



Security Audit Report

Stride - Autopilot & Claim modules

Authors: Darko Deuric, Marko Juric, Nikola Jovicevic

Last revised 28 July, 2023

Table of Contents

Audit Overview	1
The Project:	1
Scope of this audit:	1
Conducted work	1
Conclusions	2
Audit Dashboard	3
Target Summary	3
Engagement Summary	3
Severity Summary	3
System Overview.....	4
Autopilot module	4
Claim module	5
Threat Inspection.....	8
Unauthorized Creation of Pre-Reserved Custom Airdrop for Host Zone Using Chain ID and/or Airdrop ID	8
Unauthorized Creation of Bogus Claim Records and Potential Chain Performance Impact	9
Airdrop theft after fixing the problem reported by Jump Crypto	10
Findings	12
Leakage of stTokens is made possible through a specially crafted JSON message	14
Leakage of native tokens is made possible through a specially crafted JSON message	16
Attacker could block unbondings by sending insignificant amounts of stTokens.	18
Consider prefix store with individual keys for airdrops over single key for for the Params object	20
Consider Pagination in GetClaimRecords	22
Inefficient airdrop update with potential performance impact	25
Gas Optimization: avoid redundant gas consumption in GetClaimRecord function	27
Performance Optimization: Reserve space in advance for Go slices when you have a good estimate of the expected size and plan to extensively use append	29
Consider refactoring ClaimCoinsForAction function	32
Improve code organization and separation of responsibilities	35
Unnecessary Object Creation in Arithmetic Operations	37

Miscellaneous Code Improvements and Refinements	39
Appendix: Vulnerability Classification	41
Impact Score	41
Exploitability Score	41
Severity Score	42
Disclaimer	44

Audit Overview

The Project:

The audit focused on two modules: autopilot and claim.

The **autopilot** module simplifies user transactions by enabling them to perform multiple actions using a single IBC transfer with a formatted memo. In the previously audited version, users could transfer funds to the Stride chain and automatically liquid stake their tokens, resulting in stTokens being deposited into their Stride account. The current version expands the functionality to include transferring funds back to the sender's chain or forwarding them to another chain. It also introduces automatic redeem functionality through IBC transfer. Additionally, the autopilot module facilitates claim record updates for users from chains like Evmos, which use different coin types, enabling them to claim their stTokens.

On the other hand, the **claim** module focuses on Stride's airdrop mechanism. Airdrops are distributed based on specific criteria. Eligibility for an airdrop depends on whether a user was staking on a chain integrated with Stride at a certain block height. Unlike other chains that claw back unclaimed airdrop tokens after a set period, Stride's mechanism allows addresses to reset and claim again without losing eligibility. The claimable amount is calculated as a percentage of the pool, taking into account each address's weight.

It's important to note that airdrops have an end date, typically around three years. To claim the airdrops, users must take specific actions based on different categories defined by parameters. The percentage assigned to each category determines the claimable portion. For instance, the free portion (20%) requires no action and the coins are immediately accessible without vesting. However, for other portions, users need to stake and/or liquid stake their tokens, and the coins vest over a specified period, typically three months. It should be emphasized that deterministic linking of host and Stride addresses is only possible for chains with the same derivation path. For chains using different coin types, such as Stride and Evmos, users must link their Evmos address to a Stride address to claim their airdrops. This linking process is facilitated by the autopilot module.

Scope of this audit:

For the autopilot module, a critical area of concern was investigated to determine the possibility of any leakage of native or stTokens due to the usage of specially crafted JSON messages represented by the memo or receiver fields. It was crucial to thoroughly examine these fields and assess whether there were any vulnerabilities that could potentially result in the unauthorized transfer or loss of tokens.

Additionally, the audit paid close attention to the IBC functionality of the autopilot module. It was important to evaluate the security measures in place and determine if there were any potential attack vectors that could be exploited to directly target Stride's transfer module over a custom-created channel. Special attention was given to ensuring that the airdrop distribution pool for chains like Evmos was adequately protected against potential drain attacks.

Regarding the claim module, which handled Stride's airdrop implementation, the audit primarily focused on the accounting logic. It was essential to verify how the amount of tokens a user received upon claiming was determined. This involved examining the calculation methodology, considering different rounds and epochs, and understanding the precise rules governing token distribution during the audit duration.

Furthermore, the audit thoroughly assessed the authorization mechanisms and input validation procedures for the corresponding messages related to the claim module. It was crucial to ensure that these components were robust and effectively prevented unauthorized access or manipulation of the airdrop claiming process.

Conducted work

The audit was conducted by the following individuals:

- Darko Deuric
- Marko Juric

- Nikola Jovicevic

As part of the audit project, the team conducted the following work:

1. Review of Specification and Technical Diagrams:
 - The team conducted a comprehensive examination of all the provided materials pertaining to the autopilot and claim module, including the accompanying readme files, thoroughly analyzing their contents
2. Manual Code Review:
 - The team performed a manual inspection of the code base for both the autopilot and claim module. Our findings and comprehension of the code are extensively documented in the "System Overview" section

Conclusions

We were highly impressed with the overall quality of the codebase, which exhibited excellent structure and readability. The inclusion of both unit and integration tests in the test suite further enhanced our confidence in the reliability of the code. While the majority of our findings fell into the Medium or Informational severity categories, we did identify two High severity findings.

Your meticulous attention to detail and thoughtful code modifications truly stood out. We commend your exemplary efforts and eagerly anticipate future discussions and opportunities for collaboration.

Audit Dashboard

Target Summary

- **Type:** Specification and Implementation
- **Platform:** Golang
- **Artifacts:**
 - Autopilot: <https://github.com/Stride-Labs/stride/tree/main/x/autopilot> + <https://github.com/Stride-Labs/stride/pull/771> Repo
 - Claim: <https://github.com/Stride-Labs/stride/tree/main/x/claim>

Engagement Summary

- **Dates:** 13.06.2023 to 30.06.2023
- **Method:** Manual code review, protocol analysis
- **Employees Engaged:** 3

