BEOSIN
Blockchain Security

Self Chain

Blockchain Security Audit

No. 202501251627

Jan 25th, 2025

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Result

After auditing, 1 Medium-risks ,1 Low-risks and 1 Info items were identified in the Self Chain. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

Medium	Fixed : 1 Acknowledged: 0
Low	Fixed : 1 Acknowledged: 0
Info	Fixed : 1 Acknowledged: 0

Business overview

The Self Chain is a Layer1 blockchain built on Cosmos. In addition to the basic modules of Cosmos, more details can be found in past audit reports: https://beosin.com/audits/Self-Chain_202404191527.pdf

In the audit of investing-schedule-updates, the main updates are as follows:

- Added support for wasm contracts.
- Added the IBC fee module, which provides incentives for cross-chain transactions, ensuring that cross-chain transactions are more efficient and reliable, while incentivizing relayers to participate in the forwarding of transactions.
- Added upgrade module , which primarily handles data updates for certain VestingAccount. This includes two main types: one is to postpone the pending unvested amounts of vestingAddresses by three months, and the other is to replace the pending vesting schedule of currentAddress set in addressReplacements to newAddress.

1 Overview

1.1 Project Overview

Project Name	Self Chain
Project Language	Go
Platform	Self Chain
Code Base	https://github.com/selfchainxyz/selfchain/tree/vesting-schedule-updates
Commit ID	e4099ec71dae7369edf7c77b76b52aae62b7d6ce 30da8503bb154a37226cf4cc79b4e471ba29a866 ec0b20c109f5147a0658d26154ed5752a5f1f506

1.2 Audit Overview

Audit work duration: Jan 9, 2025 – Jan 25, 2025

Audit team: Beosin Security Team

2 Findings

Index	Risk description	Severity level	Status
Self Chain-01	Execution order of the RunMigrations function	**Medium**	**Fixed**
Self Chain-02	The lock time is reset after migration	**Low**	**Fixed**
Self Chain-03	Redundant code	**Info**	**Fixed**

[Self Chain-01] Execution order of the RunMigrations function

Severity Level	**Medium**
Lines	upgrades/v2/handler.go#L54-L70
Description	In the `CreateUpgradeHandler` function, the `RunMigrations` operation should be performed at the very beginning. This prevents the chain state from being modified before the migration, which can cause conflicts and lead to data consistency errors.

```
func CreateUpgradeHandler(
    mm *module.Manager,
    configurator module.Configurator,
    accountKeeper authkeeper.AccountKeeper,
) upgradetypes.UpgradeHandler {
    return func(ctx sdk.Context, plan upgradetypes.Plan, fromVM
module.VersionMap) (module.VersionMap, error) {
        ctx.Logger().Info("Starting upgrade v2")
        if err := updateVestingSchedules(ctx, accountKeeper); err !=
nil {
            ctx.Logger().Error("Failed to execute v2 upgrade",
"error", err)
            return nil, err
        }
        ctx.Logger().Info("Completed upgrade v2 successfully")
        return mm.RunMigrations(ctx, configurator, fromVM)
    }
}
```

Recommendation	It is recommended that `RunMigrations` be set at the front of the function.
Status	**Fixed.** Changed the order of execution and added comments.

```
func CreateUpgradeHandler(mm *module.Manager, configurator
module.Configurator, accountKeeper authkeeper.AccountKeeper,
bankkeeper bankkeeper.Keeper, ) upgradetypes.UpgradeHandler {
    return func(ctx sdk.Context, plan upgradetypes.Plan, fromVM
module.VersionMap) (module.VersionMap, error) {
        ctx.Logger().Info("Starting upgrade v2")
        // 1. Run all module migrations first
        newVM, err := mm.RunMigrations(ctx, configurator, fromVM)
        if err != nil {
```

```
            ctx.Logger().Error("Failed to run module migrations for v2",
"error", err)
            return nil, err
        }
        // 2. After all modules are migrated, run your custom logic
        if err := updateVestingSchedules(ctx, accountKeeper,
bankkeeper); err != nil {
            ctx.Logger().Error("Failed to execute v2 upgrade
(vestings)", "error", err)
            return nil, err
        }
        ctx.Logger().Info("Completed upgrade v2 successfully")
        return newVM, nil
    }
}
```

