



# Pingu Exchange

Smart Contract Security Audit

Prepared by ShellBoxes

September 25<sup>th</sup>, 2025 - October 6<sup>th</sup>, 2025

[Shellboxes.com](https://shellboxes.com)

[contact@shellboxes.com](mailto:contact@shellboxes.com)

## Document Properties

Client	Pingu
Version	1.0
Classification	Public

## Scope

Repository	Commit Hash
<a href="https://github.com/OxSuperPingu/pingu-protocol">https://github.com/OxSuperPingu/pingu-protocol</a>	a55a2fb325dee16f1e4e26dccb1271b630773de0

## Re-Audit

Repository	Commit Hash
<a href="https://github.com/OxSuperPingu/pingu-protocol">https://github.com/OxSuperPingu/pingu-protocol</a>	f570e0ecfd91be1b212cfdba2bc5a64a43f9de59

## Contacts

COMPANY	EMAIL
ShellBoxes	<a href="mailto:contact@shellboxes.com">contact@shellboxes.com</a>

# Contents

1	Introduction	5
1.1	About Pingu	5
1.2	Approach & Methodology	5
1.2.1	Risk Methodology	6
2	Findings Overview	7
2.1	Summary	7
2.2	Key Findings	7
3	Finding Details	9
SHB.1	Unchecked Pyth Price Sign Causes Unsigned Wraparound	9
SHB.2	Division By Zero in Loss Streaming When Per Asset Payout Period Is Unset	10
SHB.3	Profit Payout Reverts When Owed Profit Equals Pool Balance	12
SHB.4	Keeper-Set Global UPL Can Freeze or Heavily Tax Withdrawals	13
SHB.5	Liquidations Revert When Fee Exceeds Margin	14
SHB.6	Configured Max Open Interest Not Enforced on Position Increases	15
SHB.7	ETH Deposits Do Not Validate <code>msg.value</code> Against <code>amount</code>	16
SHB.8	Collateral Not Restricted to ETH or USDC	17
SHB.9	Absence of Hard 100× Leverage Cap in Governance Settings	19
SHB.10	Quadratic Gas in <code>Staking.collectMultiple</code>	20
SHB.11	Fee Routing Stops When Any Share Is Zero	21
SHB.12	<code>DataStore</code> Setters Silently No-Op When <code>overwrite == false</code>	22
SHB.13	<code>ReferralStore</code> Writable by Any <code>CONTRACT</code> Role	23
SHB.14	Missing Events for Role and Parameter Changes	24
SHB.15	Staker Fee Share Defaults to 5% Instead of 35%	25
SHB.16	Staker Rewards Paid in Arbitrary Assets Instead of ETH or USDC	26
SHB.17	Unbounded Oracle Price Confidence (Pyth) Permits Low-Quality Executions	27
SHB.18	<code>setGov</code> Allows Zero Address	28
SHB.19	<code>Pool.withdraw</code> Requires Amount Greater Than 10,000 (Unusual Threshold)	29
SHB.20	0(n) Governance Scans Are Acceptable but Could Be Optimized	30
4	Best Practices	32
BP.1	DRY oracle price conversion and assert positivity	32
BP.2	Cache loop bound and use unchecked increment in <code>updateRewards</code>	33

BP.3	Avoid quadratic work in <code>collectMultiple</code> . . . . .	34
BP.4	Enforce ETH deposit value equality in <code>FundStore.transferIn</code> . . . . .	35
BP.5	Emit events for role changes and critical parameter updates . . . . .	36
BP.6	Use custom errors instead of revert strings . . . . .	37
BP.7	Replace repeated <code>keccak256("CONTRACT")</code> with a role constant . . . . .	37
BP.8	Fallback to global payout period before division in <code>Pool</code> . . . . .	38
BP.9	Permit exact-balance profit payouts . . . . .	39
BP.10	Wire Max OI enforcement in submit and execution paths . . . . .	40
BP.11	Avoid premature returns in fee routing . . . . .	40
5	Conclusion . . . . .	42
6	Scope Files . . . . .	43
6.1	Audit . . . . .	43
6.2	Re-Audit . . . . .	44
7	Disclaimer . . . . .	47

# 1 Introduction

Pingu engaged ShellBoxes to conduct a security assessment on the Pingu Exchange beginning on September 25<sup>th</sup>, 2025 and ending October 6<sup>th</sup>, 2025. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

## 1.1 About Pingu

Pingu is a decentralized perpetuals platform where you can trade crypto and forex from your Web3 wallet with up to 100x leverage using ETH or USDC as collateral, provide liquidity to earn real yield as pools capture traders losses plus 45% of fees, and stake PINGU to receive 35% of fees with rewards paid in ETH or USDC.

Issuer	Pingu
Website	<a href="https://pingu.exchange">https://pingu.exchange</a>
Type	Solidity Smart Contract
Documentation	Pingu Exchange Docs
Audit Method	Whitebox

## 1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

## 1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

# 2 Findings Overview

## 2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Pingu Exchange implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

## 2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 5 high-severity, 6 medium-severity, 9 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Unchecked Pyth Price Sign Causes Unsigned Wraparound	HIGH	Fixed
SHB.2. Division By Zero in Loss Streaming When Per Asset Payout Period Is Unset	HIGH	Fixed
SHB.3. Profit Payout Reverts When Owed Profit Equals Pool Balance	HIGH	Fixed
SHB.4. Keeper-Set Global UPL Can Freeze or Heavily Tax Withdrawals	HIGH	Acknowledged
SHB.5. Liquidations Revert When Fee Exceeds Margin	HIGH	Acknowledged
SHB.6. Configured Max Open Interest Not Enforced on Position Increases	MEDIUM	Acknowledged

