BLOCKSEC

Security Audit

Report for Atoshi Chain

Date: July 10, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Atoshi
Target	Atoshi Chain

Version History

Version	Date	Description
1.0	July 10, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-17 02:52:55+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	L1 Chain
Language	Go
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Atoshi Chain of Atoshi.

The Atoshi Chain of Atoshi is a Cosmos SDK-based blockchain featuring the native ATOS coin with three custom modules: energy, oracle, and tokenomic. The energy module enables ATOS holders to accrue energy (a resource for offsetting gas fees) based on their ATOS holdings and delegate it to other accounts via ATOS locking. The oracle module allows whitelisted feeders to submit price and volume data. The tokenomic module distributes a predefined token supply across four pools (miner immediate, miner locked, project treasury, and migration airdrop), with price-gated progressive unlocking driven by the oracle feed.

Note this audit focuses on the code located in the following directories/files: x/oracle/keeper, x/oracle/types, x/tokenomics/keeper, x/tokenomics/types, x/energy/keeper, x/energy/types, x/energy/ante/decorator.go, proto/atoshi/oracle/v1/, proto/atoshi/tokenomics/v1/ and proto/atoshi/energy/v1/, excluding the following directories/files:

- *.pb.go
- *.pb.gw.go
- *_test.go
- x/energy/keeper/grpc_query.go

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Atoshi-chain	`Version 1`	`67e450e56e2a638d2d3ac643d82a92003af732ee`
	`Version 2`	`7007bcddefd99f3abbecb80aba7c381fa84156f6`

[1] https://github.com/ATOSHI-ORG/Atoshi-chain.git

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation

- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **twenty one** potential security issues. Besides, we have **four** recommendations and **four** notes.

- High Risk: 13
- Medium Risk: 6
- Low Risk: 2
- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	High	Incorrect implementation of the function `EligibleBalance()`	Security Issue	Fixed
2	High	Permanent loss of energy when the message execution fails	Security Issue	Fixed
3	High	Incorrect energy accounting after releasing delegations	Security Issue	Fixed
4	High	Unreachable energy consumption for deployment messages	Security Issue	Fixed
5	High	Lack of registering the hook function `OnBalanceChange()`	Security Issue	Fixed
6	High	Improper token releases due to the use of stale price data	Security Issue	Fixed
7	High	Incorrect calculation of `freeBalance` in the function `Delegate()`	Security Issue	Fixed
8	High	Improper calculation of the available energy in the function `Consume()`	Security Issue	Fixed
9	High	Stale updates of the energy account in the function `Delegate()`	Security Issue	Fixed
10	High	Improper energy consumption in the function `attributeDelegatedConsumption()`	Security Issue	Fixed
11	High	Improper price updating mechanism	Security Issue	Fixed
12	High	Improper refunding of unused energy	Security Issue	Fixed
13	High	Violation of the fixed-supply assumption due to the inflation module	Security Issue	Fixed
14	Medium	Potential fee evasion due to the improper fee calculation in the function `computeShortfallFee()`	Security Issue	Fixed
15	Medium	Ineffective priority due to the use of `NoOpMempool`	Security Issue	Fixed

16	Medium	Incorrect KV prefix usage in the function `GetPriceHistory()`	Security Issue	Fixed
17	Medium	Incorrect priority calculation in the function `getTxPriority()`	Security Issue	Fixed
18	Medium	Lack of validation for `gs.PriceHistory` in the function InitGenesis()	Security Issue	Fixed
19	Medium	Lack of overflow protection in the functions `TxEnergyCapacity()` and `DeployRecoverPerSecond()`	Security Issue	Fixed
20	Low	Potential inconsistent reward accounting when there are no bonded validators	Security Issue	Fixed
21	Low	Inconsistent implementation of the function `getTxPriority()`	Security Issue	Fixed
22	-	Complete the validity checks in the function `Validate()` of the module `x/tokenomics`	Recommendation	Fixed
23	-	Avoid panics in block handlers	Recommendation	Fixed
24	-	Remove redundant code	Recommendation	Fixed
25	-	Unify `KVStore` serialization across all custom modules	Recommendation	Fixed
26	-	`TotalMinerLocked` only increases and never decreases	Note	-
27	-	The construction of Merkle proofs for the pre-mine migration	Note	-
28	-	Ensure proper price reports	Note	-
29	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect implementation of the function `EligibleBalance()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `keeper.go` of the module `x/energy`, the function `EligibleBalance()` calculates accounts' eligible balance, which intends to include the locked ATOS coins for the calculation of the energy capacity (i.e., via the function `TxEnergyCapacity()`). However, the function `EligibleBalance()` returns the raw bank balance without including the locked ATOS coins. As a result, the incorrect implementation of the function `EligibleBalance()` causes incorrect capacity calculations, leading to potential loss of energy.

```
199 func (k Keeper) EligibleBalance(ctx sdk.Context, addr sdk.AccAddress) math.Int {
200   bal := k.bankKeeper.GetBalance(ctx, addr, k.baseDenom).Amount
201   acct := k.GetEnergyAccount(ctx, addr)
202   if acct.LockedAtos.IsNil() {
203     return bal
204   }
205   eligible := bal // bank already excludes locked (locked sits in module account)
206   _ = eligible
207   return bal
208 }
```

Listing 2.1: x/energy/keeper/keeper.go

Impact Potential loss of energy due to the incorrect implementation of the function `EligibleBalance()`.

Suggestion Revise the function `EligibleBalance()` by including users' locked ATOS coins.

2.1.2 Permanent loss of energy when the message execution fails

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the files `decorator.go` and `post.go` of the module `x/energy`, the functions `Consume()` and `RefundEnergy()` are triggered in the `AnteHandler` and `PostHandler` to deduct and refund energy for the incoming messages. The `AnteHandler` persists the state changes regarding the energy deduction before the message execution. However, the state changes in the `PostHandler` are discarded when the message fails. As a result, users may permanently lose their energy when the message execution fails, violating the intended design.

```
137 consumed, err := d.energyKeeper.Consume(ctx, deductFrom, gasLimit, isDeploy, msgUrls)
```

Listing 2.2: x/energy/ante/decorator.go

```
73 d.energyKeeper.RefundEnergy(ctx, signer, refund)
```

Listing 2.3: x/energy/ante/post.go

Impact Users may permanently lose their energy when the message execution fails, violating the intended design.

Suggestion Revise the logic accordingly.

2.1.3 Incorrect energy accounting after releasing delegations

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `delegation.go` of the module `x/energy`, the function `Delegate()` computes a user's available energy (i.e., `freeEnergy`) from its `TxEnergyAccrued` and `DelegatedOut`.

However, when delegations are released via the function `Undelegate()`, energy consumed by the delegatee is not deducted from the delegator's available energy, leading to the incorrect energy accounting. As a result, a delegator can repeatedly invoke the functions `Delegate()` and `Undelegate()` to reuse consumed energy.

```
23 func (k Keeper) Delegate(
24   ctx sdk.Context, delegator, delegatee sdk.AccAddress, amount uint64, durationSeconds int64,
25 ) (uint64, math.Int, error) {
26   if amount == 0 {
27     return 0, math.ZeroInt(), types.ErrInvalidAmount
28   }
29   if durationSeconds <= 0 {
30     return 0, math.ZeroInt(), types.ErrInvalidDuration
31   }
32   if delegator.Equals(delegatee) {
33     return 0, math.ZeroInt(), types.ErrSelfDelegation
34   }
35
36   params := k.GetParams(ctx)
37
38   // Lock ATOS = amount / per_threshold * threshold (rounded UP so that
39   // fractional units still freeze a full block of ATOS).
40   if params.TxEnergyPerThreshold == 0 || params.TxEnergyHoldingThreshold.IsZero() {
41     return 0, math.ZeroInt(), fmt.Errorf("invalid params")
42   }
43   thresholdUnits := (amount + params.TxEnergyPerThreshold - 1) / params.TxEnergyPerThreshold
44   if thresholdUnits == 0 {
45     thresholdUnits = 1
46   }
47   lockedATOS := params.TxEnergyHoldingThreshold.MulRaw(int64(thresholdUnits))
48
49   // Settle delegator first so their TxEnergyAccrued reflects the moment.
50   delAcct := k.Settle(ctx, delegator)
51
52   // Verify the delegator actually owns enough free energy to lend.
53   freeEnergy := delAcct.TxEnergyAccrued
54   if delAcct.DelegatedOut > freeEnergy {
55     // defensive -should never happen
56     freeEnergy = 0
57   } else {
58     freeEnergy -= delAcct.DelegatedOut
59   }
60   if freeEnergy < amount {
61     return 0, math.ZeroInt(), types.ErrInsufficientEnergy
62   }
```

