there's no need to use some of the params of the upstream mint module.
  Adding InflationRate to the module's param enables directly adjusting our inflation via chain gov proposal.

Staking module

Context

The Realio Network at genesis uniquely supports two bonding denoms instead of just one like the upstream staking system. Each

validator will have to choose one bonding denom from the two bonding denoms and will operate solely on that chosen bonding denom (an user can only delegate to validators that has bonding denom equals to the token that user used to delegate).
Because of that, the Realio team chose to fork the sdk to make changes to the staking module. Here we (Notional-labs) document the list of changes and inspect them from a security perspective.

Changes from the origin sdk:

Data structs

- Validator:

Add denom field to Validator
type Validator struct {
	// operator_address defines the address of the validator's operator; bech encoded in JSON.
	OperatorAddress string `protobuf:"bytes,1,opt,name=operator_address,json=operatorAddress,proto3" json:"operator_address,omitempty"`
	// consensus_pubkey is the consensus public key of the validator, as a Protobuf Any.
	ConsensusPubkey *types1.Any `protobuf:"bytes,2,opt,name=consensus_pubkey,json=consensusPubkey,proto3" json:"consensus_pubkey,omitempty"`
	.....
	// bond_denom defines the bondable coin denomination.
	BondDenom string `protobuf:"bytes,12,opt,name=bond_denom,json=bondDenom,proto3" json:"bond_denom,omitempty"`
}

We should instead created a new entry in staking store to map Validator to it's bonding denom, eg: key(validator address | bond denom) -> value(bond denom) so that we avoid changing storage format from sdk's staking.

- UnbondingDelegationEntry:

Change InitialBalance and Balance type from Int to Coin.
This seems to be an unnecessary modification to UnbondingDelegationEntry since at where UnbondingDelegationEntry.InitialBalance and UnbondingDelegationEntry.Balance is used, we can always fetch

respective bond denom of the unbonding delegation. This modification does not introduce any security issue, however we shouldn't make this change to avoid storage breaking from sdk staking.

- RedelegationEntry:

Change InitialBalance type from Int to Coin.
This seems to be an unnecessary addition to RedelegationEntry since at where RedelegationEntry.InitialBalance is used, we can always fetch the respective bond denom of the redelegation. This modification does not introduce any security issue, however we shouldn't make this change to avoid storage breaking from sdk staking.

- Overall there seems to be no security issue regarding the changes to these structs.

Module logic

types:
Because of modified and additional field then all functions and objects related to Validator, UnbondingDelegationEntry, RedelegationEntry must be changed:
Modified methods:

- NewUnbondingDelegationEntry
- NewUnbondingDelegation
- AddEntry
- NewRedelegationEntry
- NewRedelegation
- NewRedelegationEntryResponse
- NewValidator

Added methods:

- ValidateBondDenom: validates two bond denoms instead one.

keeper
Since our staking system supports 2 bonding denoms, it's up to the keeper method to make sure there's no security issues related to mixing up the 2 bonding denoms.
Modified methods:

- InitGenesis: modify logic for checking bonded balance and unbonded balance, we sum up the tokens held in the bonded pool account and unbonded pool account. One way to optimize

this is to separate the amount of each of the two bonding tokens and validate the two amounts separately.

- CreateValidator: Add validator denom to MsgCreateValidator. Validator's denom. Used to prevent delegation of a different bond token to that validator.
- Delegate: add validation for checking if the delegation has the same token as the validator bond token.
- BeginRedelegate: add validation for checking token delegate's denom is equal validator's denom when create validator, delegate, undelegate, redelegate and cancel unbonding delegation are called. Prevent redelegate to validator which doesn't have same bond denom with source validator.
- Undelegate: add validation for checking if the undelegation has the same token as the validator bond token.
- CancelUnbondingDelegation: add validation for checking if the token specified in MsgCancelUnbondingDelegation has the same token as the validator bond token.
- SetUnbondingDelegationEntry, SetRedelegationEntry, Delegate, Unbond, Undelegate, BeginRedelegation, CompleteRedelegation: change type of some variables from math.Int to sdk.Coin.
- TotalBondedTokens/StakingTokenSupply: modify to sum up all balance of bond tokens in bonded pool/supply. Because these functions are used to calculate voting percentage/bonded ratio and we are treating the two denoms as equal in voting power, there's no need to differentiate the two denoms in the two functions.
- Slash: burn slashed token with related validator's bond denom.
- SlashRedelegation: burn slashed tokens with bond denom of its respective validator.
- SlashUnbondingDelegation: burn slashed tokens with bond denom of its respective validator
- ModuleAccountInvariants: Only validating the total amount of tokens in bonded pool account/unbonded pool account by summing up all tokens with multiple different denoms. Since we're validating if the tokens held in the bonded pool account/unbonded pool account are correct, we have to take into account for each of the tokens separately instead of summing them up.
- ApplyAndReturnValidatorSetUpdates: Add logic to move tokens between bonded pool account/unbonded pool account for each of the bonded tokens instead of just one bond token.

Added methods:

- BondDenomSlice: help convert chain's param BondDenom from string to array.
- IsBondDenomSupported: check if a denom is one of the two supported bond denoms.

CONCLUSION

We have successfully addressed and resolved several security issues discovered in the codebase. Additionally, we meticulously examined various other potential vulnerabilities that could pose a threat to the service relying on the codebase. Through this thorough code inspection process and extensive testing, we confirmed that the codebase is free from the identified and fixed security issues, as well as any other potential vulnerabilities.