SHB.7. ETH Deposits Do Not Validate <code>msg.value</code> Against <code>amount</code>	MEDIUM	Fixed
SHB.8. Collateral Not Restricted to ETH or USDC	MEDIUM	Acknowledged
SHB.9. Absence of Hard 100× Leverage Cap in Governance Settings	MEDIUM	Acknowledged
SHB.10. Quadratic Gas in <code>Staking.collectMultiple</code>	MEDIUM	Fixed
SHB.11. Fee Routing Stops When Any Share Is Zero	MEDIUM	Fixed
SHB.12. <code>DataStore</code> Setters Silently No-Op When <code>overwrite == false</code>	INFORMATIONAL	Acknowledged
SHB.13. <code>ReferralStore</code> Writable by Any <code>CONTRACT</code> Role	INFORMATIONAL	Acknowledged
SHB.14. Missing Events for Role and Parameter Changes	INFORMATIONAL	Fixed
SHB.15. Staker Fee Share Defaults to 5% Instead of 35%	INFORMATIONAL	Fixed
SHB.16. Staker Rewards Paid in Arbitrary Assets Instead of ETH or USDC	INFORMATIONAL	Acknowledged
SHB.17. Unbounded Oracle Price Confidence (Pyth) Permits Low-Quality Executions	INFORMATIONAL	Acknowledged
SHB.18. <code>setGov</code> Allows Zero Address	INFORMATIONAL	Fixed
SHB.19. <code>Pool.withdraw</code> Requires Amount Greater Than 10,000 (Unusual Threshold)	INFORMATIONAL	Fixed
SHB.20. $O(n)$ Governance Scans Are Acceptable but Could Be Optimized	INFORMATIONAL	Acknowledged

# 3 Finding Details

## SHB.1 Unchecked Pyth Price Sign Causes Unsigned Wraparound

- Severity: **HIGH**
- Status: Fixed
- Likelihood: 2
- Impact: 3

### Description:

A signed Pyth price is multiplied by a positive base and cast directly to `uint256` without first asserting positivity. If `price < 0`, the cast wraps to an extremely large unsigned value, corrupting execution predicates and PnL/liquidation math.

### Files Affected:

#### SHB.1.1: Unsigned cast from signed Pyth price without positivity check

```
639 /* Processor.sol */
640 PythStructs.Price memory retrievedPrice = pyth.getPriceUnsafe(
    ↪ priceFeedId);
641 uint256 baseConversion = 10 ** uint256(int256(18) + retrievedPrice.expo)
    ↪ ;
642 uint256 price = uint256(retrievedPrice.price * int256(baseConversion));
```

#### SHB.1.2: Same pattern in Positions

```
780 /* Positions.sol */
781 PythStructs.Price memory retrievedPrice = pyth.getPriceUnsafe(
    ↪ priceFeedId);
782 uint256 baseConversion = 10 ** uint256(int256(18) + retrievedPrice.expo)
    ↪ ;
783 uint256 price = uint256(retrievedPrice.price * int256(baseConversion));
```

## Recommendation:

Validate sign before casting and then cast from `int256`:

```
1 // in Processor/Positions _getPythPrice
2 PythStructs.Price memory p = pyth.getPriceUnsafe(priceFeedId);
3 require(p.price > 0, "!pyth-price");
4 uint256 base = 10 ** uint256(int256(18) + p.expo);
5 uint256 price = uint256(int256(p.price)) * base;
```

## Updates

The team has fixed the issue by routing every caller through `PythPriceUtils.toUint256`, which now requires `priceData.price > 0` before casting.

## SHB.2 Division By Zero in Loss Streaming When Per Asset Payout Period Is Unset

- Severity: **HIGH**
- Status: Fixed
- Likelihood: 2
- Impact: 3

## Description:

`creditTraderLoss` divides by a per-asset payout period that defaults to zero if not configured. After the first loss sets `lastPaid`, any subsequent loss on that asset divides by zero and reverts.

## Files Affected:

### SHB.2.1: First loss only sets `lastPaid`

```
234 /* Pool.sol */
235 if (lastPaid[asset] == 0) {
236     lastPaid[asset] = block.timestamp;
```

```
237     return;
238 }
```

### SHB.2.2: Division by per-asset period that may be zero

```
239 /* Pool.sol */
240 uint256 bufferPayoutPeriod = poolStore.getBufferPayoutPeriod(asset);
241 uint256 elapsed = block.timestamp - lastPaid[asset];
242 uint256 toBuffer = (loss * elapsed) / bufferPayoutPeriod;
```

### SHB.2.3: Getter returns mapping default (zero) if unset

```
53 /* PoolStore.sol */
54 function getBufferPayoutPeriod(address asset) external view returns (
    ↪ uint256) {
55     return bufferPayoutPeriods[asset];
56 }
```

## Recommendation:

Fallback to the global period and guard non-zero before division:

```
1 /* Pool.sol */
2 uint256 pp = poolStore.getBufferPayoutPeriod(asset);
3 if (pp == 0) { pp = poolStore.bufferPayoutPeriod(); }
4 require(pp > 0, "!payout-period");
5 uint256 toBuffer = (loss * (block.timestamp - lastPaid[asset])) / pp;
```

## Updates

The team has fixed the issue by adding a fallback to the global payout period and a `require(bufferPayoutPeriod > 0)` guard before division in `creditTraderLoss`.

## SHB.3 Profit Payout Reverts When Owed Profit Equals Pool Balance

- Severity: **HIGH**
- Likelihood: 2
- Status: Fixed
- Impact: 3

### Description:

`debitTraderProfit` and its variant require `diffToPayFromPool < poolBalance`. When a trader's owed profit exactly equals the pool balance (after buffer depletion), the strict inequality causes a revert, blocking settlement.