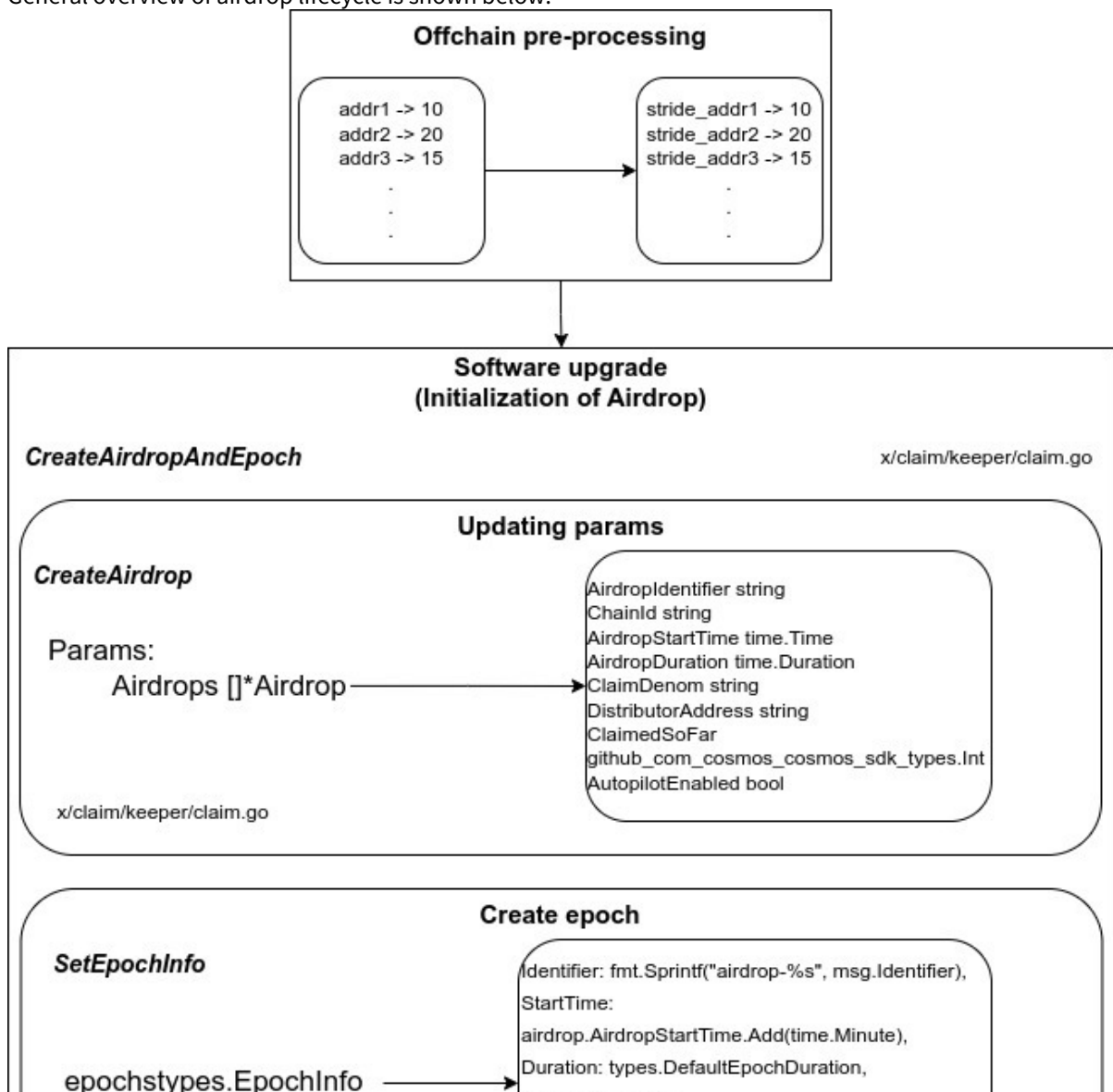
Severity Summary

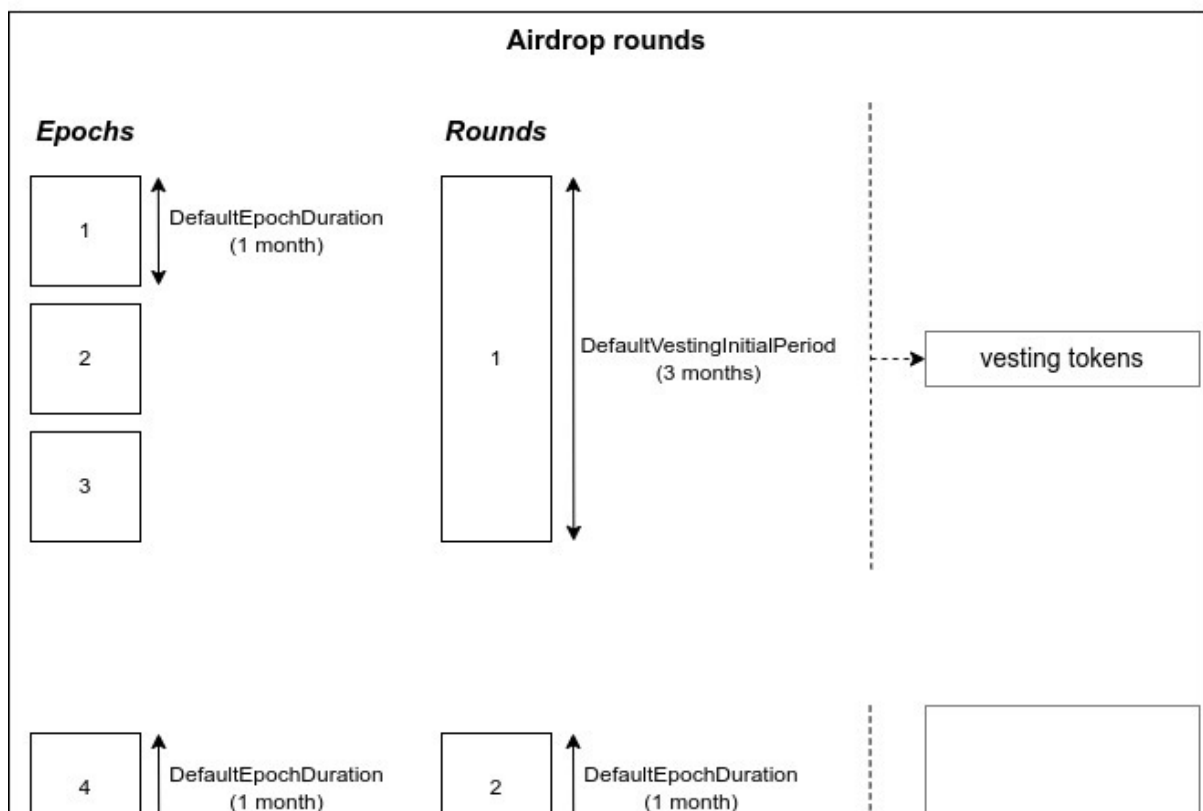
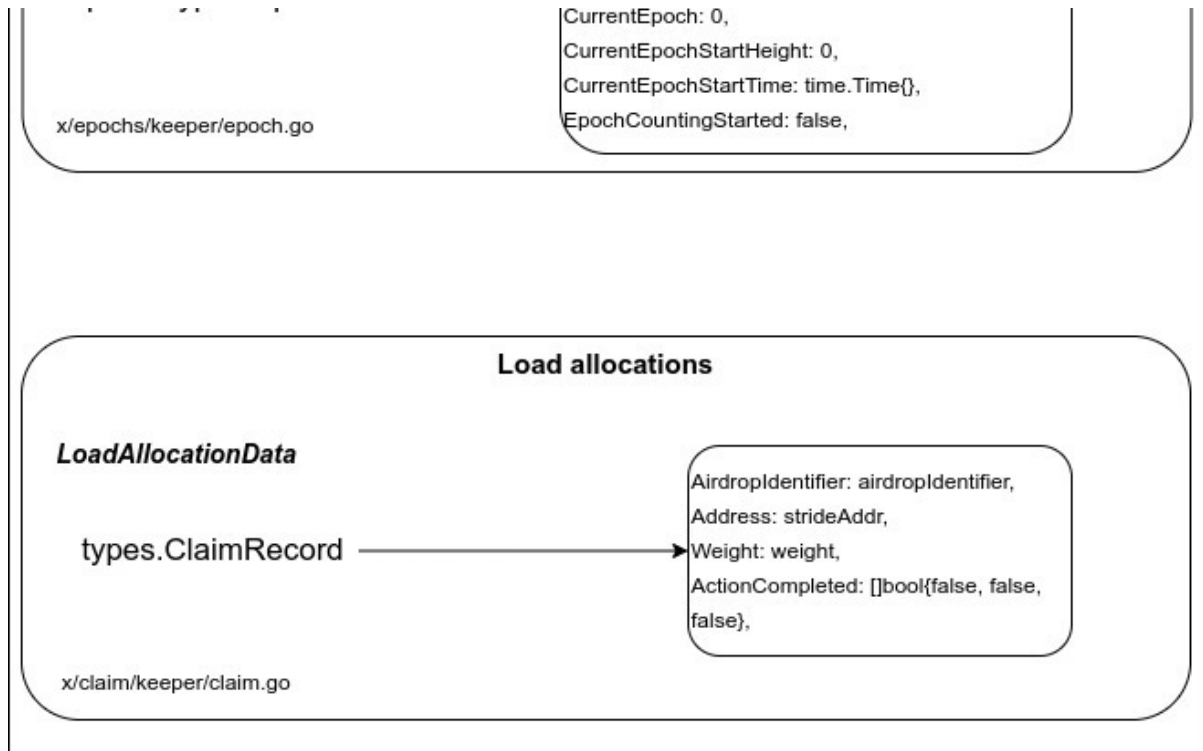
Finding Severity	#
Critical	0
High	2
Medium	1
Low	0
Informational	9
Total	12