[Self Chain-02] The lock time is reset after migration

Severity Level	Low
Lines	upgrades/v2/handler.go#L181-L197
Description	When migrating a locked address to a new address, the user's lockout time starts over and is not subtracted from the time already locked. For example, if the lock time is 30 days and the user migrates on day 10, they will need to wait another 30 days to unlock instead of 20 days.

```
      // Create new account with only unvested amounts
    newAcc := &vestingtypes.PeriodicVestingAccount{
        BaseVestingAccount: &vestingtypes.BaseVestingAccount{
            BaseAccount:      baseAcc,
            OriginalVesting:  unvestedCoins,
            DelegatedFree:    sdk.NewCoins(),
            DelegatedVesting: sdk.NewCoins(),
            EndTime:          oldAcc.EndTime,
        },
        StartTime:      currentTime,
        VestingPeriods: unvestedPeriods,
    }
```

Recommendation	It is recommended that the elapsed locking time be subtracted from the migration and that the unlocking time for the new address be updated.
Status	**Fixed.** Added `firstUnVested` for updating the user's lockout time.

```
    firstUnVested := true
    // Find unvested periods
    for i, period := range oldAcc.VestingPeriods {
        if cumulativeTime+period.Length > currentTime {
            // This and all subsequent periods are unvested
            unvestedPeriods = append(unvestedPeriods,
oldAcc.VestingPeriods[i:]...)
            for _, p := range oldAcc.VestingPeriods[i:] {
                unvestedCoins = unvestedCoins.Add(p.Amount...)
            }
            if firstUnVested {
                usedInThisPeriod := currentTime - cumulativeTime
                partialElapsed = usedInThisPeriod
                firstUnVested = false
```

```
            }
            break
        }
        vestedPeriods = append(vestedPeriods, period)
        cumulativeTime += period.Length
    }
```

[Self Chain-03] Redundant code

Severity Level	Info
Lines	app/app.go #L857-875
Description	As shown below, when `loadLatest` is executed, `SetUpgradeHandler` will be executed twice with the same data.

```
app.UpgradeKeeper.SetUpgradeHandler(
    "v2",
    func(ctx sdk.Context, plan upgradetypes.Plan, vm
module.VersionMap) (module.VersionMap, error) {
        return app.mm.RunMigrations(ctx, app.configurator, vm)
    },
)
if loadLatest {
    app.UpgradeKeeper.SetUpgradeHandler(
        v2.UpgradeName,
        v2.CreateUpgradeHandler(
            app.mm,
            app.configurator,
            app.AccountKeeper,
        ),
    )
```

Recommendation	It is recommended to remove redundant code or add corresponding logic according to the design requirements.
Status	**Fixed.** The `SetUpgradeHandler` will now only be executed at most once in the `loadLatest` case, depending on the condition.

```
    if manager := app.SnapshotManager(); manager != nil {
        err := manager.RegisterExtensions(
            wasmkeeper.NewWasmSnapshotter(app.CommitMultiStore(),
&app.WasmKeeper),
        )
        if err != nil {
            panic(fmt.Errorf("failed to register snapshot
extension: %s", err))
        }
    }
    if loadLatest {
```

```
        if isDevelopmentEnv() {
            if err := app.LoadLatestVersion(); err != nil {
                tmos.Exit(err.Error())
            }
            ctx := app.BaseApp.NewUncachedContext(true,
tmproto.Header{})
            if err := app.WasmKeeper.InitializePinnedCodes(ctx); err !=
nil {
                tmos.Exit(fmt.Sprintf("failed initialize pinned
codes %s", err))
            }
        } else {
            app.UpgradeKeeper.SetUpgradeHandler(
                v2.UpgradeName,
                v2.CreateUpgradeHandler(
                    app.mm,
                    app.configurator,
                    app.AccountKeeper,
                    app.BankKeeper,
                ),
            )
```

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1(Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

4.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

4.1.3 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

4.1.4 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.3 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com