### Files Affected:

#### SHB.3.1: Strict inequality blocks exact-balance payouts

```
289 /* Pool.sol */
290 require(diffToPayFromPool < poolBalance, "!pool-balance");
```

#### SHB.3.2: Same check in alternative branch

```
329 /* Pool.sol */
330 require(diffToPayFromPool < poolBalance, "!pool-balance");
```

### Recommendation:

Permit equality:

```
1 /* Pool.sol */
2 require(diffToPayFromPool <= poolBalance, "!pool-balance");
```

### Updates

The team has fixed the issue by relaxing the pool-balance check to `diffToPayFromPool <= poolBalance` in both payout paths.

## SHB.4 Keeper-Set Global UPL Can Freeze or Heavily Tax Withdrawals

- Severity: **HIGH**
- Likelihood: 2
- Status: Acknowledged
- Impact: 3

### Description:

Whitelisted keepers can write arbitrary `globalUPLs[asset]`. Withdrawal tax scales with reported UPL and can effectively reach 100% in common paths, making LP withdrawals uneconomical or impossible.

### Files Affected:

#### SHB.4.1: Keeper can set global UPL without sanity bounds

```
152 /* Pool.sol */
153 function setGlobalUPLs(address[] calldata assets, int256[] calldata upls
    ↪ ) external onlyContract {
154     for (uint256 i; i < assets.length; i++) {
155         globalUPLs[assets[i]] = upls[i];
156     }
157 }
```

#### SHB.4.2: Withdrawal tax behavior depends on UPL and balance

```
198 /* Pool.sol */
199 if (amount >= balance) return BPS_DIVIDER; // 100% tax if withdrawing
    ↪ all
200 // ... tax scales with (globalUPLs - bufferBalance) ...
```

### Recommendation:

Clamp and circuit-break:

```

1 /* Pool.sol */
2 int256 cap = int256(balance + bufferBalance) * int256(MAX_UPL_BPS) /
    ↪ int256(BPS_DIVIDER);
3 require(upls[i] >= -cap && upls[i] <= cap, "!upl-bounds");
4 // additionally cap max tax in getWithdrawalTaxBps

```

## Updates

The team states that they “use custom withdrawal fees that are always set, and the maximum allowed fee is capped by MAX\_POOL\_WITHDRAWAL\_FEE in PoolStore.” This does not mitigate the keeper-controlled [globalUPLs](#) that continue to drive the tax calculation, so the original freeze vector remains.

## SHB.5 Liquidations Revert When Fee Exceeds Margin

- Severity: **HIGH**
- Likelihood: 2
- Status: Acknowledged
- Impact: 3

### Description:

During liquidation, the code subtracts the execution fee from `position.margin` without ensuring `fee <= margin`, causing underflow and revert for high-fee/low-margin configurations.

### Files Affected:

#### SHB.5.1: Fee subtraction can underflow

```

441 /* Processor.sol */
442 uint256 fee = _calcExecFee(...);
443 uint256 credit = position.margin - fee; // underflows if fee > margin
444 pool.creditTraderLoss(user, asset, market, credit);

```

## Recommendation:

Guard subtraction or preclude unsafe configs:

```
1 /* Processor.sol */
2 uint256 fee = _calcExecFee(...);
3 require(fee <= position.margin, "!fee>margin");
4 pool.creditTraderLoss(user, asset, market, position.margin - fee);
```

## Updates

The team argues that “in all practical configurations, `position.margin` is always higher than the fee.” The liquidation path still subtracts `fee` from margin without guarding, so a config change or transient state could reintroduce the underflow revert.

## SHB.6 Configured Max Open Interest Not Enforced on Position Increases

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

## Description:

`RiskStore.checkMaxOI` exists but is never invoked by order submission or execution, allowing OI to exceed configured limits.

## Files Affected:

### SHB.6.1: Max OI checker exists but unused by callers

```
188 /* RiskStore.sol */
189 function checkMaxOI(address asset, bytes32 market, uint256 newSize)
    ↪ external view {
```

```
190     require(openInterest[asset][market] + newSize <= maxOI[asset][market  
        ↪ ], "!max-oi");  
191 }
```

## Recommendation:

Invoke at submit and before OI increment:

```
1 /* Orders.sol (submit path) */  
2 riskStore.checkMaxOI(asset, market, size);  
3  
4 /* Positions.sol (execution path before increment) */  
5 riskStore.checkMaxOI(asset, market, sizeDelta);  
6 positionStore.incrementOI(asset, market, sizeDelta);
```

## Updates

The team explains that `checkMaxOI` is called inside `checkMaxDelta`; the code indeed now enforces the OI cap from that helper and both order submission and execution invoke it.

## SHB.7 ETH Deposits Do Not Validate `msg.value` Against amount

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

## Description:

In `FundStore.transferIn`, the native-asset branch returns without asserting `msg.value ↪ == amount`, enabling silent accounting mismatches.

## Files Affected:

### SHB.7.1: Early return for ETH without value equality check

```
26 /* FundStore.sol */
27 function transferIn(address asset, address from, uint256 amount)
    ↪ external payable onlyContract {
28     if (amount == 0 || asset == address(0)) return;
29     IERC20(asset).safeTransferFrom(from, address(this), amount);
30 }
```

## Recommendation:

Enforce equality and event emission:

```
1 /* FundStore.sol */
2 if (asset == address(0)) {
3     require(amount > 0 && msg.value == amount, "ETH:bad-amount");
4     emit Deposit(asset, from, amount);
5     return;
6 }
7 require(msg.value == 0, "ERC20:nonzero-msg.value");
8 IERC20(asset).safeTransferFrom(from, address(this), amount);
```

## Updates

The team has fixed the issue by adding strict `msg.value` equality checks in `FundStore.transferIn`.