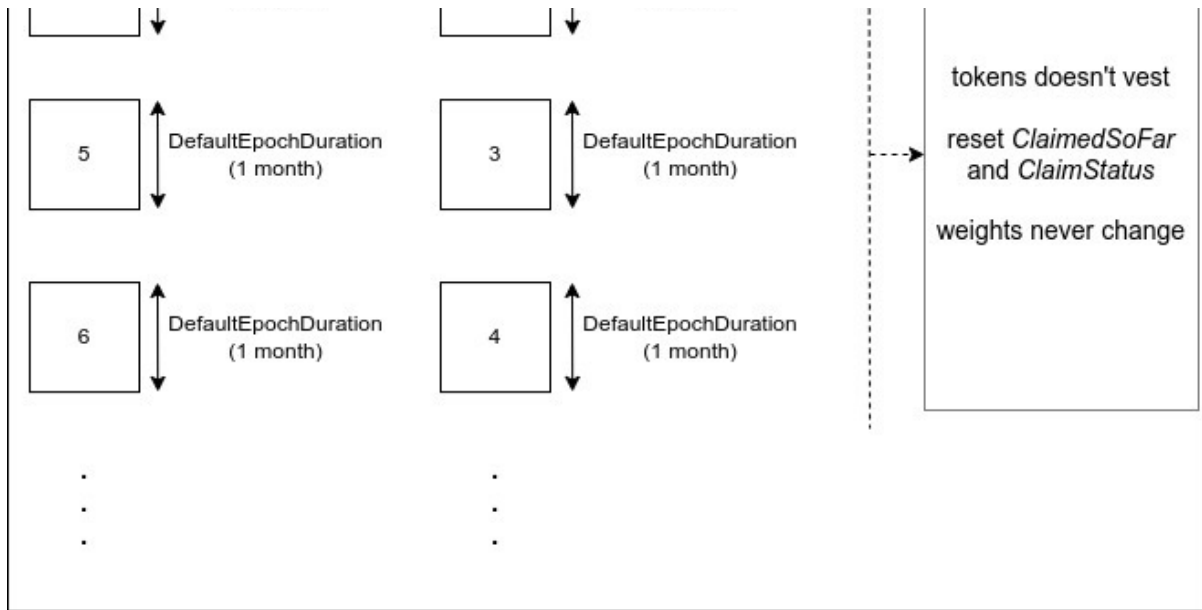
Claim module

Airdrop lifecycle:

- Off-chain preprocessing of Stride host chains is performed at a specific block height to determine addresses eligible for the airdrop and calculate weights.
- Host chain addresses are converted to Stride addresses.
- The initial airdrop is created with a software upgrade.
- When an airdrop is created, airdrop metadata and epoch are formed to facilitate rounds.
- Claim records are created during the process of airdrop allocations, where each airdrop/address pair receives a claim record.
- The airdrop process consists of epochs and rounds, with the first round having three epochs and subsequent rounds having a 1-1 ratio.
- During round 1, which has a duration of `DefaultVestingInitialPeriod` (typically 3 months), tokens vest over that period.
- The number of epochs in round 1 is calculated as $\text{DefaultVestingInitialPeriod} / \text{DefaultEpochDuration}$ (typically equal to 3).
- General overview of airdrop lifecycle is shown below:



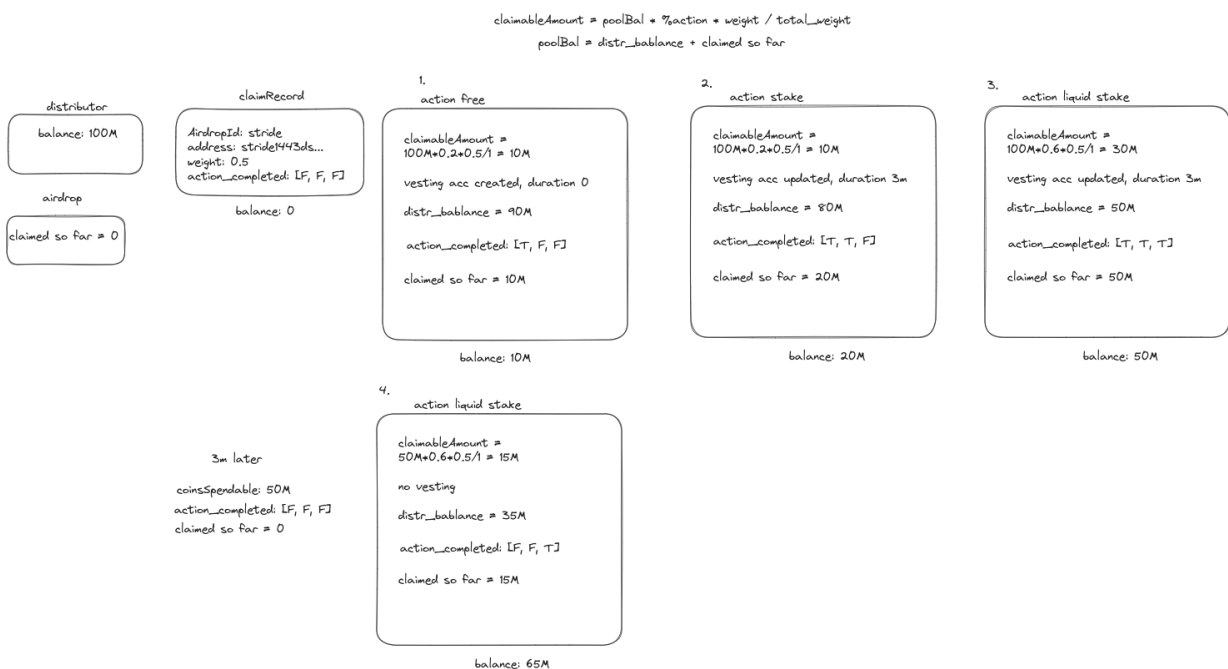




2 General overview

Claiming Actions:

- There are three types of actions that can be performed to claim tokens:
 - free portion (20% claimable),
 - after staking (additional 20% claimable), and
 - after liquid stake (remaining 60% claimable).
- The total weight represents the cumulative weight across all addresses for an airdrop.
- The claimable amount is calculated using the formula:
 - $\text{claimableAmount} = \text{poolBal} * \% \text{action} * \text{weight} / \text{total_weight}$.
- The pool balance (poolBal) is calculated as the sum of the distributor balance and the “claimed so far” amount.
- To prevent claims from decreasing over the course of a round, the claimed so far amount is tracked.
- An example of airdrop flow is shown below



3 An example of airdrop flow

Threat Inspection

The threats presented here are not necessarily a direct result of inspecting the code but rather a product of considering various attack vectors after reading the specification and examining the diagrams and protocol verification. All the threats were identified to be harmless (✓).

Unauthorized Creation of Pre-Reserved Custom Airdrop for Host Zone Using Chain ID and/or Airdrop ID ✓

The `CreateAirdrop` function allows any `MsgCreateAirdrop.Distributor` address to create an airdrop with specific names for `AirdropIdentifier` and `ChainId`. This creates a potential problem as users can "take a seat" in advance for upcoming airdrops by reserving the desired `AirdropIdentifier` and `ChainId`. Once an airdrop is defined with a particular `AirdropIdentifier` or `ChainId`, no additional airdrops can be added with the same values.

```
for _, airdrop := range params.Airdrops {
    if airdrop.AirdropIdentifier == msg.Identifier {
        return types.ErrAirdropAlreadyExists
    }
    if airdrop.ChainId == msg.ChainId {
        return types.ErrAirdropChainIdAlreadyExists
    }
}
```

Furthermore, only the `Distributor` address associated with the airdrop has the authority to invoke the `DeleteAirdrop` function, limiting control over the airdrop's lifecycle.

```
if !addr.Equals(distributor) {
    return nil, errorsmod.Wrapf(sdkerrors.ErrInvalidAddress, "invalid distributor address")
}
```

Additionally, the airdrop can be created with a custom duration, potentially leading to long-lasting airdrops.

```
AirdropDuration: time.Duration(msg.Duration * uint64(time.Second)),
```

This scenario should be considered when the naming of airdrops is relevant, as it may impact the availability and integrity of future airdrops.

The mentioned **threat is not applicable** in this scenario since users do not have the capability to explicitly invoke the `CreateAirdrop` function. Although the claim module encompasses various message types, it does not include the `CreateAirdrop` message inside the handler:

```
func NewHandler(k keeper.Keeper) sdk.Handler {
    msgServer := keeper.NewMsgServerImpl(k)
    return func(ctx sdk.Context, msg sdk.Msg) (*sdk.Result, error) {
        ctx = ctx.WithEventManager(sdk.NewEventManager())

        switch msg := msg.(type) {
```

```

    case *types.MsgSetAirdropAllocations:
        res, err := msgServer.SetAirdropAllocations(sdk.WrapSDKContext(ctx), msg)
        return sdk.WrapServiceResult(ctx, res, err)
    case *types.MsgClaimFreeAmount:
        res, err := msgServer.ClaimFreeAmount(sdk.WrapSDKContext(ctx), msg)
        return sdk.WrapServiceResult(ctx, res, err)
    case *types.MsgDeleteAirdrop:
        res, err := msgServer.DeleteAirdrop(sdk.WrapSDKContext(ctx), msg)
        return sdk.WrapServiceResult(ctx, res, err)
    default:
        errMsg := fmt.Sprintf("unrecognized %s message type: %T",
types.ModuleName, msg)
        return nil, errorsmod.Wrap(sdkerrors.ErrUnknownRequest, errMsg)
    }
}

```

Unauthorized Creation of Bogus Claim Records and Potential Chain Performance Impact

The `SetAirdropAllocations` function allows any user to create claim records based on the provided `Allocator`. However, the `Allocator` must match the `airdropDistributor`, which could be controlled by a malicious user who previously created a fake airdrop.

```

addr, err := sdk.AccAddressFromBech32(msg.Allocator)
if err != nil {
    return nil, err
}

if !addr.Equals(airdropDistributor) {
    return nil, errorsmod.Wrapf(sdkerrors.ErrInvalidAddress, "invalid distributor
address")
}

```

This means that an attacker can generate a large number of bogus claim records by manipulating the input `msg.Users`. While the attacker may incur transaction fees for setting these records (`server.keeper.SetClaimRecords(ctx, records)`), the real concern lies in the potential impact on chain performance.