Listing 2.4: x/energy/keeper/delegation.go

```
130 func (k Keeper) releaseDelegation(ctx sdk.Context, d types.EnergyDelegation, eventType string)
        error {
131   delegator, err := sdk.AccAddressFromBech32(d.Delegator)
132   if err != nil {
133     return err
```

```
134   }
135   delegatee, err := sdk.AccAddressFromBech32(d.Delegatee)
136   if err != nil {
137     return err
138   }
139
140   remaining := uint64(0)
141   if d.Amount > d.Used {
142     remaining = d.Amount - d.Used
143   }
144
145   // Refund locked ATOS to delegator.
146   if d.LockedAtos.IsPositive() {
147     if err := k.bankKeeper.SendCoinsFromModuleToAccount(
148       ctx, types.LockedEnergyPoolName, delegator, sdk.NewCoins(sdk.NewCoin(k.baseDenom, d.
              LockedAtos)),
149     ); err != nil {
150       return err
151     }
152   }
153
154   // Update delegator account.
155   delAcct := k.Settle(ctx, delegator)
156   if delAcct.DelegatedOut >= d.Amount {
157     delAcct.DelegatedOut -= d.Amount
158   } else {
159     delAcct.DelegatedOut = 0
160   }
```

Listing 2.5: x/energy/keeper/delegation.go

Impact A delegator can repeatedly invoke the functions `Delegate()` and `Undelegate()` to reuse consumed energy.

Suggestion Revise the code accordingly.

2.1.4 Unreachable energy consumption for deployment messages

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `decorator.go` of the module `x/energy`, the function `isContractDeployMsg()` determines whether to consume `DeployEnergy`, but it only recognizes CosmWasm deployment messages and MsgEthereumTx. However, the current chain does not integrate CosmWasm, and MsgEthereumTx transactions are routed to a separate EVM ante handler rather than passing through the Cosmos energy ante path. As a result, the function `isContractDeployMsg()` does not match any currently reachable production transaction path and `DeployEnergy` cannot be spent.

```
135 // Probe energy first.
136 isDeploy := isContractDeployMsg(msgs)
```

```
137 consumed, err := d.energyKeeper.Consume(ctx, deductFrom, gasLimit, isDeploy, msgUrls)
```

Listing 2.6: x/energy/ante/decorator.go

```
251 func isContractDeployMsg(msgs []sdk.Msg) bool {
252   for _, m := range msgs {
253     switch sdk.MsgTypeURL(m) {
254     case "/cosmwasm.wasm.v1.MsgInstantiateContract",
255       "/cosmwasm.wasm.v1.MsgInstantiateContract2",
256       "/cosmwasm.wasm.v1.MsgStoreCode":
257       return true
258     case "/ethermint.evm.v1.MsgEthereumTx":
259       // EVM deployments are recognized by To == nil. The MonoDecorator
260       // handles them outside this Cosmos chain, so reaching here
261       // means the tx was wrapped as a Cosmos msg -treat as deploy.
262       return true
263     }
264   }
265   return false
266 }
```

Listing 2.7: x/energy/ante/decorator.go

Impact The consumption of `DeployEnergy` is unreachable.

Suggestion Revise the code accordingly.

2.1.5 Lack of registering the hook function `OnBalanceChange()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `settle.go` of the module `x/energy`, the function `OnBalanceChange()` is intended to run during coin transfers to synchronize users' energy states. However, this hook is not registered, so synchronization is never triggered on transfers. As a result, energy states can become inconsistent, violating the intended design.

```
77 // OnBalanceChange is called by the bank Send hook (and any other code
78 // that mutates an account's balance). It first settles using the OLD
79 // snapshot, then snaps the new eligible balance, then caps any over-
80 // capacity accrued energy down to the new ceiling.
81 //
82 // The cap-down step is deliberate: a holder that sells most of their
83 // stake should not retain a giant prefilled energy buffer. Otherwise
84 // rotating accounts could harvest energy with one wallet then move
85 // the ATOS to another wallet for free transactions everywhere.
86 func (k Keeper) OnBalanceChange(ctx sdk.Context, addr sdk.AccAddress) {
87   // Step 1: close the old epoch.
88   acct := k.Settle(ctx, addr)
89
90   // Step 2: read fresh eligible balance.
91   newEligible := k.EligibleBalance(ctx, addr)
```

```
 92   acct.LastBalanceSnapshot = newEligible
 93   acct.LastUpdatedTime = ctx.BlockTime().Unix()
 94
 95   // Step 3: cap accrued to the new (possibly lower) capacity.
 96   params := k.GetParams(ctx)
 97   newTxCap := types.TxEnergyCapacity(newEligible, params)
 98   if acct.TxEnergyAccrued > newTxCap {
 99     acct.TxEnergyAccrued = newTxCap
100   }
101   if acct.DeployEnergyAccrued > params.DeployEnergyCapacity {
102     acct.DeployEnergyAccrued = params.DeployEnergyCapacity
103   }
104
105   k.SetEnergyAccount(ctx, acct)
106 }
```

Listing 2.8: x/energy/keeper/settle.go

```
512 app.EnergyKeeper = energykeeper.NewKeeper(
513   appCodec,
514   keys[energytypes.StoreKey],
515   app.AccountKeeper,
516   app.BankKeeper,
517   authtypes.NewModuleAddress(govtypes.ModuleName).String(),
518   atoshitypes.BaseDenom,
519 )
```

Listing 2.9: app/app.go

Impact Energy states can become inconsistent, violating the intended design.

Suggestion Register the hook function `OnBalanceChange()`.

2.1.6 Improper token releases due to the use of stale price data

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `rewards.go` of the module `x/tokenomics`, the function `EndBlocker()` releases rewards when the defined requirement (i.e., `state.ConsecutiveDays` >= `params.ConsecutiveDaysRequired`) is satisfied. The state `ConsecutiveDays` only increments when the current price and volume (i.e., `priceData` = `GetCurrentPrice()`) satisfy the minimum requirements (i.e., `requiredPrice` and `requiredVolume`). However, the function `EndBlocker()` fetches the current price and volume without verifying their staleness. As a result, stale price data may be used, leading to improper token releases.

```
118 priceData, err := k.oracleKeeper.GetCurrentPrice(ctx)
119 if err != nil {
120   k.Logger(ctx).Error("failed to get oracle price", "err", err)
121   state.LastCheckBlock = ctx.BlockHeight()
122   return k.SetReleaseState(ctx, state)
```

```
123 }
124
125 requiredPrice := params.PriceBase.Mul(params.TierMultiplier.Power(uint64(state.CurrentTier)))
126 requiredVolume := params.VolumeBase.Mul(params.TierMultiplier.Power(uint64(state.CurrentTier)))
127
128 if priceData.Price.GTE(requiredPrice) && priceData.Volume24h.GTE(requiredVolume) {
129   state.ConsecutiveDays++
130 } else {
131   state.ConsecutiveDays = 0
132 }
133
134 state.LastCheckBlock = ctx.BlockHeight()
135 state.LastCheckTimeUnix = ctx.BlockTime().Unix()
136
137 if state.ConsecutiveDays >= params.ConsecutiveDaysRequired {
138   if err := k.TriggerRelease(ctx, &state, params); err != nil {
139     return err
140   }
141   state.ConsecutiveDays = 0
142   state.CurrentTier++
143 }
144
145 return k.SetReleaseState(ctx, state)
```

Listing 2.10: x/tokenomics/keeper/rewards.go

```
81 func (k Keeper) GetCurrentPrice(ctx sdk.Context) (types.PriceData, error) {
82   store := ctx.KVStore(k.storeKey)
83   bz := store.Get(types.KeyPrefixCurrentPrice)
84   if bz == nil {
85     return types.PriceData{}, types.ErrPriceNotFound
86   }
87   var price types.PriceData
88   if err := json.Unmarshal(bz, &price); err != nil {
89     return types.PriceData{}, err
90   }
91   return price, nil
92 }
```

Listing 2.11: x/oracle/keeper/keeper.go

```
17// DefaultParams returns sensible defaults for the oracle module.
18 func DefaultParams() Params {
19   return Params{
20     AllowedFeeders:    []string{},
21     MaxPriceAgeSeconds: 3600, // 1 hour
22     MinValidReports:   1,
23     Denom:             "aatos",
24     TWAPLookbackSeconds: 86400, // 24 hours
25     MaxPriceDeviationBps: 1000, // 10%
26   }
27 }
```

Listing 2.12: x/oracle/types/helpers.go

Impact Stale price data may be used, leading to improper token releases.