## SHB.8 Collateral Not Restricted to ETH or USDC

- Severity: **MEDIUM**
- Status: Acknowledged
- Likelihood: 1
- Impact: 3

## Description:

Any asset with `minSize > 0` is considered supported for deposits and orders. The code does not enforce the specification constraint of ETH/USDC-only collateral.

## Files Affected:

### SHB.8.1: Any asset with `minSize > 0` is "supported"

```
58 /* AssetStore.sol */
59 function isSupported(address asset) external view returns (bool) {
60     return assets[asset].minSize > 0;
61 }
```

### SHB.8.2: Order path only checks min size

```
224 /* Orders.sol */
225 require(assetStore.isSupported(asset), "!asset");
```

## Recommendation:

Allow-list collateral at ingress points:

```
1 /* Orders.sol / Pool.sol */
2 require(asset == address(0) || asset == USDC, "!collateral");
```

## Updates

The team intentionally keeps any asset with a positive min size usable as collateral "because we operate with multiple assets on Monad," leaving `AssetStore.isSupported` unchanged.

## SHB.9 Absence of Hard 100× Leverage Cap in Governance Settings

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

### Description:

Per-market `maxLeverage` is required to be `>= 1` but is not upper-bounded to `100×`, allowing governance to configure riskier markets than specified.

### Files Affected:

#### SHB.9.1: Setter lacks upper bound

```
52 /* MarketStore.sol */
53 require(_maxLeverage >= 1, "!lev");
54 markets[id].maxLeverage = _maxLeverage;
```

### Recommendation:

#### Clamp to spec:

```
1 /* MarketStore.sol */
2 require(_maxLeverage >= 1 && _maxLeverage <= 100, "!lev");
```

### Updates

The team wants to “allow the possibility of setting higher leverage ratios in the future,” and `MarketStore.set` still lacks the `100×` upper bound.

## SHB.10 Quadratic Gas in `Staking.collectMultiple`

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

### Description:

`collectMultiple` loops  $N$  assets, and each call runs `updateRewards` which itself loops over all assets. Total work is  $\mathcal{O}(N^2)$  and can OOG as asset count grows.

### Files Affected:

#### SHB.10.1: Batch calls nestedly recompute rewards

```
86 /* Staking.sol */
87 function collectMultiple(address[] calldata assets) external {
88     for (uint256 i = 0; i < assets.length; i++) {
89         collectReward(assets[i]); // calls updateRewards() internally
90     }
91 }
```

#### SHB.10.2: `updateRewards` iterates all assets

```
110 /* Staking.sol */
111 function updateRewards(address account) public {
112     for (uint256 i = 0; i < assetStore.getAssetCount(); i++) {
113         // ...
114     }
115 }
```

### Recommendation:

Compute once, then zero & transfer:

```
1 /* Staking.sol */
```

```

2 function collectMultiple(address[] calldata assets) external {
3     updateRewards(msg.sender);
4     for (uint256 i = 0; i < assets.length; i++) {
5         _collectSingle(msg.sender, assets[i]);
6     }
7 }

```

## Updates

The team has fixed the issue by calling `updateRewards` once at the start of `collectMultiple` and iterating with `unchecked` increments.

## SHB.11 Fee Routing Stops When Any Share Is Zero

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

### Description:

`creditFee` returns early if any share (keeper, staking, pool) is zero, preventing distribution to the remaining sinks and effectively locking collected fees.

### Files Affected:

#### SHB.11.1: Early returns block downstream routing

```

678 /* Positions.sol */
679 if (positionStore.keeperFeeShare() == 0) return;
680 if (stakingStore.feeShare() == 0) return;
681 if (poolStore.feeShare() == 0) return;

```

## Recommendation:

Remove early returns; compute zero shares as zero allocations and continue routing to other sinks.

## Updates

The team has fixed the issue by removing early returns and computing zero shares as zero allocations in `creditFee`.

## SHB.12 DataStore Setters Silently No-Op When `overwrite == false`

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Acknowledged
- Impact: 3

## Description:

`setUint/setInt/setAddress` return `false` instead of reverting or emitting when a key exists and `overwrite == false`, creating configuration ambiguity.

## Files Affected:

### SHB.12.1: Silent no-op on existing key with `overwrite=false`

```
25 /* DataStore.sol */
26 function setUint(bytes32 k, uint256 v, bool overwrite) external returns(
    ↪ bool) {
27     if (!overwrite && uints[k] != 0) return false;
28     uints[k] = v;
29     return true;
30 }
```

## Recommendation:

Emit events and/or revert on blocked updates:

```
1 /* DataStore.sol */
2 if (!overwrite && exists(k)) revert KeyExists();
3 emit UintSet(k, uints[k], v);
```

## Updates

The team states that datastore writes “are rare and are performed carefully by governance scripts,” so the setters continue to silently return false when blocked.

## SHB.13 ReferralStore Writable by Any CONTRACT Role

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Acknowledged
- Impact: 3

## Description:

ReferralStore mutation functions are `onlyContract` without verifying the specific controller, allowing any address with the generic `CONTRACT` role to bypass validations implemented in `Referral.sol`.