Within the `EndBlocker` function, there is a for loop that iterates over all airdrops, including the potentially fake ones, to remove expired airdrops.

```

func (k Keeper) EndBlocker(ctx sdk.Context) {
    // Check airdrop elapsed time every 1000 blocks
    if ctx.BlockHeight()%1000 == 0 {
        params, err := k.GetParams(ctx)
        if err != nil {
            panic(err)
        }

        // End Airdrop
    }
}

```

```

    for _, airdrop := range params.Airdrops {
        goneTime := ctx.BlockTime().Sub(airdrop.AirdropStartTime)
        if goneTime > airdrop.AirdropDuration {
            // airdrop time has passed
            err := k.EndAirdrop(ctx, airdrop.AirdropIdentifier)
            if err != nil {
                panic(err)
            }
        }
    }
}

```

If there are numerous fake airdrops filled with millions of records, the deletion process could significantly impact the chain's performance or even lead to chain halting.

```

func (k Keeper) EndAirdrop(ctx sdk.Context, airdropIdentifier string) error {
    ctx.Logger().Info("Clearing claims module state entries")
    k.clearInitialClaimables(ctx, airdropIdentifier)
    k.DeleteTotalWeight(ctx, airdropIdentifier)
    return k.DeleteAirdropAndEpoch(ctx, airdropIdentifier)
}

```

The mentioned **threat is not applicable** in this scenario due to the following reasons:

1. Users can indeed call the `SetAirdropAllocations` function, but the `Allocator` will not match the `airdropDistributor`. This means that even if users attempt to manipulate the airdrop allocations, they will not be able to create an airdrop from the outside world.
2. It is important to note that the creation of an airdrop is not possible for anyone outside the system. Therefore, the risk associated with unauthorized creation of an airdrop is mitigated.

Airdrop theft after fixing [the problem](#) reported by Jump Crypto

The vulnerability in the integration of IBC within Stride allowed for potential airdrop theft. The vulnerability could have enabled an attacker to steal all unclaimed airdrops on the Stride chain. At the time of discovery, approximately 1.6 million STRD tokens (equivalent to around \$4 million) were at risk. The issue has been fixed after being reported to the Stride contributors, ensuring that no malicious exploitation occurred.

The vulnerability specifically affected the mechanism of updating the destination address of an unclaimed ClaimRecord through a cross-chain IBC message. The x/autopilot module intercepted incoming IBC transfers and extracted Stride-specific instructions from the metadata fields. However, the update mechanism lacked sufficient validation to ensure that the airdrop update was initiated by the actual recipient. This vulnerability exposed the airdrop update mechanism to potential manipulation by unauthorized actors.

The mentioned **threat is no longer applicable** due to recent changes. To exploit the vulnerability, an attacker would now need to have a registered host zone on the Stride side, as outlined below:

```

func (k Keeper) TryUpdateAirdropClaim {
    ...
    hostZone, found := k.stakeibcKeeper.GetHostZoneFromTransferChannelID(ctx,
        packet.GetDestChannel())
    if !found {
        return errorsmod.Wrapf(stakeibctypes.ErrHostZoneNotFound,
            "host zone not found for transfer channel %s", packet.GetDestChannel())
    }
}

```

```
...  
}
```

The registration of new host zones is tightly controlled by Stride administrators.

Findings

Title	Type	Severity	Impact	Exploitability	Issue
Leakage of stTokens is made possible through a specially crafted JSON message	IMPLEMENTATION	3 HIGH	3 HIGH	2 MEDIUM	
Leakage of native tokens is made possible through a specially crafted JSON message	IMPLEMENTATION	3 HIGH	3 HIGH	2 MEDIUM	
Attacker could block unbondings by sending insignificant amounts of stTokens.	IMPLEMENTATION	2 MEDIUM	2 MEDIUM	2 MEDIUM	
Consider prefix store with individual keys for airdrops over single key for the Params object	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	
Consider refactoring ClaimCoinsForAction function	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	
Inefficient airdrop update with potential performance impact	IMPLEMENTATION	0 INFORMATIONAL	1 LOW	0 NONE	
Gas Optimization: avoid redundant gas consumption in GetClaimRecord function	IMPLEMENTATION	0 INFORMATIONAL	1 LOW	0 NONE	
Performance Optimization: Reserve space in advance for Go slices when you have a good estimate of the expected size and plan to extensively use append	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	
Consider Pagination in GetClaimRecords	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	

Title	Type	Severity	Impact	Exploitability	Issue
Improve code organization and separation of responsibilities	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	
Unnecessary Object Creation in Arithmetic Operations	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	
Miscellaneous Code Improvements and Refinements	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE	

Leakage of stTokens is made possible through a specially crafted JSON message

Title	Leakage of stTokens is made possible through a specially crafted JSON message
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Issue	

Involved artifacts

- x/autopilot/keeper/airdrop.go
- x/autopilot/module_ibc.go

Description

In this scenario, the attacker exploits the IBC transfer mechanism by sending a specially crafted transaction with specific metadata fields. By manipulating the `Memo` or `Receiver` field, the attacker tricks the victim into performing actions on their behalf. The attacker successfully initiates a liquid stake operation using the victim's tokens and receives stTokens in return. Furthermore, the attacker gains control over the victim's IBC transfers, determining the destination address through the manipulated metadata. This scenario highlights the potential dangers of unauthorized actions and the importance of validating and securing metadata in IBC transfers.

Problem Scenarios

1. The attacker initiates an IBC transfer, sending, for example, 2 Atom, along with a specially crafted `Memo` or `Receiver` field. This field will be parsed to extract the following information:

- a.

```
StakeIBCPacketMetadata.Action = LiquidStake
StakeIBCPacketMetadata.StrideAddress = victim's stride addr
StakeIBCPacketMetadata.IBCReceiver = attacker's cosmos addr

PacketForwardMetadata.Receiver = attacker's stride addr
```
- b. The attacker receives 2 ibc/Atom at the address defined in `PacketForwardMetadata.Receiver` within the Stride.

2. The functions `TryLiquidStaking` and `RunLiquidStake` are invoked:

- a. `MsgLiquidStake.Creator` = victim's stride addr
`MsgLiquidStake.Amount` = the amount that the attacker sent
- b. The victim, assuming they possess 2 or more `ibc/Atom` for Liquid Staking, proceeds to liquid stake 2 `ibc/Atom` and receives 2 `stAtom`.
- c. Note: In this step, the attacker has effectively liquid staked on behalf of the victim, given that there was a sufficient token amount to stake.

3. Eventually, the function `IBCTransferStAsset` is called:

- a. `MsgTransfer.Sender` = victim's stride addr
`MsgTransfer.Receiver` = attacker's cosmos addr
- b. Note: What makes this entire message chain particularly dangerous is its conclusion, where the victim might initiate an IBC transfer of `stTokens` to an address of the attacker's choice. This address is determined through the `StakeibcPacketMetadata.IbcReceiver` field.

Recommendation

To avoid this scenario, it is necessary to enforce the condition that:

```
StakeibcPacketMetadata.StrideAddress == PacketForwardMetadata.Receiver
```

Leakage of native tokens is made possible through a specially crafted JSON message

Title	Leakage of native tokens is made possible through a specially crafted JSON message
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Issue	

Involved artifacts

- x/autopilot/keeper/airdrop.go
- x/autopilot/module_ibc.go

Description

In this scenario, the attacker exploits an IBC transfer by manipulating the `Memo` or `Receiver` field and initiating a transfer of some amount of `ibc/stuatom`s from `CosmosHub`. By crafting the metadata fields, the attacker deceives the victim into triggering a `MsgRedeemStake` operation on their behalf. The attacker receives the transferred tokens within their Stride address. They then utilize the victim's Stride address as the creator of the `MsgRedeemStake`, with the victim's tokens as the stake amount. The concerning part is that the attacker can control the receiver address through the manipulated metadata, potentially directing the tokens to their own address. This allows the attacker to effectively steal the victim's native tokens. It underscores the importance of validating and securing metadata in IBC transfers to prevent unauthorized actions and protect users' assets.