Suggestion Add staleness checks for the price data in the function `EndBlocker()`.

2.1.7 Incorrect calculation of `freeBalance` in the function `Delegate()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `delegation.go` of the module `x/energy`, the function `Delegate()` allows users to delegate their energy by locking required ATOS in the module account `LockedEnergyPoolName`. However, when calculating the free balance (i.e., `freeBalance`), the previously locked ATOS is deducted from the user's balance again. As a result, users may fail to delegate their energy due to the incorrect calculation of `freeBalance`.

```
65 bal := k.bankKeeper.GetBalance(ctx, delegator, k.baseDenom).Amount
66 currentLocked := delAcct.LockedAtos
67 if currentLocked.IsNil() {
68   currentLocked = math.ZeroInt()
69 }
70 freeBalance := bal.Sub(currentLocked)
71 if freeBalance.LT(lockedATOS) {
72   return 0, math.ZeroInt(), types.ErrInsufficientBalance
73 }
74
75 // Move the locked ATOS to the module account.
76 if err := k.bankKeeper.SendCoinsFromAccountToModule(
77   ctx, delegator, types.LockedEnergyPoolName, sdk.NewCoins(sdk.NewCoin(k.baseDenom, lockedATOS)),
78 ); err != nil {
79   return 0, math.ZeroInt(), err
80 }
```

Listing 2.13: x/energy/keeper/delegation.go

```
23func (k Keeper) Delegate(
24 ctx sdk.Context, delegator, delegatee sdk.AccAddress, amount uint64, durationSeconds int64,
25) (uint64, math.Int, error) {
26 if amount == 0 {
27   return 0, math.ZeroInt(), types.ErrInvalidAmount
28 }
29 if durationSeconds <= 0 {
30   return 0, math.ZeroInt(), types.ErrInvalidDuration
31 }
32 if delegator.Equals(delegatee) {
33   return 0, math.ZeroInt(), types.ErrSelfDelegation
34 }
35
36 params := k.GetParams(ctx)
37
38 // Lock ATOS = amount / per_threshold * threshold (rounded UP so that
39 // fractional units still freeze a full block of ATOS).
```

```
if params.TxEnergyPerThreshold == 0 || params.TxEnergyHoldingThreshold.IsZero() {
  return 0, math.ZeroInt(), fmt.Errorf("invalid params")
}
thresholdUnits := (amount + params.TxEnergyPerThreshold - 1) / params.TxEnergyPerThreshold
if thresholdUnits == 0 {
  thresholdUnits = 1
}
lockedATOS := params.TxEnergyHoldingThreshold.MulRaw(int64(thresholdUnits))

// Settle delegator first so their TxEnergyAccrued reflects the moment.
delAcct := k.Settle(ctx, delegator)

// Verify the delegator actually owns enough free energy to lend.
freeEnergy := delAcct.TxEnergyAccrued
if delAcct.DelegatedOut > freeEnergy {
  // defensive -should never happen
  freeEnergy = 0
} else {
  freeEnergy -= delAcct.DelegatedOut
}
if freeEnergy < amount {
  return 0, math.ZeroInt(), types.ErrInsufficientEnergy
}

// Verify free bank balance covers the lock.
bal := k.bankKeeper.GetBalance(ctx, delegator, k.baseDenom).Amount
currentLocked := delAcct.LockedAtos
if currentLocked.IsNil() {
  currentLocked = math.ZeroInt()
}
freeBalance := bal.Sub(currentLocked)
if freeBalance.LT(lockedATOS) {
  return 0, math.ZeroInt(), types.ErrInsufficientBalance
}

// Move the locked ATOS to the module account.
if err := k.bankKeeper.SendCoinsFromAccountToModule(
  ctx, delegator, types.LockedEnergyPoolName, sdk.NewCoins(sdk.NewCoin(k.baseDenom, lockedATOS)),
); err != nil {
  return 0, math.ZeroInt(), err
}

// Record on the delegator's account.
delAcct.DelegatedOut += amount
delAcct.LockedAtos = currentLocked.Add(lockedATOS)
// Bank balance dropped: snapshot reflects the new eligible balance.
delAcct.LastBalanceSnapshot = k.EligibleBalance(ctx, delegator)
k.SetEnergyAccount(ctx, delAcct)

// Settle delegatee then bump their usable inbound.
deeAcct := k.Settle(ctx, delegatee)
deeAcct.DelegatedInUsable = saturatingAdd(deeAcct.DelegatedInUsable, amount)
k.SetEnergyAccount(ctx, deeAcct)
```

```
 93
 94 // Allocate id and persist the delegation record.
 95 id := k.nextDelegationID(ctx)
 96 now := ctx.BlockTime().Unix()
 97 d := types.EnergyDelegation{
 98   Id:        id,
 99   Delegator: delegator.String(),
100   Delegatee: delegatee.String(),
101   Amount:    amount,
102   LockedAtos: lockedATOS,
103   StartTime: now,
104   ExpiresAt: now + durationSeconds,
105   Used:      0,
106 }
107 k.setDelegation(ctx, d)
108 return id, lockedATOS, nil
109}
```

Listing 2.14: x/energy/keeper/delegation.go

Impact Users may fail to delegate their energy due to the incorrect calculation of `freeBalance`.

Suggestion Revise the code accordingly.

2.1.8 Improper calculation of the available energy in the function `Consume()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `consume.go` of the module `x/energy`, the function `Consume()` recalculates a user's available energy by deducting their delegated energy (i.e., `ownAvail` = `TxEnergyAccrued` - `DelegatedOut`). However, the user's accrued energy is capped by a dynamic energy capacity (i.e., `TxEnergyAccrued` <= `TxEnergyCapacity()`), which is computed based on the user's balance of ATOS coins. Specifically, when the user transfers out their ATOS coins, their accrued energy may decrease. As a result, the design may cause incorrect calculation of users' available energy in the function `Consume()`, leading to potential DoS issues.

```
66 // Own TxEnergy (less anything currently delegated out -DelegatedOut
67 // is bookkeeping; the actual lent energy has been added to the
68 // delegatee's DelegatedInUsable, so subtracting here ensures we
69 // don't double-spend).
70 ownAvail := acct.TxEnergyAccrued
71 if acct.DelegatedOut < ownAvail {
72   ownAvail -= acct.DelegatedOut
73 } else {
74   ownAvail = 0
75 }
```

Listing 2.15: x/energy/keeper/consume.go

```
27 func (k Keeper) Settle(ctx sdk.Context, addr sdk.AccAddress) types.EnergyAccount {
28   acct := k.GetEnergyAccount(ctx, addr)
29   now := ctx.BlockTime().Unix()
30   if acct.LastUpdatedTime == 0 {
31     // First touch: initialize the snapshot to current eligible balance.
32     acct.LastBalanceSnapshot = k.EligibleBalance(ctx, addr)
33     acct.LastUpdatedTime = now
34     k.SetEnergyAccount(ctx, acct)
35     return acct
36   }
37   if now <= acct.LastUpdatedTime {
38     return acct
39   }
40
41   params := k.GetParams(ctx)
42   elapsed := uint64(now - acct.LastUpdatedTime)
43
44   // --- TxEnergy refill ---
45   // Compute (capacity * elapsed / window) to avoid sub-second
46   // integer-division underflow (50000 / 86400 == 0 in uint64).
47   txCap := types.TxEnergyCapacity(acct.LastBalanceSnapshot, params)
48   if txCap > 0 && acct.TxEnergyAccrued < txCap && params.TxEnergyMaxAccrueWindow > 0 {
49     add := saturatingMul(elapsed, txCap) / uint64(params.TxEnergyMaxAccrueWindow)
50     newAccrued := saturatingAdd(acct.TxEnergyAccrued, add)
51     if newAccrued > txCap {
52       newAccrued = txCap
53     }
54     acct.TxEnergyAccrued = newAccrued
55   }
```

Listing 2.16: x/energy/keeper/settle.go

Impact Potential DoS due to improper calculation of users' available energy in the function `Consume()`.

Suggestion Revise the code accordingly.