## Files Affected:

### SHB.13.1: Broad `onlyContract` gate on writes

```
72 /* ReferralStore.sol */
73 function registerReferral(address user, bytes32 code) external
    ↪ onlyContract { ... }
74 function setReferrer(address user, address referrer) external
    ↪ onlyContract { ... }
```

## Recommendation:

Restrict to a designated controller:

```
1 /* ReferralStore.sol */
2 address public referralController;
3 function setReferralController(address c) external onlyGov {
    ↪ referralController = c; }
4 modifier onlyReferralController() { require(msg.sender ==
    ↪ referralController, "!ctrl"); _; }
```

## Updates

The team says the broad `onlyContract` pattern “is consistent across all stores,” yet `ReferralStore` remains writable by any `CONTRACT`-role address, leaving the original privilege-bypass risk.

## SHB.14 Missing Events for Role and Parameter Changes

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Fixed
- Impact: 3

### Description:

Critical governance mutations (role grants/revokes, fee shares, intervals, asset list updates) lack event emission, hindering monitoring and forensics.

### Files Affected:

#### SHB.14.1: Role changes emit no events

```
27 /* RoleStore.sol */
28 function grantRole(bytes32 role, address account) external onlyGov {
    ↪ roles[role].add(account); }
```

```
29 function revokeRole(bytes32 role, address account) external onlyGov {  
    ↪ roles[role].remove(account); }
```

## Recommendation:

Add and emit events for all governance state changes:

```
1 /* RoleStore.sol */  
2 event RoleGranted(bytes32 role, address account, address sender);  
3 event RoleRevoked(bytes32 role, address account, address sender);
```

## Updates

Partially addressed. The team added role grant/revoke events, but other governance setters still emit nothing, so the wider “parameter change” observability gap persists.

## SHB.15 Staker Fee Share Defaults to 5% Instead of 35%

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Fixed
- Impact: 3

## Description:

StakingStore initializes `feeShare` to 500 bps (5%), diverging from the stated 35%.

## Files Affected:

### SHB.15.1: Default share is 5%

```
13 /* StakingStore.sol */  
14 uint256 public feeShare = 500;
```

## Recommendation:

Initialize to 3500 and optionally bound in setter:

```
1 /* StakingStore.sol */
2 uint256 public feeShare = 3500;
3 function setFeeShare(uint256 bps) external onlyGov { require(bps <=
    ↪ 3500, "!cap"); feeShare = bps; }
```

## Updates

The team has fixed the issue by initializing the staker fee share to **3500** (35%) directly in **StakingStore**.

## SHB.16 Staker Rewards Paid in Arbitrary Assets Instead of ETH or USDC

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Acknowledged
- Impact: 3

## Description:

Staking accrues and pays rewards for all supported assets; no restriction enforces ETH/USDC only payouts.

## Files Affected:

### SHB.16.1: Accrual iterates all supported assets

```
110 /* Staking.sol */
111 for (uint256 i = 0; i < assetStore.getAssetCount(); i++) {
112     // per-asset rewardPerToken logic...
113 }
```

## SHB.16.2: Payout uses the asset passed in

```
103 /* Staking.sol */
104 FundStore.transferOut(asset, user, rewardToSend);
```

### Recommendation:

Enforce allow-list at accrual/payout:

```
1 /* Staking.sol */
2 require(asset == address(0) || asset == USDC, "!reward-asset");
```

### Updates

Echoing SHB.8, the team keeps multi-asset reward payouts; `_collectReward` still transfers whatever asset is supplied.

## SHB.17 Unbounded Oracle Price Confidence (Pyth) Permits Low-Quality Executions

- Severity: **INFORMATIONAL**
- Status: Acknowledged
- Likelihood: 0
- Impact: 3

### Description:

Confidence from Pyth is read/emitted but not bounded; highly uncertain prices can pass age checks and drive execution.

### Files Affected:

#### SHB.17.1: Confidence obtained but not bounded

```
648 /* Processor.sol */
649 uint256 confidence = uint256(uint64(retrievedPrice.conf));
```

```
650 emit PriceUsed(price, publishTime, confidence);
```

## Recommendation:

Gate by market-configured maximum:

```
1 /* Processor.sol after fetching price */
2 require(confidence <= market.maxPythConfidence, "!pyth-conf");
```

## Updates

The team cites reliance on “high confidence price feeds” and concern over small variations; there is still no explicit confidence threshold per market.

## SHB.18 setGov Allows Zero Address

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Fixed
- Impact: 3

## Description:

`_setGov` lacks a non-zero check, enabling accidental bricking of governance.

## Files Affected:

### SHB.18.1: No non-zero guard

```
24 /* Governable.sol */
25 function _setGov(address _gov) internal {
26     address prevGov = gov;
27     gov = _gov;
28     emit SetGov(prevGov, _gov);
29 }
```

## Recommendation:

Require non-zero:

```
1 /* Governable.sol */
2 require(_gov != address(0), "!zero-gov");
```

## Updates

The team has fixed the issue by rejecting zero-address governance updates in `_setGov`.

## SHB.19 Pool.withdraw Requires Amount Greater Than 10,000 (Unusual Threshold)

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Fixed
- Impact: 2

## Description:

Withdrawals require `amount > BPS_DIVIDER (10,000)`, preventing small withdrawals; confirm intended unit.

## Files Affected:

SHB.19.1: High minimum withdraw amount

```
415 /* Pool.sol */
416 require(amount > BPS_DIVIDER, "!amount");
```

## Recommendation:

If unintended, relax:

```
1 /* Pool.sol */
2 require(amount > 0, "!amount");
```

## Updates

The team has fixed the issue by requiring only `amount > 0` for withdrawals.

## SHB.20 O(n) Governance Scans Are Acceptable but Could Be Optimized

- Severity: **INFORMATIONAL**
- Likelihood: 0
- Status: Acknowledged
- Impact: 2

### Description:

Governance-only setters scan arrays linearly to maintain uniqueness; gas grows with list size but does not impact user flows.