Problem Scenarios

1. The attacker initiates an IBC transfer, sending, for example, 10 `ibc/stuatom` from `CosmosHub`, along with a specially crafted `Memo` or `Receiver` field. This field will be parsed to extract the following information:

- a.

```
StakeibcPacketMetadata.Action = RedeemStake
StakeibcPacketMetadata.StrideAddress = victim's stride addr
StakeibcPacketMetadata.IbcReceiver = attacker's cosmos addr

PacketForwardMetadata.Receiver = attacker's stride addr
```

2. The attacker receives 10 stuatom at the address defined in `PacketForwardMetadata.Receiver` within the Stride.
3. The functions `TryRedeemStake` and `RunRedeemStake` are invoked:

- a.

```
MsgRedeemStake.Creator = victim's stride addr
MsgRedeemStake.Amount = the amount that the attacker sent
MsgRedeemStake.Receiver = attacker's cosmos addr
```

4. It appears that the attacker could send 10 ibc/stuatom to their own address on Stride. They can then initiate a `MsgRedeemStake`, where the message creator is the victim's Stride address. Assuming the victim has 10 ibc/stuatom or more, this action will cause the `MsgRedeemStake` to successfully complete. The most dangerous aspect of this scenario is that the `MsgRedeemStake.Receiver` is not under the control of the `MsgRedeemStake.Creator`, but rather determined by whatever the attacker sets in the `Memo` or `Receiver` field. This means that the attacker can set it to an address they control, ultimately allowing them to steal the native tokens from the victim.

Recommendation

To avoid this scenario, it is necessary to enforce the condition that:

```
StakeIbcPacketMetadata.StrideAddress == PacketForwardMetadata.Receiver
```

Attacker could block unbondings by sending insignificant amounts of stTokens.

Title	Attacker could block unbondings by sending insignificant amounts of stTokens.
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Issue	

Involved artifacts

- x/autopilot/keeper/redeemstake.go
- x/autopilot/module_ibc.go

Description

In this scenario, an attacker initiates an IBC transfer of let's say 10 ibc/stuatom from CosmosHub, accompanied by a carefully crafted `Memo` or `Receiver` field. This metadata is parsed to extract specific information, including the victim's stride address and the attacker's cosmos address. Additionally, the `PacketForwardMetadata.Receiver` is set to the victim's stride address.

As a result, the victim receives 10 stuatom at their designated Stride address. Subsequently, the functions `TryRedeemStake` and `RunRedeemStake` are triggered. The `MsgRedeemStake` is created with the victim's stride address as the creator, the amount sent by the attacker, and the attacker's cosmos address as the receiver.

The redemption process involves retrieving the redemption ID based on the creator of the `MsgRedeemStake`.

The `UserRedemptionRecord` is then checked to determine if the user has already redeemed their stake within the current epoch (24 hours). If the redemption has already taken place, the user will be unable to redeem again until the next 24-hour period.

It's important to be aware that a malicious user can disrupt the unbonding process for other users by sending small amounts of stTokens, causing delays or obstructions. This highlights the need for measures to mitigate such attacks and ensure a smooth redemption process for all users.

Problem Scenarios

1. The attacker initiates an IBC transfer, sending, for example, 10 ibc/stuatom from CosmosHub, along with a specially crafted `Memo` or `Receiver` field. This field will be parsed to extract the following information:

- a.


```

StakeIBCPacketMetadata.Action = RedeemStake
StakeIBCPacketMetadata.StrideAddress = victim's stride addr
StakeIBCPacketMetadata.IBCReceiver = attacker's cosmos addr

PacketForwardMetadata.Receiver = victim's stride addr
          
```
 - b. The victim receives 10 stautom at the address defined in `PacketForwardMetadata.Receiver` within the Stride.
 2. The functions `TryRedeemStake` and `RunRedeemStake` are invoked:
 - a.


```

MsgRedeemStake.Creator = victim's stride addr
MsgRedeemStake.Amount = the amount that the attacker sent
MsgRedeemStake.Receiver = attacker's cosmos addr
              
```
 - b. The `redemptionId` is retrieved based on `MsgRedeemStake.Creator`. Subsequently, the `UserRedemptionRecord` is checked to verify if the user has already redeemed their stake in the current epoch (24 hours). If the redemption has already occurred, the user will be unable to redeem for the next 24 hours.
 3. Even if there is enforced condition: `StakeIBCPacketMetadata.StrideAddress == PacketForwardMetadata.Receiver`, it doesn't protect against the following step:
 4. The malicious user has the ability to disrupt the unbonding process for other users by sending small, insignificant amounts of stTokens. While they cannot directly drain the victims' native assets as demonstrated in [this scenario](#), their actions can compromise the overall unbonding process because of constraints shown in 2b.

Consider prefix store with individual keys for airdrops over single key for the Params object

Title	Consider prefix store with individual keys for airdrops over single key for the Params object
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/types/keys.go
- x/claim/keeper/claim.go

Description

The `GetAirdropByIdentifier` function retrieves the desired airdrop object by utilizing storage to access the `Params` object and searching for a matching airdrop identifier within the `Airdrops` slice.

Problem Scenarios

1. **Limited Flexibility:** The use of a single key for the `Params` object can limit the flexibility of managing individual airdrops within the claim module. With a single key approach, it becomes challenging to perform specific operations on individual airdrops, such as retrieving, updating, or deleting a particular airdrop. This lack of flexibility can hinder the ability to efficiently manage airdrop data.
2. **Inefficient Retrieval:** When using a single key for the `Params` object, retrieving specific airdrops based on their identifiers can be inefficient. It requires iterating over the entire `Params` object, which may contain a large number of airdrops, just to find the desired airdrop. This process can be time-consuming and resource-intensive, especially as the number of airdrops increases.
3. **Limited Organization:** Storing all airdrop data under a single key may result in limited organization and clarity within the storage. As the number of airdrops grows, it becomes challenging to differentiate and manage individual airdrop objects effectively. This lack of organization can make it harder to maintain and understand the airdrop data structure.

Recommendation

If the `GetAirdropByIdentifier` method is **invoked frequently** and **performance is a crucial factor**, using a **prefix store** with individual keys for each `Airdrop` would generally be more efficient. This approach allows for direct retrieval of the desired `Airdrop` object by its identifier, without the need to iterate over a list. It provides faster access and can significantly improve the response time when the method is called frequently. Here is how the approach with using a prefix store for data storage could be implemented:

```
const (
    // ...
    // AirdropsStorePrefix defines the store prefix for the airdrops
    AirdropsStorePrefix = "airdrops"

    // ...
)

// Storing an airdrop
airdrop := &types.Airdrop{
    AirdropIdentifier: "Osmosis",
    // ...
}

// Create the key using the appropriate prefix
key := []byte(types.AirdropsStorePrefix + airdrop.AirdropIdentifier)

// Marshal the airdrop object into bytes
airdropBytes := k.cdc.MustMarshalBinaryBare(airdrop)

// Store the airdrop in the prefix store
store.Set(key, airdropBytes)
```

On the other hand, if the `GetAirdropByIdentifier` method is **rarely invoked** or **the number of `Airdrop` objects is relatively small**, the performance difference between using a prefix store with individual keys or a single key for the `Params` object might not be significant. In such cases, simplicity and code readability can be prioritized over the minor performance gain.

Consider Pagination in GetClaimRecords

Title	Consider Pagination in GetClaimRecords
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- `x/claim/keeper/claim.go`

Description

The provided code shows a function called `GetClaimRecords` that retrieves all claim records for a specific airdrop identifier. However, the code does not implement pagination, which can lead to potential problems when dealing with a large number of claim records.

```
func (k Keeper) GetClaimRecords(ctx sdk.Context, airdropIdentifier string)
[]types.ClaimRecord {
    store := ctx.KVStore(k.storeKey)
    prefixStore := prefix.NewStore(store, append([]byte(types.ClaimRecordsStorePrefix), []byte(airdropIdentifier)...))

    iterator := prefixStore.Iterator(nil, nil)
    defer iterator.Close()

    claimRecords := []types.ClaimRecord{}
    for ; iterator.Valid(); iterator.Next() {

        claimRecord := types.ClaimRecord{}

        err := proto.Unmarshal(iterator.Value(), &claimRecord)
        if err != nil {
            panic(err)
        }

        claimRecords = append(claimRecords, claimRecord)
    }
    return claimRecords
}
```

```
}

```

Problem Scenarios

1. **Fetching All Records:** The code retrieves all claim records for the specified airdrop identifier without any pagination mechanism. This means that if the number of claim records is substantial, fetching and processing all records at once can result in performance issues, excessive memory usage, and potential timeouts.
2. **Memory Consumption:** Loading and storing all claim records in memory can consume a significant amount of resources, especially when the number of records is large. This can lead to increased memory usage, slower response times, and potential out-of-memory errors.
3. **Performance Impact:** Processing and iterating through a large number of claim records in a single operation can introduce performance bottlenecks. It can cause delays in retrieving the records and impact the overall responsiveness of the system.