2.1.9 Stale updates of the energy account in the function `Delegate()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `delegation.go` of the module `x/energy`, the function `Delegate()` lets users lock ATOS coins to delegate free energy. The coins are transferred to the module account `LockedEnergyPoolName`, which triggers the hook function `OnBalanceState()` to synchronize energy states. However, after the coin transfer, the function `Delegate()` updates the user's energy account using stale state values (e.g., `delAcct.TxEnergyAccrued`). As a result, these stale updates cause inconsistent energy accounting, violating the intended design.

```
75 // Move the locked ATOS to the module account.
```

```
76 if err := k.bankKeeper.SendCoinsFromAccountToModule(
77   ctx, delegator, types.LockedEnergyPoolName, sdk.NewCoins(sdk.NewCoin(k.baseDenom, lockedATOS)),
78 ); err != nil {
79   return 0, math.ZeroInt(), err
80 }
81
82 // Record on the delegator's account.
83 delAcct.DelegatedOut += amount
84 delAcct.LockedAtos = currentLocked.Add(lockedATOS)
85 // Bank balance dropped: snapshot reflects the new eligible balance.
86 delAcct.LastBalanceSnapshot = k.EligibleBalance(ctx, delegator)
87 k.SetEnergyAccount(ctx, delAcct)
```

Listing 2.17: x/energy/keeper/delegation.go

```
86 func (k Keeper) OnBalanceChange(ctx sdk.Context, addr sdk.AccAddress) {
87   // Step 1: close the old epoch.
88   acct := k.Settle(ctx, addr)
89
90   // Step 2: read fresh eligible balance.
91   newEligible := k.EligibleBalance(ctx, addr)
92   acct.LastBalanceSnapshot = newEligible
93   acct.LastUpdatedTime = ctx.BlockTime().Unix()
94
95   // Step 3: cap accrued to the new (possibly lower) capacity.
96   params := k.GetParams(ctx)
97   newTxCap := types.TxEnergyCapacity(newEligible, params)
98   if acct.TxEnergyAccrued > newTxCap {
99     acct.TxEnergyAccrued = newTxCap
100  }
101  if acct.DeployEnergyAccrued > params.DeployEnergyCapacity {
102    acct.DeployEnergyAccrued = params.DeployEnergyCapacity
103  }
104
105  k.SetEnergyAccount(ctx, acct)
106 }
```

Listing 2.18: x/energy/keeper/settle.go

Impact The stale updates cause inconsistent energy accounting, violating the intended design.

Suggestion Revise the code accordingly.

2.1.10 Improper energy consumption in the function `attributeDelegatedConsumption()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `consume.go` of the module `x/energy`, the `attributeDelegatedConsumption()` function is invoked to consume energy from users' delegations to pay gas fees. However,

the function iterates delegations (via the function `IterateDelegationsByDelegatee()`) without considering delegation expiration times. As a result, some earlier delegations may become unreachable before expiration.

```
185 k.IterateDelegationsByDelegatee(ctx, delegatee.String(), func(d types.EnergyDelegation) bool {
186   if remaining == 0 {
187     return true
188   }
189   if d.Amount <= d.Used {
190     return false
191   }
192   avail := d.Amount - d.Used
193   take := minU64(remaining, avail)
194   d.Used += take
195   remaining -= take
196   toUpdate = append(toUpdate, d)
197   return remaining == 0
198 })
199 for _, d := range toUpdate {
200   k.setDelegation(ctx, d)
201 }
202 if remaining > 0 {
203   // Bookkeeping mismatch -DelegatedInUsable said we had more than
204   // the index actually exposes. Fail loudly.
205   return fmt.Errorf("delegated_in_usable accounting drift: %d unattributed", remaining)
206 }
```

Listing 2.19: x/energy/keeper/consume.go

```
166 func (k Keeper) IterateDelegationsByDelegatee(ctx sdk.Context, delegatee string, fn func(types.
        EnergyDelegation) bool) {
167   k.iterateDelegationsByIndex(ctx, types.DelegationByDelegateeKey(delegatee, 0)[:1+len(delegatee)
          +1], fn)
168 }
```

Listing 2.20: x/energy/keeper/keeper.go

```
72 // DelegationByDelegateeKey indexes inbound delegations.
73 func DelegationByDelegateeKey(delegatee string, id uint64) []byte {
74   prefix := append(KeyPrefixDelegationsByDelegee, []byte(delegatee)...)
75   prefix = append(prefix, '/')
76   idBz := make([]byte, 8)
77   binary.BigEndian.PutUint64(idBz, id)
78   return append(prefix, idBz...)
79 }
```

Listing 2.21: x/energy/types/keys.go

Impact Some earlier delegations may become unreachable before expiration.

Suggestion Iterate delegations based on the expiration time.

2.1.11 Improper price updating mechanism

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the file msg_server.go of the module x/oracle, the function ReportPrice() allows valid feeders to update current and historical prices via the functions SetCurrentPrice() and AppendePriceHistory(). However, the design cannot handle multiple price reports within the same block. As a result, when multiple prices are reported in a block, only the latest price is persisted successfully to the KV store due to the key prefixes used (i.e., KeyPrefixCurrentPrice and PriceHistoryKey(price.Timestamp)).

```
23 func (k msgServer) ReportPrice(goCtx context.Context, msg *types.MsgReportPrice) (*types.
       MsgReportPriceResponse, error) {
24   ctx := sdk.UnwrapSDKContext(goCtx)
25   params := k.GetParams(ctx)
26
27   if !params.IsAllowedFeeder(msg.Feeder) {
28     return nil, types.ErrUnauthorizedFeeder
29   }
30
31   priceData := types.PriceData{
32     Price:    msg.Price,
33     Volume24h: msg.Volume24h,
34     Timestamp: ctx.BlockTime().Unix(),
35     Feeder:   msg.Feeder,
36     Source:   msg.Source,
37   }
38
39   if err := k.SetCurrentPrice(ctx, priceData); err != nil {
40     return nil, err
41   }
42
43   if err := k.AppendPriceHistory(ctx, priceData); err != nil {
44     return nil, err
45   }
```

Listing 2.22: x/oracle/keeper/msg_server.go

```
71 func (k Keeper) SetCurrentPrice(ctx sdk.Context, price types.PriceData) error {
72   store := ctx.KVStore(k.storeKey)
73   bz, err := json.Marshal(price)
74   if err != nil {
75     return err
76   }
77   store.Set(types.KeyPrefixCurrentPrice, bz)
78   return nil
79 }
```

Listing 2.23: x/oracle/keeper/keeper.go

Moreover, in the module x/oracle, the defined security parameters such as MinValidReports and MaxPriceDeviationBps are never used (e.g., in the function ReportPrice()) for validation. As a result, any authorized feeder may report malicious or incorrect prices, leading to potential losses.

```go
func (k msgServer) ReportPrice(goCtx context.Context, msg *types.MsgReportPrice) (*types.MsgReportPriceResponse, error) {
  ctx := sdk.UnwrapSDKContext(goCtx)
  params := k.GetParams(ctx)

  if !params.IsAllowedFeeder(msg.Feeder) {
    return nil, types.ErrUnauthorizedFeeder
  }

  priceData := types.PriceData{
    Price:    msg.Price,
    Volume24h: msg.Volume24h,
    Timestamp: ctx.BlockTime().Unix(),
    Feeder:   msg.Feeder,
    Source:   msg.Source,
  }

  if err := k.SetCurrentPrice(ctx, priceData); err != nil {
    return nil, err
  }

  if err := k.AppendPriceHistory(ctx, priceData); err != nil {
    return nil, err
  }

  ctx.EventManager().EmitEvent(
    sdk.NewEvent(
      types.EventTypeReportPrice,
      sdk.NewAttribute(types.AttributeKeyPrice, msg.Price.String()),
      sdk.NewAttribute(types.AttributeKeyVolume, msg.Volume24h.String()),
      sdk.NewAttribute(types.AttributeKeyFeeder, msg.Feeder),
      sdk.NewAttribute(types.AttributeKeySource, msg.Source),
      sdk.NewAttribute(types.AttributeKeyTimestamp, fmt.Sprintf("%d", ctx.BlockTime().Unix())),
    ),
  )

  k.Logger(ctx).Info(
    "price reported",
    "price", msg.Price.String(),
    "volume", msg.Volume24h.String(),
    "feeder", msg.Feeder,
    "source", msg.Source,
  )

  return &types.MsgReportPriceResponse{}, nil
}
```

Listing 2.24: x/oracle/keeper/msg_server.go

```go
// allowed_feeders is the list of addresses authorized to submit price reports
AllowedFeeders []string `protobuf:"bytes,1,rep,name=allowed_feeders,json=allowedFeeders,proto3" json:"allowed_feeders,omitempty"`
// max_price_age_seconds is the maximum age of a price report before it's considered stale
```

```
101 MaxPriceAgeSeconds uint64 `protobuf:"varint,2,opt,name=max_price_age_seconds,json=
        maxPriceAgeSeconds,proto3" json:"max_price_age_seconds,omitempty"`
102 // min_valid_reports is the minimum number of valid reports needed for consensus
103 MinValidReports uint32 `protobuf:"varint,3,opt,name=min_valid_reports,json=minValidReports,
        proto3" json:"min_valid_reports,omitempty"`
104 // denom is the token denom being tracked (e.g., "aatos")
105 Denom string `protobuf:"bytes,4,opt,name=denom,proto3" json:"denom,omitempty"`
106 // twap_lookback_seconds defines the lookback window for TWAP calculation
107 TWAPLookbackSeconds uint64 `protobuf:"varint,5,opt,name=twap_lookback_seconds,json=
        twapLookbackSeconds,proto3" json:"twap_lookback_seconds,omitempty"`
108 // max_price_deviation_bps is the maximum allowed deviation between reports (basis points)
109 MaxPriceDeviationBps uint32 `protobuf:"varint,6,opt,name=max_price_deviation_bps,json=
        maxPriceDeviationBps,proto3" json:"max_price_deviation_bps,omitempty"`
110}
```

Listing 2.25: x/oracle/types/oracle.pb.go

Impact Potential losses due to the improper price updating mechanism.