### Files Affected:

#### SHB.20.1: Linear scan to maintain asset uniqueness

```
29 /* AssetStore.sol */
30 for (uint256 i; i < assetList.length; i++) {
31     if (assetList[i] == asset) { found = true; break; }
32 }
```

#### SHB.20.2: Market uniqueness check via loop

```
66 /* MarketStore.sol */
67 for (uint256 i; i < marketIds.length; i++) {
68     if (marketIds[i] == id) { exists = true; break; }
69 }
```

### Recommendation:

Optional: maintain an `isInList` mapping to avoid  $O(n)$  scans in governance paths.

## Updates

The team acknowledges the linear scans in governance-only flows and deems them acceptable for now.

# 4 Best Practices

## BP.1 DRY oracle price conversion and assert positivity

### Description:

Both `Processor` and `Positions` duplicate the Pyth price conversion logic and cast a signed price directly to `uint256`. Centralizing this logic avoids bytecode/maintenance drift and enables a single positivity assertion before casting.

### Files Affected:

#### BP.1.1: Processor.sol: duplicate conversion with unsigned cast

```
639 PythStructs.Price memory retrievedPrice = pyth.getPriceUnsafe(  
    ↪ priceFeedId);  
640 uint256 baseConversion = 10 ** uint256(int256(18) + retrievedPrice.expo)  
    ↪ ;  
641 uint256 price = uint256(retrievedPrice.price * int256(baseConversion));
```

#### BP.1.2: Positions.sol: same conversion duplicated

```
780 PythStructs.Price memory retrievedPrice = pyth.getPriceUnsafe(  
    ↪ priceFeedId);  
781 uint256 baseConversion = 10 ** uint256(int256(18) + retrievedPrice.expo)  
    ↪ ;  
782 uint256 price = uint256(retrievedPrice.price * int256(baseConversion));
```

### Recommendation:

Create one internal helper (in a shared base or library) that asserts sign and returns 18 decimals; call it from both contracts.

```
1 // OracleBase.sol
```

```

2 function _toPrice18(PythStructs.Price memory p) internal pure returns (
    ↪ uint256) {
3 require(p.price > 0, "!pyth-price");
4 uint256 base = 10 ** uint256(int256(18) + p.expo);
5 return uint256(int256(p.price)) * base;
6 }
7
8 // Processor.sol / Positions.sol
9 PythStructs.Price memory p = pyth.getPriceUnsafe(priceFeedId);
10 uint256 price = _toPrice18(p);

```

## BP.2 Cache loop bound and use unchecked increment in updateRewards

### Description:

updateRewards calls `assetStore.getAssetCount()` on every iteration and uses a checked increment. Caching the bound and using unchecked saves SLOADs and arithmetic checks in a hot path.

### Files Affected:

BP.2.1: Staking.sol: repeated `getAssetCount()` and checked `++i`

```

110 function updateRewards(address account) public {
111 for (uint256 i = 0; i < assetStore.getAssetCount(); i++) {
112 // ...
113 }
114 }

```

### Recommendation:

Cache the count and use an unchecked increment (the loop bound protects overflow).

```

1 // Staking.sol
2 function updateRewards(address account) public {
3     uint256 n = assetStore.getAssetCount();
4     for (uint256 i; i < n;) {
5         // ...
6         unchecked { ++i; }
7     }
8 }

```

## BP.3 Avoid quadratic work in collectMultiple

### Description:

collectMultiple calls collectReward per asset, and each call internally invokes updateRewards, which itself loops all assets. This induces  $O(N^2)$  work.

### Files Affected:

#### BP.3.1: Staking.sol: nested recomputation across assets

```

86 function collectMultiple(address[] calldata assets) external {
87     for (uint256 i = 0; i < assets.length; i++) {
88         collectReward(assets[i]); // calls updateRewards() internally
89     }
90 }

```

### Recommendation:

Invoke updateRewards once, then zero and transfer per asset.

```

1 // Staking.sol
2 function collectMultiple(address[] calldata assets) external {
3     updateRewards(msg.sender);
4     for (uint256 i; i < assets.length; ) {
5         _collectSingle(msg.sender, assets[i]);

```

```
6 unchecked { ++i; }
7 }
8 }
```

## BP.4 Enforce ETH deposit value equality in FundStore.transferIn

### Description:

The ETH branch returns early without asserting `msg.value == amount`, allowing accounting drift and complicating invariants across callers.

### Files Affected:

#### BP.4.1: FundStore.sol: ETH path lacks value equality check

```
26 function transferIn(address asset, address from, uint256 amount)
    ↪ external payable onlyContract {
27   if (amount == 0 || asset == address(0)) return;
28   IERC20(asset).safeTransferFrom(from, address(this), amount);
29 }
```

### Recommendation:

Require exact value match for ETH and forbid stray `msg.value` for ERC 20.

```
1 // FundStore.sol
2 if (asset == address(0)) {
3   require(amount > 0 && msg.value == amount, "ETH:bad-amount");
4   emit Deposit(asset, from, amount);
5   return;
6 }
7 require(msg.value == 0, "ERC20:nonzero-msg.value");
8 IERC20(asset).safeTransferFrom(from, address(this), amount);
```

## BP.5 Emit events for role changes and critical parameter updates

### Description:

Governance mutations lack events, reducing observability and complicating off-chain indexing and audits.