Recommendation

If the number of claim records is expected to be large, it might be necessary to **implement pagination to fetch records in smaller batches**. This approach can help manage memory consumption and improve performance when dealing with a significant number of claim records.

```
// GetClaimRecordsWithPagination fetches claim records in smaller batches using
// pagination.
func (k Keeper) GetClaimRecordsWithPagination(ctx sdk.Context, airdropIdentifier
string, page, limit int) ([]types.ClaimRecord, int) {
    store := ctx.KVStore(k.storeKey)
    prefixStore := prefix.NewStore(store, append([]byte(types.ClaimRecordsStorePrefix),
[]byte(airdropIdentifier)...))
    iterator := prefixStore.Iterator(nil, nil)
    defer iterator.Close()

    totalRecords := 0
    startIndex := (page - 1) * limit
    endIndex := page * limit

    claimRecords := []types.ClaimRecord{}
    for i := 0; iterator.Valid() && i < endIndex; iterator.Next() {
        // Skip records until the start index is reached
        if i < startIndex {
            i++
            continue
        }
        claimRecord := types.ClaimRecord{}
        err := proto.Unmarshal(iterator.Value(), &claimRecord)
        if err != nil {
            panic(err)
        }

        claimRecords = append(claimRecords, claimRecord)
        i++
        totalRecords++
    }
}
```

```
    return claimRecords, totalRecords  
}
```

Inefficient airdrop update with potential performance impact

Title	Inefficient airdrop update with potential performance impact
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

The code in functions such as `IncrementClaimedSoFar`, `ResetClaimedSoFar`, and `MigrateClaimDistributorAddress` demonstrates an inefficient method for updating an airdrop. This approach has the potential to negatively impact performance, especially when dealing with a large number of airdrops. Here is code example for `IncrementClaimedSoFar`:

```
func (k Keeper) IncrementClaimedSoFar(ctx sdk.Context, identifier string, amount
sdkmath.Int) error {
    params, err := k.GetParams(ctx)
    if err != nil {
        panic(err)
    }

    if amount.LT(sdkmath.ZeroInt()) {
        return types.ErrInvalidAmount
    }

    newAirdrops := []*types.Airdrop{}
    for _, airdrop := range params.Airdrops {
        if airdrop.AirdropIdentifier == identifier {
            airdrop.ClaimedSoFar = airdrop.ClaimedSoFar.Add(amount)
        }
        newAirdrops = append(newAirdrops, airdrop)
    }
    params.Airdrops = newAirdrops
    return k.SetParams(ctx, params)
```

```
}

```

Problem Scenarios

1. **Inefficient Airdrop Update:** The code iterates through each airdrop in the `params.Airdrops` slice and checks if the `AirdropIdentifier` matches the given `identifier`. If there is a match, it updates the `ClaimedSoFar` field of the airdrop by adding the specified `amount`. However, instead of directly updating the matched airdrop in-place, the code creates a new slice `newAirdrops` and appends each airdrop, including the updated one, to this new slice. This results in unnecessary memory allocation and copying for each airdrop, regardless of whether it requires an update or not.
2. **Performance Impact:** As the number of airdrops in `params.Airdrops` increases, this inefficient update approach becomes more resource-intensive. Each iteration involves creating a new slice and copying the airdrops, leading to unnecessary memory allocation and increased processing time. This can result in performance degradation, especially in scenarios where there are a significant number of airdrops or frequent updates to the airdrop information.

Recommendation

1. Instead of creating a new slice and replacing the entire `params.Airdrops`, you can directly modify the `ClaimedSoFar` field of the specific `Airdrop` within the existing slice. Here's an example:

```
2. for _, airdrop := range params.Airdrops {
    if airdrop.AirdropIdentifier == identifier {
        airdrop.ClaimedSoFar = airdrop.ClaimedSoFar.Add(amount)
        break
    }
}
```

3. By avoiding the creation of a new slice and appending elements, this approach reduces memory usage and improves efficiency when updating the `ClaimedSoFar` field of the specific `Airdrop`.

Gas Optimization: avoid redundant gas consumption in GetClaimRecord function

Title	Gas Optimization: avoid redundant gas consumption in GetClaimRecord function
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

The provided code includes an inefficient approach to check for the existence of a claim record and retrieve it for a specific address. This approach utilizes both the `Has` and `Get` functions on the prefix store, leading to unnecessary gas consumption and decreased efficiency.

```
func (k Keeper) GetClaimRecord(ctx sdk.Context, addr sdk.AccAddress,
airdropIdentifier string) (types.ClaimRecord, error) {
    store := ctx.KVStore(k.storeKey)
    prefixStore := prefix.NewStore(store, append([]byte(types.ClaimRecordsStorePrefix), []byte(airdropIdentifier)...))
    if !prefixStore.Has(addr) {
        return types.ClaimRecord{}, nil
    }
    bz := prefixStore.Get(addr)
    ...
}

func KVGasConfig() GasConfig {
    return GasConfig{
        HasCost:      1000,
        DeleteCost:   1000,
        ReadCostFlat: 1000,
        ReadCostPerByte: 3,
        WriteCostFlat: 2000,
```

```
        WriteCostPerByte: 30,  
        IterNextCostFlat: 30,  
    }  
}
```

Problem Scenarios

1. Redundant Use of `Has` Function: The code first uses the `Has` function on the `prefixStore` to check if a claim record exists for the given `addr`. However, immediately after the `Has` check, the code proceeds to use the `Get` function on the same `prefixStore` to retrieve the claim record. This redundant use of the `Has` function before calling `Get` results in unnecessary gas consumption and additional processing overhead.
2. Wasted Gas Consumption: The `Has` function, in addition to performing the existence check, also consumes gas (1000 per item). By using both the `Has` and `Get` functions, the code incurs unnecessary double gas costs for executing the `Has` check, even though the `Get` function will retrieve the claim record regardless. This leads to wasted gas consumption, especially when repeated for multiple address lookups or in situations with a large number of claim records.

Recommendation

In the context of the `GetClaimRecord` function, you can use `prefixStore.Get` to retrieve the value and check if it's an empty byte slice to determine if the claim record exists or not. **You can also avoid the additional gas consumption** that would occur when calling `Has()`, regardless of whether the key exists or not. By directly checking the length of the byte slice, you can determine the existence of the claim record without incurring that extra gas cost.

Performance Optimization: Reserve space in advance for Go slices when you have a good estimate of the expected size and plan to extensively use append

Title	Performance Optimization: Reserve space in advance for Go slices when you have a good estimate of the expected size and plan to extensively use append
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

The provided code shows a function called `LoadAllocationData` within the keeper that processes allocation data and populates a slice of `ClaimRecord` structs. However, the code could benefit from performance optimization by reserving space in advance for the Go slice when there is a good estimate of the expected size. This optimization can be particularly useful when the number of lines in the `allocationData` is known in advance and is likely to match the size of the `records` slice.

```
func (k Keeper) LoadAllocationData(ctx sdk.Context, allocationData string) bool {
    records := []types.ClaimRecord{}
    ...

    for _, line := range lines {
        data := strings.Split(line, ",")
        airdropIdentifier := data[0]
        sourceChainAddr := data[1]
        airdropWeight := data[2]
        strideAddr := utils.ConvertAddressToStrideAddress(sourceChainAddr)
        ...
        weight, err := sdk.NewDecFromStr(weightStr)
        ...
    }
}
```



```

    records = append(records, types.ClaimRecord{
        AirdropIdentifier: airdropIdentifier,
        Address:          strideAddr,
        Weight:           weight,
        ActionCompleted: []bool{false, false, false},
    })

    ...
}
...
}

```

Problem Scenarios

Slice Resizing Overhead: The code initializes an empty slice called `records` to store the `ClaimRecord` structs. It then iterates over each line in the `allocationData` and populates the `records` slice using the `append` function. However, the slice grows dynamically as new elements are added, which involves resizing the underlying array, potentially resulting in frequent reallocations and memory copying. This resizing overhead can lead to decreased performance, especially when there is a large number of records.