Suggestion Using the safety parameters `MinValidReports` and `MaxPriceDeviationBps` for validation.

Clarification from BlockSec The project intends to protect valid price updates by increasing `MaxPriceDeviationBps` from 1000 (10%) to 5000 (50%). However, this mitigation can still fail when price deviation is extremely high.

2.1.12 Improper refunding of unused energy

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `consume.go` of the module `x/energy`, the function `RefundEnergy()` refunds unused energy to the message signer. However, the unused energy is potentially delegated by other users. As a result, the delegated energy is refunded to the message signer, violating the intended design.

```
89 if remaining > 0 && acct.DelegatedInUsable > 0 {
90   take := minU64(remaining, acct.DelegatedInUsable)
91   acct.DelegatedInUsable -= take
92   out.EnergyDeducted += take
93   remaining -= take
94   if err := k.attributeDelegatedConsumption(ctx, signer, take); err != nil {
95     return out, err
96   }
97 }
```

Listing 2.26: x/energy/keeper/consume.go

```
123 func (k Keeper) RefundEnergy(ctx sdk.Context, signer sdk.AccAddress, amount uint64) {
124   if amount == 0 {
125     return
126   }
```

```
127   acct := k.GetEnergyAccount(ctx, signer)
128   params := k.GetParams(ctx)
129   cap := types.TxEnergyCapacity(acct.LastBalanceSnapshot, params)
130   acct.TxEnergyAccrued = minU64(saturatingAdd(acct.TxEnergyAccrued, amount), cap)
131   k.SetEnergyAccount(ctx, acct)
132 }
```

Listing 2.27: x/energy/keeper/consume.go

Impact The delegated energy is refunded to the message signer, violating the intended design.

Suggestion Revise the code accordingly.

Clarification from BlockSec It is worthy noting that the refunded energy for swept and undelegated delegations will be directly discarded in the refunding process.

2.1.13 Violation of the fixed-supply assumption due to the inflation module

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `rewards.go` of the `module x/tokenomics`, the function `GetCirculatingSupply()` calculates the circulating supply based on tokens which are pre-minted to tokenomics pools. However, the calculation circulating supply ignores the inflation mechanism (i.e., the module `x/inflation`), which is enabled by default. As a result, this design violates the fixed-supply assumption.

```
259 func (k Keeper) GetCirculatingSupply(ctx sdk.Context) math.Int {
260   params := k.GetParams(ctx)
261   state := k.GetReleaseState(ctx)
262   return params.MigrationPoolTotal.
263     Add(state.TotalImmediateDistributed).
264     Add(state.TotalMinerReleased).
265     Add(state.TotalProjectReleased)
266 }
```

Listing 2.28: x/tokenomics/keeper/rewards.go

Impact Enabling the inflation module violates the fixed-supply assumption.

Suggestion Disable the inflation module.

2.1.14 Potential fee evasion due to the improper fee calculation in the function `computeShortfallFee()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `decorator.go` of the module `x/energy`, the function `computeShortfallFee()` computes the shortfall fee based on the values of offered and shortfallGas when the value of `offered` is greater than zero. However, when the user offers a very small non-zero fee (e.g., 1 aatos), the numerator `offered * shortfallGas` can be smaller than `gasLimit`, causing the integer division to truncate to zero. As a result, the shortfall fee (i.e., `chargeAtos`) is potentially set to zero, allowing fee evasion.

```
201 func computeShortfallFee(
202   k keeper.Keeper, ctx sdk.Context, shortfallGas uint64, fee sdk.Coins, gasLimit uint64,
203 ) sdk.Coins {
204   if shortfallGas == 0 || gasLimit == 0 {
205     return sdk.NewCoins()
206   }
207   denom := k.BaseDenom()
208   offered := fee.AmountOf(denom)
209   if offered.IsZero() {
210     // User offered no fee at all →fall back to the param gas price
211     // so we still collect SOMETHING (the alternative is letting
212     // users dodge the shortfall by setting tx.fee = 0).
213     params := k.GetParams(ctx)
214     if params.InsufficientGasPrice.IsNil() || !params.InsufficientGasPrice.IsPositive() {
215       return sdk.NewCoins()
216     }
217     amt := params.InsufficientGasPrice.MulInt64(int64(shortfallGas)).Ceil().TruncateInt()
218     return sdk.NewCoins(sdk.NewCoin(denom, amt))
219   }
220   // Pro-rate: charge offered_fee * shortfallGas / gasLimit.
221   num := offered.Mul(math.NewIntFromUint64(shortfallGas))
222   amt := num.Quo(math.NewIntFromUint64(gasLimit))
223   if amt.IsNegative() {
224     amt = math.ZeroInt()
225   }
226   return sdk.NewCoins(sdk.NewCoin(denom, amt))
227 }
```

Listing 2.29: x/energy/ante/decorator.go

```
164 chargeAtos := computeShortfallFee(d.energyKeeper, ctx, consumed.ShortfallGas, stdFee, gasLimit)
165 if chargeAtos.IsZero() {
166   return next(ctx, tx, simulate)
167 }
```

Listing 2.30: x/energy/ante/decorator.go

Impact Users can evade the shortfall gas by providing a minimal fee.

Suggestion Revise the code accordingly.

2.1.15 Ineffective priority due to the use of `NoOpMempool`

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description In the file decorator.go of the module x/energy, the function AnteHandle() computes a transaction priority via the function getTxPriority() for potential ordering. However, the application configures a NoOpMempool, whose Insert() method ignores the calculated priority by discarding the context. As a result, the priority computed by the function AnteHandle() becomes ineffective.

```
187 priority := getTxPriority(stdFee, int64(gasLimit))
188 ctx = ctx.WithPriority(priority)
```

Listing 2.31: x/energy/ante/decorator.go

```
329 baseAppOptions = append(baseAppOptions, func(app *baseapp.BaseApp) {
330   mempool := mempool.NoOpMempool{}
331   app.SetMempool(mempool)
332   handler := baseapp.NewDefaultProposalHandler(mempool, app)
333   app.SetPrepareProposal(handler.PrepareProposalHandler())
334   app.SetProcessProposal(handler.ProcessProposalHandler())
335 })
```

Listing 2.32: app/app.go

Impact The transaction priority mechanism is ineffective under the current mempool configuration.

Suggestion Revise the code accordingly.

2.1.16 Incorrect KV prefix usage in the function GetPriceHistory()

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description In the file keeper.go of the module x/oracle, the function GetPriceHistory() uses an incorrect hardcoded prefix (i.e., use byte(2) instead of byte(3)). As a result, incorrect prices are fetched in the function GetPriceHistory(), leading to potential losses.

```
108 func (k Keeper) GetPriceHistory(ctx sdk.Context, limit uint32) []types.PriceData {
109   store := ctx.KVStore(k.storeKey)
110   iter := storetypes.KVStoreReversePrefixIterator(store, []byte{byte(2)}) // prefixPriceHistory =
          2
111   defer iter.Close()
112
113   var result []types.PriceData
114   count := uint32(0)
115   for ; iter.Valid() && count < limit; iter.Next() {
116     var pd types.PriceData
117     if err := json.Unmarshal(iter.Value(), &pd); err != nil {
118       continue
119     }
120     result = append(result, pd)
121     count++
```

```
122   }
123   return result
124 }
```

Listing 2.33: x/oracle/keeper/keeper.go

```
10 const (
11   prefixParams = iota + 1
12   prefixCurrentPrice
13   prefixPriceHistory
14 )
```

Listing 2.34: x/oracle/types/keys.go

Impact Incorrect prices are fetched in the function `GetPriceHistory()`, leading to potential losses.