### Files Affected:

#### BP.5.1: RoleStore.sol: no events on role changes

```
27 function grantRole(bytes32 role, address account) external onlyGov {
28   roles[role].add(account);
29 }
30 function revokeRole(bytes32 role, address account) external onlyGov {
31   roles[role].remove(account);
32 }
```

### Recommendation:

Add and emit events for grants/revokes (and similarly for fee/interval setters elsewhere).

```
1 // RoleStore.sol
2 event RoleGranted(bytes32 role, address account, address sender);
3 event RoleRevoked(bytes32 role, address account, address sender);
4
5 function grantRole(bytes32 role, address account) external onlyGov {
6   roles[role].add(account);
7   emit RoleGranted(role, account, msg.sender);
8 }
9 function revokeRole(bytes32 role, address account) external onlyGov {
10  roles[role].remove(account);
11  emit RoleRevoked(role, account, msg.sender);
12 }
```

## BP.6 Use custom errors instead of revert strings

### Description:

String reverts allocate memory and bloat bytecode. Custom errors are cheaper and standardize failure modes.

### Files Affected:

#### BP.6.1: FundStore.sol: string-based revert on ETH send

```
43 if (asset == address(0)) {
44   (bool success, ) = payable(to).call{value: amount}("");
45   require(success, "Transfer failed");
46 } else {
47   IERC20(asset).safeTransfer(to, amount);
48 }
```

### Recommendation:

Define custom errors and use them in place of strings.

```
1 // FundStore.sol
2 error EthTransferFailed();
3
4 (bool ok, ) = payable(to).call{value: amount}("");
5 if (!ok) revert EthTransferFailed();
```

## BP.7 Replace repeated keccak256("CONTRACT") with a role constant

### Description:

Hashing the same role identifier inline increases bytecode and risks drift across call sites. A single `bytes32` constant is clearer and cheaper.

## Files Affected:

### BP.7.1: Roles.sol: inline role hash each check

```
15 modifier onlyContract() {
16   require(roleStore.hasRole(msg.sender, keccak256("CONTRACT")), "!contract
    ↪ ");
17   _;
18 }
```

## Recommendation:

Introduce a file level constant and reuse it.

```
1 // Roles.sol
2 bytes32 internal constant CONTRACT_ROLE = keccak256("CONTRACT");
3
4 modifier onlyContract() {
5   require(roleStore.hasRole(msg.sender, CONTRACT_ROLE), "!contract");
6   _;
7 }
```

## BP.8 Fallback to global payout period before division in Pool

### Description:

Per asset payout period can be unset (zero). Using it as a divisor without fallback risks division by zero; handling the default improves robustness.

## Files Affected:

### BP.8.1: Pool.sol: division by per-asset period that may be zero

```
239 uint256 bufferPayoutPeriod = poolStore.getBufferPayoutPeriod(asset);
240 uint256 elapsed = block.timestamp - lastPaid[asset];
```

```
241 uint256 toBuffer = (loss * elapsed) / bufferPayoutPeriod;
```

### Recommendation:

Fallback to the global period and assert non zero before division.

```
1 // Pool.sol
2 uint256 pp = poolStore.getBufferPayoutPeriod(asset);
3 if (pp == 0) { pp = poolStore.bufferPayoutPeriod(); }
4 require(pp > 0, "!payout-period");
5 uint256 toBuffer = (loss * (block.timestamp - lastPaid[asset])) / pp;
```

## BP.9 Permit exact-balance profit payouts

### Description:

A strict < check blocks cases where owed profit equals the pool balance, causing unnecessary reverts; allowing equality is safe and avoids settlement failures.

### Files Affected:

#### BP.9.1: Pool.sol: strict inequality on pool balance

```
289 require(diffToPayFromPool < poolBalance, "!pool-balance");
```

### Recommendation:

Allow equality.

```
1 // Pool.sol
2 require(diffToPayFromPool <= poolBalance, "!pool-balance");
```

## BP.10 Wire Max OI enforcement in submit and execution paths

### Description:

The `RiskStore.checkMaxOI` guard exists but is not invoked by `Orders/Positions`, weakening architectural guarantees. Hooking it in aligns code with configured limits.

### Files Affected:

#### BP.10.1: RiskStore.sol: available but unused OI guard

```
188 function checkMaxOI(address asset, bytes32 market, uint256 newSize)
    ↪ external view {
189     require(openInterest[asset][market] + newSize <= maxOI[asset][market],
        ↪ "!max-oi");
190 }
```

### Recommendation:

Invoke at order submission and before OI increment.

```
1 // Orders.sol (submit)
2 riskStore.checkMaxOI(asset, market, size);
3
4 // Positions.sol (execution)
5 riskStore.checkMaxOI(asset, market, sizeDelta);
6 positionStore.incrementOI(asset, market, sizeDelta);
```

## BP.11 Avoid premature returns in fee routing

### Description:

Early returns when any fee share is zero prevent distribution to remaining sinks. Removing these returns simplifies control flow and avoids locked fees.

## Files Affected:

### BP.11.1: Positions.sol: early returns block routing

```
678 if (positionStore.keeperFeeShare() == 0) return;
679 if (stakingStore.feeShare() == 0) return;
680 if (poolStore.feeShare() == 0) return;
```

## Recommendation:

Proceed with zero amounts for zero shares; route the rest.

```
1 // Positions.sol
2 uint256 keeperShare = positionStore.keeperFeeShare();
3 uint256 stakeShare = stakingStore.feeShare();
4 uint256 poolShare = poolStore.feeShare();
5 // compute allocations; any zero simply yields zero transfer for that
  ↪ sink
```

# 5 Conclusion

In this audit, we examined the design and implementation of Pingu Exchange contract and discovered several issues of varying severity. Pingu team addressed 9 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Pingu Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