We identified the same approach in functions such as: `GetAirdropIds`, `GetClaimRecords`, `GetClaimStatus`, `GetClaimMetadata`, `GetAirdropIdentifiersForUser`, `IncrementClaimedSoFar`, `ResetClaimedSoFar`, `DeleteAirdropAndEpoch`.

Recommendation

To optimize the performance and reduce slice resizing overhead, consider the following recommendation:

1. **Estimate the Expected Size:** Determine an estimate of the expected size based on the number of lines in the `allocationData`. If the number of lines is known in advance or can be reasonably estimated, use that value to reserve space for the `records` slice.
2. **Reserve Space in Advance:** Before iterating over the lines and appending elements to the `records` slice, use the `make` function to create a slice with the reserved capacity. The capacity should be set to the estimated size to avoid frequent resizing.

```

func (k Keeper) LoadAllocationData(ctx sdk.Context, allocationData string) bool {

    lines := strings.Split(allocationData, "\n")
    estimatedSize := len(lines) // Estimate the expected size based on the number of
lines
    records := make([]types.ClaimRecord, 0, estimatedSize) // Reserve space in advance

    // Rest of the code remains the same

    // Append elements to the records slice
    records = append(records, types.ClaimRecord{
        // Populate the ClaimRecord struct
    })

    // Rest of the code remains the same

```

```
}
```

However, it's important to note that preallocating space for a slice should be done when you have a good estimate of the expected size and when the size is relatively large compared to the default capacity of slices. Preallocation may not provide significant benefits for small or dynamically changing sizes and can potentially waste memory, so it's always recommended to benchmark and profile your code to validate the impact of preallocation in your specific use case.

Consider refactoring ClaimCoinsForAction function

Title	Consider refactoring ClaimCoinsForAction function
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

```
func (k Keeper) ClaimCoinsForAction(ctx sdk.Context, addr sdk.AccAddress, action
types.Action, airdropIdentifier string) (sdk.Coins, error) {
    isPassed := k.IsInitialPeriodPassed(ctx, airdropIdentifier)
    if airdropIdentifier == "" {
        return nil, errorsmod.Wrapf(sdkerrors.ErrInvalidRequest, "invalid airdrop
identifier: ClaimCoinsForAction")
    }

    claimableAmount, err := k.GetClaimableAmountForAction(ctx, addr, action,
airdropIdentifier, false)
    if err != nil {
        return claimableAmount, err
    }

    if claimableAmount.Empty() {
        return claimableAmount, nil
    }

    claimRecord, err := k.GetClaimRecord(ctx, addr, airdropIdentifier)
    if err != nil {
        return nil, err
    }

    // Only BaseAccounts and StridePeriodicVestingAccount can claim
    acc := k.accountKeeper.GetAccount(ctx, addr)
```

```

_, isStrideVestingAccount := acc.(*vestingtypes.StridePeriodicVestingAccount)
_, isBaseAcc := acc.(*authtypes.BaseAccount)
canClaim := isStrideVestingAccount || isBaseAcc
if !canClaim {
    return nil, errorsmod.Wrapf(types.ErrInvalidAccount, "Account: %v", acc)
}

// Claims don't vest if action type is ActionFree or initial period of vesting is
passed
if !isPassed {
    acc = k.accountKeeper.GetAccount(ctx, addr)
    strideVestingAcc, isStrideVestingAccount := acc.
(*vestingtypes.StridePeriodicVestingAccount)
    // Check if vesting tokens already exist for this account.
    if !isStrideVestingAccount {
        // Convert user account into stride vesting account.
        baseAccount := k.accountKeeper.NewAccountWithAddress(ctx, addr)
        if _, ok := baseAccount.(*authtypes.BaseAccount); !ok {
            return nil, errorsmod.Wrapf(sdkerrors.ErrInvalidRequest, "invalid
account type; expected: BaseAccount, got: %T", baseAccount)
        }

        periodLength := GetAirdropDurationForAction(action)
        vestingAcc := vestingtypes.NewStridePeriodicVestingAccount(baseAccount.
(*authtypes.BaseAccount), claimableAmount, []vestingtypes.Period{{
            StartTime: ctx.BlockTime().Unix(),
            Length:    periodLength,
            Amount:    claimableAmount,
            ActionType: int32(action),
        }})
        k.accountKeeper.SetAccount(ctx, vestingAcc)
    } else {
        // Grant a new vesting to the existing stride vesting account
        periodLength := GetAirdropDurationForAction(action)
        strideVestingAcc.AddNewGrant(vestingtypes.Period{
            StartTime: ctx.BlockTime().Unix(),
            Length:    periodLength,
            Amount:    claimableAmount,
            ActionType: int32(action),
        })
        k.accountKeeper.SetAccount(ctx, strideVestingAcc)
    }
}

distributor, err := k.GetAirdropDistributor(ctx, airdropIdentifier)
if err != nil {
    return nil, err
}

err = k.bankKeeper.SendCoins(ctx, distributor, addr, claimableAmount)
if err != nil {
    return nil, err
}

claimRecord.ActionCompleted[action] = true

```

```

    err = k.SetClaimRecord(ctx, claimRecord)
    if err != nil {
        return claimableAmount, err
    }

    airdrop := k.GetAirdropByIdentifier(ctx, airdropIdentifier)
    if airdrop == nil {
        return nil, errorsmod.Wrapf(sdkerrors.ErrInvalidRequest, "invalid airdrop
identifier: ClaimCoinsForAction")
    }
    err = k.AfterClaim(ctx, airdropIdentifier,
claimableAmount.AmountOf(airdrop.ClaimDenom))
    if err != nil {
        return nil, err
    }

    ctx.EventManager().EmitEvents(sdk.Events{
        sdk.NewEvent(
            types.EventTypeClaim,
            sdk.NewAttribute(sdk.AttributeKeySender, addr.String()),
            sdk.NewAttribute(sdk.AttributeKeyAmount, claimableAmount.String()),
        ),
    })

    return claimableAmount, nil
}

```

- Unnecessary check on lines 38-40 can be removed since the account is guaranteed to be a `BaseAccount` at that point.
- Code Duplication: The code currently retrieves the account using `k.accountKeeper.GetAccount(ctx, addr)` twice, which is redundant. It would be more efficient to retrieve the account once and reuse the `account` variable throughout the function.
- The check for `airdropIdentifier` should be performed before forwarding it to `IsInitialPeriodPassed` (line 3).
- The variable `isPassed` is defined at the top of the function but used in the middle, which is not a recommended practice. It is better to postpone the variable's definition until right before it is used. This way, if there is an early return or exit before the variable is used, the call to `IsInitialPeriodPassed` would be avoided.

Improve code organization and separation of responsibilities

Title	Improve code organization and separation of responsibilities
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

The provided code includes a function called `ResetClaimStatus` which has the responsibility of resetting the claim status and the `ClaimedSoFar` value of an airdrop. However, the function is handling two distinct tasks in a single method, violating the principle of separation of responsibilities.

```
// ResetClaimStatus clear users' claimed status only after initial period of vesting
is passed
func (k Keeper) ResetClaimStatus(ctx sdk.Context, airdropIdentifier string) error {
    passed := k.IsInitialPeriodPassed(ctx, airdropIdentifier)
    k.Logger(ctx).Info(fmt.Sprintf("[CLAIM] k.IsInitialPeriodPassed(ctx,
airdropIdentifier) %v", passed))
    if passed {
        k.Logger(ctx).Info("Resetting claim status")
        // first, reset the claim records
        records := k.GetClaimRecords(ctx, airdropIdentifier)
        k.Logger(ctx).Info(fmt.Sprintf("[CLAIM] len(records) %v", len(records)))
        for idx := range records {
            records[idx].ActionCompleted = []bool{false, false, false}
        }

        k.Logger(ctx).Info("[CLAIM] SetClaimRecords...")
        if err := k.SetClaimRecords(ctx, records); err != nil {
            k.Logger(ctx).Info(fmt.Sprintf("[CLAIM] SetClaimRecords %v",
err.Error()))
            return err
        }
    }
}
```

```

        // then, reset the airdrop ClaimedSoFar
        k.Logger(ctx).Info("[CLAIM] ResetClaimedSoFar...")
        if err := k.ResetClaimedSoFar(ctx, airdropIdentifier); err != nil {
            k.Logger(ctx).Info(fmt.Sprintf("[CLAIM] ResetClaimedSoFar %v",
err.Error()))
            return err
        }
    }
    return nil
}

```

Problem Scenarios

1. **Mixing Responsibilities:** The `ResetClaimStatus` function is responsible for both resetting the claim status of individual claim records and resetting the `ClaimedSoFar` value of the airdrop. Combining these tasks into a single function violates the single responsibility principle, making the code less modular and potentially harder to understand, maintain, and test.
2. **Code Coupling:** The current implementation tightly couples the logic for resetting claim records and resetting the airdrop's `ClaimedSoFar` value within the same function. This coupling can make it challenging to modify or extend one aspect without impacting the other. It also reduces code reusability since the resetting of claim records cannot be easily separated and used in other parts of the codebase independently.