Suggestion Use the correct key prefix in the function `GetPriceHistory()`.

2.1.17 Incorrect priority calculation in the function `getTxPriority()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `decorator.go` of the module `x/energy`, the function `AnteHandle()` invokes the function `getTxPriority()` with the fee `stdFee` to compute transaction priority. However, the fee `stdFee` does not reflect the actual cost (i.e., `chargeAtos`). As a result, the incorrect priority is produced and used for transaction ordering.

Moreover, the function `getTxPriority()` iterates the fee coins in the variable `stdFee` but does not verify that the fee coin is ATOS. As a result, the incorrect priority may be produced when other fee coins are introduced in the future.

```
163 // Partial / full ATOS payment: derive the actual amount to charge.
164 chargeAtos := computeShortfallFee(d.energyKeeper, ctx, consumed.ShortfallGas, stdFee, gasLimit)
165 if chargeAtos.IsZero() {
166   return next(ctx, tx, simulate)
167 }
168
169 // Use feegrant if applicable.
170 if !feeGranter.Empty() && !feeGranter.Equals(feePayer) {
171   if d.feegrantKeeper == nil {
172     return ctx, sdkerrors.ErrInvalidRequest.Wrap("fee grants are not enabled")
173   }
174   if err := d.feegrantKeeper.UseGrantedFees(ctx, feeGranter, feePayer, chargeAtos, msgs); err !=
          nil {
175     return ctx, err
176   }
177 }
178
179 // Move the shortfall ATOS from payer to fee_collector.
180 if err := d.bankKeeper.SendCoinsFromAccountToModule(
```

```
181   ctx, deductFrom, authtypes.FeeCollectorName, chargeAtos,
182 ); err != nil {
183   return ctx, sdkerrors.ErrInsufficientFee.Wrapf("shortfall fee transfer: %v", err)
184 }
185
186 // Set tx priority based on the gas price the user offered.
187 priority := getTxPriority(stdFee, int64(gasLimit))
188 ctx = ctx.WithPriority(priority)
189
190 return next(ctx, tx, simulate)
```

Listing 2.35: x/energy/ante/decorator.go

```
232 func getTxPriority(fee sdk.Coins, gas int64) int64 {
233   var priority int64
234   for _, c := range fee {
235     p := int64(0)
236     gasPrice := c.Amount.QuoRaw(gas)
237     if gasPrice.IsInt64() {
238       p = gasPrice.Int64()
239     }
240     if priority == 0 || p < priority {
241       priority = p
242     }
243   }
244   return priority
245 }
```

Listing 2.36: x/energy/ante/decorator.go

Impact The incorrect priority may be produced and used for transaction ordering.

Suggestion Revise the function `getTxPriority()` accordingly.

2.1.18 Lack of validation for `gs.PriceHistory` in the function InitGenesis()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the module `x/oracle`, the function `InitGenesis()` initializes `CurrentPrice` by selecting the last element of `gs.PriceHistory`. However, `GenesisState.Validate()` only validates each `PriceData` entry individually and does not enforce that `PriceHistory` is sorted by timestamp. If the provided historical prices are not properly sorted, the last element may not reflect the proper price, leading to potential losses (e.g., improper token releases).

```
 9 func (k Keeper) InitGenesis(ctx sdk.Context, gs types.GenesisState) {
10   if err := k.SetParams(ctx, gs.Params); err != nil {
11     panic(err)
12   }
13   for _, pd := range gs.PriceHistory {
14     if err := k.AppendPriceHistory(ctx, pd); err != nil {
15       panic(err)
```

```
16      }
17    }
18    if len(gs.PriceHistory) > 0 {
19      latest := gs.PriceHistory[len(gs.PriceHistory)-1]
20      if err := k.SetCurrentPrice(ctx, latest); err != nil {
21        panic(err)
22      }
23    }
24  }
25
26  func (k Keeper) ExportGenesis(ctx sdk.Context) *types.GenesisState {
27    params := k.GetParams(ctx)
28    history := k.GetPriceHistory(ctx, 10000)
29    return &types.GenesisState{
30      Params:     params,
31      PriceHistory: history,
32    }
33  }
```

Listing 2.37: x/oracle/keeper/genesis.go

Impact Potential losses due to the lack of validation for `gs.PriceHistory` in the function `InitGenesis()`.

Suggestion Add proper validation for `gs.PriceHistory` in the function `InitGenesis()`.

2.1.19 Lack of overflow protection in the `TxEnergyCapacity()` and `DeployRecoverPerSecond()` functions

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `helpers.go` of the `module x/energy`, the function `TxEnergyCapacity()` computes the energy capacity using raw uint64 multiplication without overflow protection (i.e., `units.Uint64() * p.TxEnergyPerThreshold`). As a result, the potential multiplication overflows may lead to corrupted capacity accounting, causing affected users to silently lose accrued energy. Moreover, the function `DeployRecoverPerSecond()` (i.e., Line 147) has the same issue.

```
121  cap := units.Uint64() * p.TxEnergyPerThreshold
```

Listing 2.38: x/energy/types/helpers.go

```
147  return (units.Uint64() * p.DeployEnergyCapacity) / denom
```

Listing 2.39: x/energy/types/helpers.go

```
47  txCap := types.TxEnergyCapacity(acct.LastBalanceSnapshot, params)
48  if txCap > 0 && acct.TxEnergyAccrued < txCap && params.TxEnergyMaxAccrueWindow > 0 {
49    add := saturatingMul(elapsed, txCap) / uint64(params.TxEnergyMaxAccrueWindow)
50    newAccrued := saturatingAdd(acct.TxEnergyAccrued, add)
51    if newAccrued > txCap {
52      newAccrued = txCap
```

```
53   }
54   acct.TxEnergyAccrued = newAccrued
55 }
```

Listing 2.40: x/energy/keeper/settle.go

Impact Incorrect accounting due to the multiplication overflow.

Suggestion Add overflow protections in the `TxEnergyCapacity()` and `DeployRecoverPerSecond()` functions.

2.1.20 Potential inconsistent reward accounting when there are no bonded validators

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `rewards.go` of the module `x/tokenomics`, the function `BeginBlocker()` transfers the immediate portion of the block reward to the fee collector before checking whether any bonded validators exist. However, if `totalBonded` is zero, the function returns without persisting `BlockRewardState` or `ReleaseState`. As a result, the bank balance of the fee collector increases while the tokenomics accounting state remains unchanged, causing the two states to diverge.

```
34 immediate := currentReward.MulRaw(int64(params.ImmediateRewardBps)).QuoRaw(10000)
35 locked := currentReward.Sub(immediate)
36
37 if immediate.IsPositive() {
38   coin := sdk.NewCoin(k.baseDenom(), immediate)
39   if err := k.bankKeeper.SendCoinsFromModuleToModule(ctx, tokenomicstypes.MinerPoolName, k.
         feeCollectorName, sdk.NewCoins(coin)); err != nil {
40     return err
41   }
42 }
```

Listing 2.41: x/tokenomics/keeper/rewards.go

```
53 if totalBonded.IsZero() {
54   return nil
```

Listing 2.42: x/tokenomics/keeper/rewards.go

Impact Inconsistent reward accounting and potential over-distribution when `totalBonded` is equal to zero.

Suggestion Move the `totalBonded` check before any reward transfer or tokenomics state update.

2.1.21 Inconsistent implementation of the function `getTxPriority()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `decorator.go` of the module `x/energy`, the function `getTxPriority()` initializes the priority to 0 when `gasPrice` exceeds the int64 range. However, this contradicts the standard SDK `getTxPriority()` implementation, which sets priority to math.MaxInt64 in the same case. As a result, the priority of a transaction with an extremely high `gasPrice` is set to 0, violating the transaction ordering mechanism.

```
232 func getTxPriority(fee sdk.Coins, gas int64) int64 {
233   var priority int64
234   for _, c := range fee {
235     p := int64(0)
236     gasPrice := c.Amount.QuoRaw(gas)
237     if gasPrice.IsInt64() {
238       p = gasPrice.Int64()
239     }
240     if priority == 0 || p < priority {
241       priority = p
242     }
243   }
244   return priority
245 }
```

Listing 2.43: x/energy/ante/decorator.go

Impact The priority of a transaction with an extremely high `gasPrice` is set to 0, violating the transaction ordering mechanism.