# 6 Scope Files

## 6.1 Audit

Files	MD5 Hash
Pingu/AssetStore.sol	b50c210e15f7d9974d8aeed89ec1a465
Pingu/Context.sol	313fcc99f1338da53d498e8ad7228ece
Pingu/DataStore.sol	d47c7878fca57a4e974f3643b72a0eca
Pingu/EnumerableSet.sol	bd1bcd216a9c5cb3c0a17a09e4c9c7ae
Pingu/ERC20.sol	ac61f64b139cf5fa08b22258397d6692
Pingu/Funding.sol	93857e7e3f3b251bbaae040aba4827f4
Pingu/FundingStore.sol	d79b59a3c232bb819dd23016be83a4f1
Pingu/FundingUpdater.sol	d376e082fb778f03e4d831f7c4cd0e58
Pingu/FundStore.sol	b42d321816e987eb0b10de3556bb0a0c
Pingu/Governable.sol	e8e10a2f2cc43dd4d0d02623cb345552
Pingu/IERC20.sol	f42c16e907c13674a1ec9611e6da3a20
Pingu/IERC20Metadata.sol	a2186a2141c6123b6d6dfdb4580944e2
Pingu/MarketStore.sol	158e3f3d37fe53a47affd5f0465b37e7
Pingu/Orders.sol	eb7c2027cfde7eed96891ecf1649eab5
Pingu/OrderStore.sol	570fb50be34c79e754e205560ba9ae8e
Pingu/Ownable.sol	0af4517bed5561349ebd746f52793232
Pingu/Pingu.sol	8541832a3593bfa0ce76f9d1cf211d1c

Pingu/Pool.sol	c424af2ae47e1b9f3e3b78d086772c05
Pingu/PoolStore.sol	517d64764af008db617db4d8a801b1c5
Pingu/Positions.sol	7db137425710ad527a2abb029597529d
Pingu/PositionStore.sol	5a7f4fc31c10e1234c01eb1345149860
Pingu/Processor.sol	edc919bee4baa2d7d34a11cb911b6255
Pingu/Proxy.sol	c92cc104d6fb75b653d410392d018dfb
Pingu/PythUpdater.sol	2e27aeacead1408d099704c774e864a2
Pingu/Rebate.sol	5346d18709d07ea055dc34aef7376e86
Pingu/RebateStore.sol	e06a1d9e5358847620fbb4f23c1f0629
Pingu/Referral.sol	7854f3e4e53b1734a24fc5c795ef2123
Pingu/ReferralStore.sol	44323fb039a6326aeb81a10ff83a4c61
Pingu/RiskStore.sol	6f0718ea3508ff88b95ac08ea0a11d48
Pingu/Roles.sol	50b49286040b7006e182633290497aea
Pingu/RoleStore.sol	7b9568b59366861811a203b4aa8b8d91
Pingu/Signable.sol	fe5fedb0eae779ae7542856f24c5521d
Pingu/Staking.sol	47a99b83122e84628b063f0b3d3dcaa6
Pingu/StakingStore.sol	df5e436a3591283d3ba98381e0613a45

## 6.2 Re-Audit

Files	MD5 Hash
-------	----------

./Roles.sol	86c38d19b5639dd2c6fe2089dc9a3689
./Rebate.sol	5346d18709d07ea055dc34aef7376e86
./ReferralStore.sol	ceb0cfe2aaa6f29e5e7533781e7dcfa4
./IERC20.sol	f42c16e907c13674a1ec9611e6da3a20
./AssetStore.sol	b50c210e15f7d9974d8aeed89ec1a465
./FundingStore.sol	d79b59a3c232bb819dd23016be83a4f1
./ERC20.sol	ac61f64b139cf5fa08b22258397d6692
./RiskStore.sol	6f0718ea3508ff88b95ac08ea0a11d48
./PythPriceUtils.sol	bee69315bafd041aa87602d3217467d2
./Pool.sol	d992d4155108a7f229b8e80768b4ae30
./Ownable.sol	0af4517bed5561349ebd746f52793232
./Positions.sol	6b5219c092033387bb69f2a9908edc91
./Signable.sol	571a1ec30a3a90fe0e88643979b2b9ec
./RoleStore.sol	c674cc6f02ccc0bc936e2e5c01f11b15
./Referral.sol	de70c312287b4af47b1fdef89a345423
./DataStore.sol	d47c7878fca57a4e974f3643b72a0eca
./RebateStore.sol	e06a1d9e5358847620fbb4f23c1f0629
./Governable.sol	fbb72c03c59da618e54b66e487c54453
./Proxy.sol	c92cc104d6fb75b653d410392d018dfb
./FundStore.sol	eb0561766095a5b496a3636531124e5c
./EnumerableSet.sol	bd1bcd216a9c5cb3c0a17a09e4c9c7ae

./IERC20Metadata.sol	a2186a2141c6123b6d6dfdb4580944e2
./PythUpdater.sol	2e27aeacead1408d099704c774e864a2
./MarketStore.sol	158e3f3d37fe53a47affd5f0465b37e7
./Processor.sol	76e8ab426596132351c4d108c8b9ef8e
./Context.sol	313fcc99f1338da53d498e8ad7228ece
./StakingStore.sol	43c59c419909980c4b1cdc66366d02de
./Pingu.sol	8541832a3593bfa0ce76f9d1cf211d1c
./Orders.sol	eb7c2027cfde7eed96891ecf1649eab5
./Funding.sol	cacb661ad49003efd28e46c34060cf7e
./PositionStore.sol	5a7f4fc31c10e1234c01eb1345149860
./OrderStore.sol	570fb50be34c79e754e205560ba9ae8e
./PoolStore.sol	517d64764af008db617db4d8a801b1c5
./FundingUpdater.sol	d376e082fb778f03e4d831f7c4cd0e58
./Staking.sol	88f2ae424dda5b7bf014740d24102452

# 7 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.



For a Contract Audit, contact us at [contact@shellboxes.com](mailto:contact@shellboxes.com)