Recommendation

To address the problem scenario mentioned above, it is recommended to separate the responsibilities of resetting claim records and resetting the airdrop's `ClaimedSoFar` value into distinct functions. Here's a recommended approach:

1. Create a separate function, such as `ResetClaimRecordsStatus`, that is responsible for resetting the claim status of individual claim records. This function can iterate through the claim records and update the `ActionCompleted` field accordingly.
2. Extract the logic for resetting the airdrop's `ClaimedSoFar` value into a separate function, such as `ResetAirdropClaimedSoFar`. This function should handle the specific task of resetting the `ClaimedSoFar` value for the given airdrop.
3. Modify the `ResetClaimStatus` function to call the newly created `ResetClaimRecordsStatus` and `ResetAirdropClaimedSoFar` functions separately, based on their respective responsibilities.

By separating the responsibilities into individual functions, you achieve better code organization, maintainability, and reusability. Each function becomes focused on a specific task, making it easier to understand, test, and modify in the future. It also reduces the coupling between different aspects of the code, enabling more flexibility and scalability.

Unnecessary Object Creation in Arithmetic Operations

Title	Unnecessary Object Creation in Arithmetic Operations
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

Immutable functions (`Add` , `Sub` , `Mul` , `Quo`) for arithmetic operations are used in situations where there is no need to create new objects. This leads to unnecessary memory allocation. One such example is shown below:

```
func (k Keeper) SetClaimRecordsWithWeights(ctx sdk.Context, claimRecords
[]types.ClaimRecord) error {
    // Set total weights
    weights := make(map[string]sdk.Dec)
    for _, record := range claimRecords {
        if weights[record.AirdropIdentifier].IsNil() {
            weights[record.AirdropIdentifier] = sdk.ZeroDec()
        }

        weights[record.AirdropIdentifier] =
weights[record.AirdropIdentifier].Add(record.Weight)
    }

    // DO NOT REMOVE: StringMapKeys fixes non-deterministic map iteration
    for _, identifier := range utils.StringMapKeys(weights) {
        weight := weights[identifier]
        k.SetTotalWeight(ctx, weight, identifier)
    }

    // Set claim records
    return k.SetClaimRecords(ctx, claimRecords)
}
```


Problem Scenarios

When you use the `Add` method in `weights[record.AirdropIdentifier] = weights[record.AirdropIdentifier].Add(record.Weight)`, it creates a new `sdk.Dec` object with the result of the addition. This means that for each iteration of the loop or each arithmetic operation, a new object is allocated in memory to hold the result.

Creating new objects for each operation can lead to increased memory usage, as each new object takes up memory space. Additionally, the allocation and deallocation of these objects can incur overhead and impact the performance of your code, especially if you perform a large number of arithmetic operations or if the loop is executed frequently.

There are some other places in the code where use of immutable functions was detected:

1. `GetUserTotalClaimable()`
 - a. `totalClaimable = totalClaimable.Add(claimableForAction...)`
2. `ClaimAllCoinsForAction()`
 - a. `totalClaimable = totalClaimable.Add(claimable...)`
3. `IncrementClaimedSoFar()`
 - a. `airdrop.ClaimedSoFar = airdrop.ClaimedSoFar.Add(amount)`

Recommendation

By using `AddMut`, you can avoid creating new objects and modify the existing `sdk.Dec` value directly. This eliminates the need for memory allocation and reduces the overhead associated with object creation, resulting in improved performance and reduced memory usage.

Therefore, using `AddMut` is recommended in scenarios where you have repeated arithmetic operations or performance-sensitive code to optimize memory allocation and improve overall efficiency.

It is also recommended to replace other immutable operations (such as `Sub`, `Mul`, `Quo`) with their mutable counterparts (`SubMut`, `MulMut`, `QuoMut`) where applicable.

Miscellaneous Code Improvements and Refinements

Title	Miscellaneous Code Improvements and Refinements
Project	Stride - Autopilot v2 & Claim
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- x/claim/keeper/claim.go

Description

Redundant Error Handling and Return Statement

```
err := k.IncrementClaimedSoFar(ctx, airdropIdentifier, claimAmount)
if err != nil {
    return err
}
return nil
```

The code snippet demonstrates a problematic implementation of error handling and return statements. It unnecessarily assigns the error value from the `IncrementClaimedSoFar` method to the `err` variable and checks if it is not `nil`. If there is an error, it is immediately returned. Otherwise, a separate `return nil` statement follows.

This can be simplified with:

```
return k.IncrementClaimedSoFar(ctx, airdropIdentifier, claimAmount)
```

Silent Error Skipping without Indication or Logging

```
func (k Keeper) LoadAllocationData(ctx sdk.Context, allocationData string) bool {
    records := []types.ClaimRecord{}
    lines := strings.Split(allocationData, "\\n")
    allocatedFlags := map[string]bool{}
    for _, line := range lines {
        data := strings.Split(line, ",")
        if data[0] == "" || data[1] == "" || data[2] == "" {
```

```

        continue
    }

    airdropIdentifier := data[0]
    sourceChainAddr := data[1]
    airdropWeight := data[2]
    strideAddr := utils.ConvertAddressToStrideAddress(sourceChainAddr)
    if strideAddr == "" {
        continue
    }
    allocationIdentifier := airdropIdentifier + strideAddr

    // Round weight value so that it always has 10 decimal places
    weightFloat64, err := strconv.ParseFloat(airdropWeight, 64)
    if err != nil {
        continue
    }

    weightStr := fmt.Sprintf("%.10f", weightFloat64)
    weight, err := sdk.NewDecFromStr(weightStr)
    if weight.IsNegative() || weight.IsZero() {
        continue
    }

    if err != nil || allocatedFlags[allocationIdentifier] {
        continue
    }

    _, err = sdk.AccAddressFromBech32(strideAddr)
    if err != nil {
        continue
    }
    ...

```

Explanation: The code snippet demonstrates a problematic use of the `continue` statement to silently skip errors without providing any indication or logging. When an error occurs in certain sections of the code, such as during parsing or validation, the `continue` statement is used to bypass the current iteration and move on to the next iteration of the loop.

This approach can be problematic as it suppresses potential errors without any indication or logging, making it difficult to identify and handle issues that may arise. Silently skipping errors can lead to unexpected behavior and make it challenging to troubleshoot problems in the code.

A better practice would be to handle errors explicitly by logging or reporting them, allowing for better visibility and appropriate action when errors occur.


Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of [Common Vulnerability Scoring System \(CVSS\) v3.1](#), which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the [Impact score](#), and the [Exploitability score](#). The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ [CVSS Qualitative Severity Rating Scale](#), and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
 None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

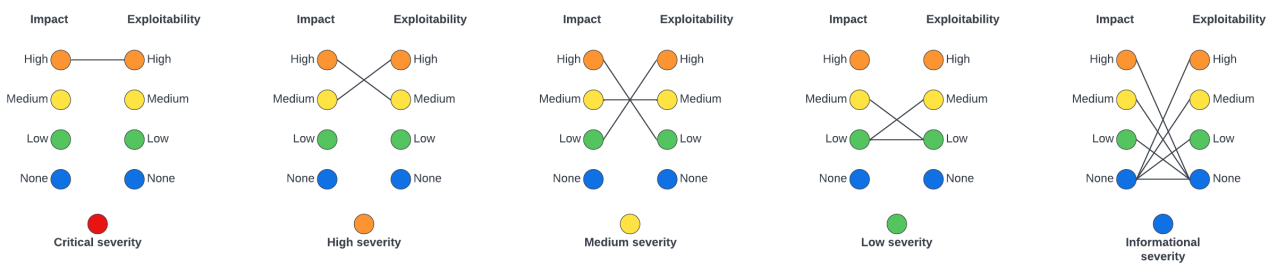
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
- *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
● None	illegitimate actions taken in a coordinated fashion by all actors


Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.



As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
● Critical	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
 Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal Systems has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal Systems makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal Systems to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.