Suggestion Set priority to `math.MaxInt64` when `gasPrice` exceeds the int64 range.

2.2 Recommendation

2.2.1 Complete the validity checks in the function `Validate()` of the module `x/tokenomics`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `helpers.go` of the module `x/tokenomics`, the function `Validate()` only validates the `Params` field, while the remaining fields of `GenesisState` (i.e., `ReleaseState`, `BlockRewardState`, `MinerLockedBalances`, and `ProjectClaimable`) are not validated. It is recommended to add proper validation for the remaining fields of `GenesisState` in the function `Validate()`.

```
147 func (gs GenesisState) Validate() error {
148   if err := gs.Params.Validate(); err != nil {
149     return fmt.Errorf("invalid tokenomics params: %w", err)
150   }
```

```
151   return nil
152 }
```

Listing 2.44: x/tokenomics/types/helpers.go

```
29 type GenesisState struct {
30   Params           Params            `protobuf:"bytes,1,opt,name=params,proto3" json:"params"`
31   ReleaseState     ReleaseState      `protobuf:"bytes,2,opt,name=release_state,json=releaseState
         ,proto3" json:"release_state"`
32   BlockRewardState BlockRewardState  `protobuf:"bytes,3,opt,name=block_reward_state,json=
         blockRewardState,proto3" json:"block_reward_state"`
33   MinerLockedBalances []MinerLockedBalance `protobuf:"bytes,4,rep,name=miner_locked_balances,json
         =minerLockedBalances,proto3" json:"miner_locked_balances"`
34   // project_claimable is the math.Int amount of project pool currently claimable.
35   ProjectClaimable cosmossdk_io_math.Int `protobuf:"bytes,5,opt,name=project_claimable,json=
         projectClaimable,proto3,customtype=cosmossdk.io/math.Int" json:"project_claimable"`
36 }
```

Listing 2.45: x/tokenomics/types/genesis.pb.go

Suggestion Complete the validity checks for `GenesisState` in the function `Validate()`.

2.2.2 Avoid panics in block handlers

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `rewards.go` of the module `x/tokenomics`, the function `EndBlocker()` may invoke `ReleaseMinerLockedRewards()`, which persists each validator's updated locked balance. However, the function currently panics if `SetMinerLockedBalance()` fails. A similar pattern exists in `BeginBlocker()` when newly accrued locked rewards are assigned. It is recommended to avoid using `panic()` in blocker handlers due to potential chain halting.

```
73   if share.IsPositive() {
74     bal := k.GetMinerLockedBalance(ctx, validator.GetOperator())
75     bal.LockedAccrued = bal.LockedAccrued.Add(share)
76     if err := k.SetMinerLockedBalance(ctx, bal); err != nil {
77       panic(err)
78     }
79     remainingLocked = remainingLocked.Sub(share)
80   }
```

Listing 2.46: x/tokenomics/keeper/rewards.go

```
246   if share.IsPositive() {
247     bal.LockedClaimable = bal.LockedClaimable.Add(share)
248     if err := k.SetMinerLockedBalance(ctx, bal); err != nil {
249       panic(err)
250     }
251     remainingToAssign = remainingToAssign.Sub(share)
252   }
```

Listing 2.47: x/tokenomics/keeper/rewards.go

Suggestion Refactor the reward accounting functions to return errors instead of panicking.

2.2.3 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several unused statements and functions. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. Redundant `if` blocks.

```
44 if thresholdUnits == 0 {
45   thresholdUnits = 1
46 }
```

Listing 2.48: x/energy/keeper/delegation.go

```
67   if index == 0 {
68     // first validator doesn't get special treatment; keep deterministic subtraction below
69   }
```

Listing 2.49: x/tokenomics/keeper/rewards.go

2. Redundant `switch` blocks.

```
253   switch sdk.MsgTypeURL(m) {
254   case "/cosmwasm.wasm.v1.MsgInstantiateContract",
255     "/cosmwasm.wasm.v1.MsgInstantiateContract2",
256     "/cosmwasm.wasm.v1.MsgStoreCode":
257     return true
258   case "/ethermint.evm.v1.MsgEthereumTx":
259     // EVM deployments are recognized by To == nil. The MonoDecorator
260     // handles them outside this Cosmos chain, so reaching here
261     // means the tx was wrapped as a Cosmos msg -treat as deploy.
262     return true
263   }
```

Listing 2.50: x/energy/ante/decorator.go

3. The function `DeployRecoverPerSecond()` is redundant.

```
130 func DeployRecoverPerSecond(eligibleBalance math.Int, p Params) uint64 {
131   if eligibleBalance.IsNil() || !eligibleBalance.IsPositive() ||
132     p.DeployHoldingThreshold.IsNil() || !p.DeployHoldingThreshold.IsPositive() {
133     return 0
134   }
135   units := eligibleBalance.Quo(p.DeployHoldingThreshold)
136   if units.IsZero() {
137     return 0
138   }
139   // units * capacity / (recover_days * 86400)
140   denom := uint64(p.DeployRecoverDays) * 86_400
141   if denom == 0 {
```

```
142     return 0
143   }
144   if !units.IsUint64() {
145     return ^uint64(0)
146   }
147   return (units.Uint64() * p.DeployEnergyCapacity) / denom
148 }
```

Listing 2.51: x/energy/types/helpers.go

```
144 func deployAddOverElapsed(eligibleBalance math.Int, elapsed uint64, p types.Params) uint64 {
145   if eligibleBalance.IsNil() || !eligibleBalance.IsPositive() ||
146     p.DeployHoldingThreshold.IsNil() || !p.DeployHoldingThreshold.IsPositive() {
147     return 0
148   }
149   units := eligibleBalance.Quo(p.DeployHoldingThreshold)
150   if units.IsZero() || !units.IsUint64() {
151     if !units.IsZero() {
152       return maxU64
153     }
154     return 0
155   }
156   denom := uint64(p.DeployRecoverDays) * 86_400
157   if denom == 0 {
158     return 0
159   }
160   // (units * capacity * elapsed) / denom
161   num := saturatingMul(saturatingMul(units.Uint64(), p.DeployEnergyCapacity), elapsed)
162   return num / denom
163 }
```

Listing 2.52: x/energy/keeper/settle.go

Suggestion Remove the redundant code.

2.2.4 Unify `KVStore` serialization across all custom modules

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the modules `x/oracle` and `x/tokenomics`, all `KVStore` serialization uses the go standard library `encoding/json`, while the module `x/energy` uses the Cosmos SDK protobuf binary codec via `cdc.MustMarshal()` and `cdc.MustUnmarshal()`. It is recommended to unify the serialization using the Cosmos SDK binary codec for consistency.

```
50   if err := json.Unmarshal(bz, &params); err != nil {
51     panic(fmt.Errorf("failed to unmarshal oracle params: %w", err))
52   }
53   return params
54 }
55
56 func (k Keeper) SetParams(ctx sdk.Context, params types.Params) error {
57   if err := params.Validate(); err != nil {
```

```
58      return err
59    }
60    store := ctx.KVStore(k.storeKey)
61    bz, err := json.Marshal(params)
62    if err != nil {
63      return err
64    }
65    store.Set(types.KeyPrefixParams, bz)
66    return nil
67  }
68
69  // --- Price Data ---
70
71  func (k Keeper) SetCurrentPrice(ctx sdk.Context, price types.PriceData) error {
72    store := ctx.KVStore(k.storeKey)
73    bz, err := json.Marshal(price)
74    if err != nil {
75      return err
76    }
77    store.Set(types.KeyPrefixCurrentPrice, bz)
78    return nil
79  }
80
81  func (k Keeper) GetCurrentPrice(ctx sdk.Context) (types.PriceData, error) {
82    store := ctx.KVStore(k.storeKey)
83    bz := store.Get(types.KeyPrefixCurrentPrice)
84    if bz == nil {
85      return types.PriceData{}, types.ErrPriceNotFound
86    }
87    var price types.PriceData
88    if err := json.Unmarshal(bz, &price); err != nil {
89      return types.PriceData{}, err
90    }
91    return price, nil
92  }
93
94  // --- Price History ---
95
96  func (k Keeper) AppendPriceHistory(ctx sdk.Context, price types.PriceData) error {
97    store := ctx.KVStore(k.storeKey)
98    key := types.PriceHistoryKey(price.Timestamp)
99    bz, err := json.Marshal(price)
```

Listing 2.53: x/oracle/keeper/keeper.go

```
72    if err := json.Unmarshal(bz, &params); err != nil {
73      panic(fmt.Errorf("failed to unmarshal tokenomics params: %w", err))
74    }
75    return params
76  }
77
78  func (k Keeper) SetParams(ctx sdk.Context, params tokenomicstypes.Params) error {
79    if err := params.Validate(); err != nil {
```

```
80      return err
81    }
82    store := ctx.KVStore(k.storeKey)
83    bz, err := json.Marshal(params)
84    if err != nil {
85      return err
86    }
87    store.Set(tokenomicstypes.KeyPrefixParams, bz)
88    return nil
89  }
90
91  func (k Keeper) GetReleaseState(ctx sdk.Context) tokenomicstypes.ReleaseState {
92    store := ctx.KVStore(k.storeKey)
93    bz := store.Get(tokenomicstypes.KeyPrefixReleaseState)
94    if bz == nil {
95      return tokenomicstypes.DefaultReleaseState()
96    }
97    var state tokenomicstypes.ReleaseState
98    if err := json.Unmarshal(bz, &state); err != nil {
99      panic(fmt.Errorf("failed to unmarshal release state: %w", err))
100   }
101   return state
102 }
103
104 func (k Keeper) SetReleaseState(ctx sdk.Context, state tokenomicstypes.ReleaseState) error {
105   store := ctx.KVStore(k.storeKey)
106   bz, err := json.Marshal(state)
107   if err != nil {
108     return err
109   }
110   store.Set(tokenomicstypes.KeyPrefixReleaseState, bz)
111   return nil
112 }
113
114 func (k Keeper) GetBlockRewardState(ctx sdk.Context) tokenomicstypes.BlockRewardState {
115   store := ctx.KVStore(k.storeKey)
116   bz := store.Get(tokenomicstypes.KeyPrefixBlockRewardState)
117   if bz == nil {
118     return tokenomicstypes.DefaultBlockRewardState()
119   }
120   var state tokenomicstypes.BlockRewardState
121   if err := json.Unmarshal(bz, &state); err != nil {
122     panic(fmt.Errorf("failed to unmarshal block reward state: %w", err))
123   }
124   return state
125 }
126
127 func (k Keeper) SetBlockRewardState(ctx sdk.Context, state tokenomicstypes.BlockRewardState)
        error {
128   store := ctx.KVStore(k.storeKey)
129   bz, err := json.Marshal(state)
130   if err != nil {
131     return err
```

```
132   }
133   store.Set(tokenomicstypes.KeyPrefixBlockRewardState, bz)
134   return nil
135 }
136
137 func (k Keeper) GetMinerLockedBalance(ctx sdk.Context, valAddr string) tokenomicstypes.
        MinerLockedBalance {
138   store := ctx.KVStore(k.storeKey)
139   bz := store.Get(tokenomicstypes.MinerLockedKey(valAddr))
140   if bz == nil {
141     return tokenomicstypes.NewMinerLockedBalance(valAddr)
142   }
143   var bal tokenomicstypes.MinerLockedBalance
144   if err := json.Unmarshal(bz, &bal); err != nil {
145     panic(fmt.Errorf("failed to unmarshal miner locked balance: %w", err))
146   }
147   return bal
148 }
149
150 func (k Keeper) SetMinerLockedBalance(ctx sdk.Context, bal tokenomicstypes.MinerLockedBalance)
        error {
151   store := ctx.KVStore(k.storeKey)
152   bz, err := json.Marshal(bal)
```

Listing 2.54: x/tokenomics/keeper/keeper.go

```
61   k.cdc.MustUnmarshal(bz, &p)
62   return p
63 }
64
65 func (k Keeper) SetParams(ctx sdk.Context, p types.Params) error {
66   if err := p.Validate(); err != nil {
67     return err
68   }
69   store := ctx.KVStore(k.storeKey)
70   store.Set(types.KeyPrefixParams, k.cdc.MustMarshal(&p))
71   return nil
72 }
73
74 // ===== Account =====
75
76 // GetEnergyAccount returns the stored account or a fresh zero-value one.
77 // The caller is responsible for calling Settle before reading energy.
78 func (k Keeper) GetEnergyAccount(ctx sdk.Context, addr sdk.AccAddress) types.EnergyAccount {
79   store := ctx.KVStore(k.storeKey)
80   bz := store.Get(types.AccountKey(addr.String()))
81   if bz == nil {
82     return types.NewEnergyAccount(addr.String())
83   }
84   var acct types.EnergyAccount
85   k.cdc.MustUnmarshal(bz, &acct)
86   return acct
87 }
```

```
88
89 func (k Keeper) SetEnergyAccount(ctx sdk.Context, acct types.EnergyAccount) {
90   store := ctx.KVStore(k.storeKey)
91   store.Set(types.AccountKey(acct.Address), k.cdc.MustMarshal(&acct))
92 }
93
94 // IterateAccounts walks every stored energy account. Stop when fn returns true.
95 func (k Keeper) IterateAccounts(ctx sdk.Context, fn func(types.EnergyAccount) bool) {
96   store := ctx.KVStore(k.storeKey)
97   it := storetypes.KVStorePrefixIterator(store, types.KeyPrefixAccount)
98   defer it.Close()
99   for ; it.Valid(); it.Next() {
100    var acct types.EnergyAccount
101    k.cdc.MustUnmarshal(it.Value(), &acct)
102    if fn(acct) {
103      return
104    }
105  }
106 }
107
108 // ===== Delegation primary store =====
109
110 func (k Keeper) GetDelegation(ctx sdk.Context, id uint64) (types.EnergyDelegation, bool) {
111   store := ctx.KVStore(k.storeKey)
112   bz := store.Get(types.DelegationKey(id))
113   if bz == nil {
114    return types.EnergyDelegation{}, false
115  }
116  var d types.EnergyDelegation
117  k.cdc.MustUnmarshal(bz, &d)
118  return d, true
119 }
120
121 // setDelegation writes the primary record and refreshes secondary indexes.
122 // Called by Delegate / consume / Undelegate / EndBlocker.
123 func (k Keeper) setDelegation(ctx sdk.Context, d types.EnergyDelegation) {
124   store := ctx.KVStore(k.storeKey)
125   store.Set(types.DelegationKey(d.Id), k.cdc.MustMarshal(&d))
```

Listing 2.55: x/energy/keeper/keeper.go

Suggestion Unify the serialization using the Cosmos SDK binary codec for consistency.

2.3 Note

2.3.1 `TotalMinerLocked` only increases and never decreases

Introduced by `Version 1`

Description In the module `x/tokenomics`, the function `BeginBlocker()` increments the locked portion of the block reward (i.e., `TotalMinerLocked`), but this variable does not decrease regardless of whether the miner reward is released via the function `TriggerRelease()` or claimed

via the function `ClaimMinerLockedReward()`. Currently, no business logic reads this variable. If future code uses this value, the monotonically increasing value will not accurately reflect the actual unlocked balance.

```
46 releaseState.TotalMinerLocked = releaseState.TotalMinerLocked.Add(locked)
```

Listing 2.56: x/tokenomics/keeper/rewards.go

2.3.2 The construction of Merkle proofs for the pre-mine migration

Introduced by `Version 1`

Description In the module `x/tokenomics`, the migration process uses a Merkle root generated from a CSV snapshot to verify pre-mine claims on-chain. The project team should ensure that all CSV fields that participate in leaf construction are checked for consistency with the on-chain encoding and validation logic.

Specifically, amounts should be normalized to the exact canonical decimal string form used by `math.Int.String()` on-chain, and claimer addresses should be validated as proper bech32 account addresses and deduplicated where necessary. Otherwise, malformed or non-canonical snapshot entries may be committed into the Merkle root but become unclaimable during on-chain verification.

2.3.3 Ensure proper price reports

Introduced by `Version 1`

Description In the module `x/oracle`, only the authorized price feeders are allowed to repot prices. The project must ensure all feeders are working properly for correct functionality.

```
23 func (k msgServer) ReportPrice(goCtx context.Context, msg *types.MsgReportPrice) (*types.
       MsgReportPriceResponse, error) {
24   ctx := sdk.UnwrapSDKContext(goCtx)
25   params := k.GetParams(ctx)
26
27   if !params.IsAllowedFeeder(msg.Feeder) {
28     return nil, types.ErrUnauthorizedFeeder
29   }
```

Listing 2.57: x/oracle/keeper/msg_server.go

2.3.4 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., the account `ProjectTreasuryAddress`) can conduct sensitive operations, which introduces potential centralization risks. For example, the account `ProjectTreasuryAddress` is able to withdraw coins from the module account `ProjectPoolName`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS