NUTS PIKE SECURITY AUDIT REPORT

September 18, 2025

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals

- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

The Pike protocol is a Compound fork that adds an efficiency mode to optimize collateral usage and employs a diamond proxy architecture for upgradeable contracts. Its core contracts include a `RiskEngineModule` (equivalent to `Comptroller`) for managing collateral and liquidity thresholds, a `PTokenModule` (equivalent to `CToken`) for handling user deposits/borrows, and a `DoubleJumpRateModel` that introduces a double-kink interest rate curve. The system integrates an `OracleEngine` for on-chain price data, supporting two oracles (a main and a fallback), with Chainlink and Pyth as the currently supported providers.

1.4 Project Dashboard

Project Summary

Title	Description
Client	NUTS Finance
Project name	Pike
Timeline	24.01.2025-04.09.2025
Number of Auditors	3

Project Log

Date	Commit Hash	Note
24.01.2025	1c1324f682986d2b2534aae0e3ac0fae82aef4e3	Initial commit
11.03.2025	8eec330761fce3ea3908820ad266901fddf39839	Re-audit commit
27.03.2025	2a1018c8924b126df71f5f0cb72acd4f5ec4bac2	Commit with updates
16.04.2025	01d22d3f48a7367125dd75cc758b24defbdcd845	Commit with support for composite oracles
05.05.2025	cb343179e874e8e9d3ad22fbb50580b575821c34	Commit with composite oracle fix
06.06.2025	a9404fb4f388032babf38e262967b8897a88dbcd	Commit for the re-audit
04.09.2025	663bb920bcb4bbf6ee2948b9851604d4d2aa6d10	Commit for the re-audit

Project Scope

The audit covered the following files:

File name	Link
src/Factory.sol	Factory.sol
src/oracles/ChainlinkOracleProvider.sol	ChainlinkOracleProvider.sol
src/oracles/OracleEngine.sol	OracleEngine.sol
src/oracles/PythOracleProvider.sol	PythOracleProvider.sol
src/governance/Timelock.sol	Timelock.sol
src/pike-market/utils/RBACMixin.sol	RBACMixin.sol
src/pike-market/utils/ExponentialNoError.sol	ExponentialNoError.sol
src/pike-market/utils/OwnableMixin.sol	OwnableMixin.sol
src/pike-market/storage/DoubleJumpRateStorage.sol	DoubleJumpRateStorage.sol
src/pike-market/storage/UpgradeStorage.sol	UpgradeStorage.sol
src/pike-market/storage/RiskEngineStorage.sol	RiskEngineStorage.sol
src/pike-market/storage/RBACStorage.sol	RBACStorage.sol
src/pike-market/storage/PTokenStorage.sol	PTokenStorage.sol
src/pike-market/storage/OwnableStorage.sol	OwnableStorage.sol
src/pike-market/errors/CommonError.sol	CommonError.sol
src/pike-market/errors/RiskEngineError.sol	RiskEngineError.sol
src/pike-market/errors/IRMError.sol	IRMError.sol
src/pike-market/errors/PTokenError.sol	PTokenError.sol
src/pike-market/modules/riskEngine/RiskEngineModule.sol	RiskEngineModule.sol

File name	Link
src/pike-market/modules/InitialModuleBeacon.sol	InitialModuleBeacon.sol
src/pike-market/modules/InitialModuleBundle.sol	InitialModuleBundle.sol
src/pike-market/modules/pToken/PTokenModule.sol	PTokenModule.sol
src/pike-market/modules/common/RBACModule.sol	RBACModule.sol
src/pike-market/modules/common/OwnableModule.sol	OwnableModule.sol
src/pike-market/modules/common/UpgradeModule.sol	UpgradeModule.sol
src/pike-market/modules/interestRateModel/DoubleJumpRateModel.sol	DoubleJumpRateModel.sol
src/oracles/ChainlinkOracleComposite.sol	ChainlinkOracleComposite.sol

Deployments

Sonic: mainnet

File name	Contract deployed on mainnet	Comment
ERC1967Proxy.sol	0xcB53d44FF0b466daebCcE311A8F7bB1DF569AceD	
Factory.sol	0xe8F5164fA1B59587003744E3b96738a4248530B1	
BeaconProxy.sol	0x069917BfCda2411B74b8adc228de016dAd8Cbd12	OracleEngine Proxy
OracleEngine.sol	0x65D6c833e36B5C47f194A68C4bA138b4019D17a0	
BeaconProxy.sol	0x46f5AEB237219A08d7eD6B71aE5518253EEDa952	RiskEngine Proxy

File name	Contract deployed on mainnet	Comment
RiskEngineModule.sol	0x0dF841a828eaa095BeD77BBe75c7eae2438dD28b	
BeaconProxy.sol	0x13E795d0cb62E9E23116F10B27E029ef447F4801	Timelock Proxy
Timelock.sol	0xa008cB665F7dc3024F58B59d69cc0b003b7caf02	
InitialModuleBeacon.sol	0x5e8A804089619FfC417a6814728082Ddd1E7BE88	
DoubleJumpRateModel.sol	0x52ca28785ADDD5aDb8B9eFf0ec100f3BBA18AA5A	
RBACModule.sol	0xE6923499ddbb333E3823656DE263517e18702381	
PTokenModule.sol	0xc7DbBC3304286489721308ed60b5D5EA62743E79	
BeaconProxy.sol	0xDcE5e506086E510d0f30Be0A37e5CdfbD305d725	psts market
BeaconProxy.sol	0x9A2d3d3B45496049290EA8C2Fa57aE8DB888bB99	pws market
BeaconProxy.sol	0xbb2b89a15c50df1b997155284CC135A1FE8C01Ee	pwspa_ws_sts market

1.5 Summary of findings

Severity	# of Findings
Critical	1
High	1
Medium	3
Low	12

ID	Name	Severity	Status
C-1	Inflation Attack in `PTokenModule.deposit` Draining Initial Deposits	Critical	Fixed
H-1	Arithmetic Overflow in `getPrice` When Feeds Return Large Values	High	Fixed
M-1	Improper Validation of Price Feed Data	Medium	Fixed
M-2	E-Mode Misconfiguration Causing Inaccurate Collateral Accounting	Medium	Fixed
M-3	Excessively High Borrow Rate Processed as Zero	Medium	Fixed
L-1	Oracle Ownership Vulnerability During Deployment	Low	Fixed
L-2	Missing Zero-Address Checks in Key Functions	Low	Fixed
L-3	Missing Range Validation for Key Parameters	Low	Fixed
L-4	RiskEngine Error Checks Using Numerical Constant Instead of `NO_ERROR`	Low	Fixed
L-5	Misleading or Incorrect Comments	Low	Fixed

L-6	Individually Changeable `RiskEngine` per `pToken` Leading to Inconsistency	Low	Fixed
L-7	Inefficient Struct Usage in `ExponentialNoError` Library	Low	Fixed
L-8	Use `require` Instead of `if (condition) revert`	Low	Fixed
L-9	Confusing Error Handling in `mintFresh` and `redeemFresh` Functions	Low	Fixed
L-10	Risk of Storage Slot Overlap Leading to Unpredictable Behavior	Low	Fixed
L-11	Missing EIP-165 Interface Compliance Checks	Low	Acknowledged
L-12	Undescriptive Errors on Arithmetic Underflow	Low	Acknowledged

1.6 Conclusion

We conducted an audit of the protocol by reviewing its code, and examined the following attack vectors:

1. Inherited Issues from Compound

    - Since the protocol heavily leverages Compound's algorithms and approaches, we verified whether it is safeguarded against attack vectors known from Compound, particularly the inflation attack.
    - We paid special attention to sections of the code where the implementation diverges from Compound's, such as:
        - The absence of certain parameter validations in the RiskEngine compared to the Comptroller.
        - Self-liquidation logic.
        - Self-seize mechanisms.
        - Solutions successfully used in Compound and adopted in Pike (e.g., setComptroller/setRiskEngine) were also re-examined for any potential risks.

2. borrowOnBehalfOf Implementation

    - We verified that delegating borrowing rights does not create opportunities for misappropriating other users' funds or enable any other malicious activities.

3. Arithmetic Operations

    - We verified that overflows, underflows, and rounding errors are handled correctly, ensuring they do not produce any unintended side effects.

4. Configuration Errors

    - We examined whether the parameters set by the system administrator are validated sufficiently to prevent any misconfiguration issues.

5. eMode Categories Logic

    - We examined scenarios related to how entering and exiting collateral and borrow eModes is handled, including the potential for manipulating LTV or liquidation thresholds.

6. Membership Logic

    - The protocol introduces different types of collateralMembership and borrowMembership states.
    - If either membership flag is true, the user's account assets must include that token.
    - We analyzed the processes for changing these membership states to identify any way an attacker could bypass checks or improperly remove required assets.

7. pToken and Reserves Configuration

- The fork adds reserve logic such as configuratorReserves and ownerReserves, which differs from the simpler reserve approach in Compound.
- We reviewed these modifications to assess any potential impact on accounting or asset management.

8. Interest Rate Model

- We verified the correctness of the double jump rate model implementation calculations, as well as the possibility of its misconfiguration.
- Particular attention was given to scenarios where the borrow rate exceeds the maxBorrowRateMantissa threshold, examining how the protocol calculates or waives interest in that situation.
- We looked for any inconsistencies that might enable unintended borrowing advantages, including in edge-case conditions.

9. Oracle and Price Feed Checks

- Multiple oracle feeds are used to provide fallback mechanisms if one feed returns incorrect data.
- We considered the possibility of stale or manipulated pricing data and how the fallback logic addresses incorrect or out-of-date information.
- We reviewed common issues specific to oracle implementations, including proper handling of decimals, stale prices, zero or negative prices.

10. Storage Collisions

- We assessed the contract's storage layout and upgrade approach to identify risks of variable overlap or overwritten data in upgradeable contracts.

11. Deployment and Proxy Model

- The fork implements a diamond proxy-based architecture, and we evaluated how initialization and upgrades are controlled.
- We focused on any potential misconfigurations that might allow unauthorized modifications to contract functionality.

12. Cross-Contract Read-Only Reentrancy

- We explored interactions among pTokens and other modules, where reentrancy could occur through read-only calls made before state updates are finalized.
- This included reviewing the use of nonReentrant and checks-effects-interactions pattern and whether they adequately address multi-contract invocation scenarios.

The issues we identified are listed below.

The project exhibits the typical level of centralization seen in similar solutions. The administrator is able to upgrade contract logic and modify critical parameters, which could potentially lead to losses, freezing of user funds, or even misappropriation. The administrator is also responsible for selecting the tokens used in the system, ensuring they are not excessively risky in terms of market conditions and that they meet the technical requirements (in particular, the protocol does not support rebaseable tokens). To mitigate these risks, it is essential to utilize multisig accounts and governance mechanisms.

2. FINDINGS REPORT

2.1 Critical

C-1	Inflation Attack in `PTokenModule.deposit` Draining Initial Deposits
Severity	Critical
Status	Fixed in 8eec3307

Description

This issue has been identified within the `deposit` function of the `PTokenModule` contract.

An attacker can front-run initial depositors by depositing 1 wei of the underlying token to receive a single pToken. By then directly transferring a large amount of the underlying token to the contract, the attacker artificially inflates the `exchangeRate` because `totalSupply` remains unchanged while `getCash()` grows significantly. Consequently, any subsequent depositor who supplies tokens may end up receiving zero pTokens (due to integer division at the inflated exchange rate) and thus lose their funds. The attacker can then redeem their single pToken and drain the entire contract balance, including other users' deposits.

The issue is classified as **critical** severity because it can result in a complete loss of funds for unsuspecting users.

Recommendation

We recommend minting a small amount (for example, 1000) of "dead shares" during the initial mint to ensure that this inflation attack is not possible. Additionally, implement a check that the minted amount is greater than 0 to ensure a user always receives a positive number of pTokens when depositing.

Client's Commentary

> We will implement an initial mint of a small shares to prevent inflation attacks. Also we'll add a check to ensure that depositors receive a positive number of pTokens.

2.2 High

H-1	Arithmetic Overflow in `getPrice` When Feeds Return Large Values
Severity	High
Status	Fixed in cb343179

Description

This issue has been identified within the `getPrice` function of the `ChainlinkOracleComposite` contract.

`getPrice` normalises each feed answer and then multiplies the current composite price by that rate:

```
rate = uint256(price)
     * 10**(SCALING_DECIMALS - feed.decimals());
compositePrice = (compositePrice * rate)
                 / SCALING_FACTOR; // 36-dec fixed-point
```

If a feed reports `price > 1.16 * 10^(5 + feed.decimals)` (≈ $100 000 when denominated in wei), the term

```
compositePrice * rate * 10^36
```

exceeds the 256-bit limit. The call reverts with arithmetic overflow, freezing every contract that depends on this oracle for lending, liquidation, or pricing logic.

The issue is classified as **high** severity because a single inflated data point bricks the entire oracle and all protocols integrated with it.

Recommendation

We recommend either replacing the simple `* /` pair with a 512-bit safe-math library such as OpenZeppelin's `mulDiv` or reducing `SCALING_DECIMALS`.

Client's Commentary

> We will fix it by implementing OZ math library `mulDiv`.

2.3 Medium

M-1	Improper Validation of Price Feed Data
Severity	Medium
Status	Fixed in 8eec3307

Description

This issue has been identified within both the `ChainlinkOracleProvider` and `PythOracleProvider` contracts.

In these modules, price data is fetched as a signed integer, then immediately cast to an unsigned integer. This casting can produce two main risks:

1. **Negative Price Values**
   Although highly unlikely for typical assets, if the oracle ever returns a negative price, casting that negative value to `uint256` would turn it into a very large positive integer (due to underflow). This can cause significant mispricing in collateral valuations and liquidity calculations, as the system would interpret a negative value as a highly inflated price.

2. **Zero Price Values**
   In more realistic scenarios, an oracle may temporarily return a zero price due to a feed error or extreme market conditions. A zero price passes through the cast without reverting, but subsequent validation steps in the Oracle Engine treat a zero price as valid if fallback oracles are configured and subsequently revert, even if the fallback oracle returns a valid price. This scenario can halt liquidation, redemption and borrowing operations, even when one of the oracles is functioning correctly. Ultimately, this could lead to bad debt if liquidations are postponed during volatile market events.

Examples:

- **ChainlinkOracleProvider**

```
(, int256 price,, uint256 updatedAt,) = config.feed.latestRoundData();
return uint256(price) * (10 ** (36 - assetDecimals - priceDecimals));
```

  - A negative `price` becomes a large positive number.
  - A zero `price` is treated as valid and can cause reverts downstream.

- **PythOracleProvider**

```
uint256 priceIn18Decimals = (uint256(uint64(priceInfo.price)) * (10 ** 18))
    / (10 ** uint8(uint32(-1 * priceInfo.expo)));
```

- A negative `price` similarly becomes a large positive number.
- A zero `price` is treated as valid and can trigger reverts in other parts of the system.

The issue is classified as **medium** severity because it can lead to a temporary halt of liquidations, borrowing, or redemptions in the protocol, potentially causing bad debt.

Recommendation

We recommend adding explicit checks for negative or zero prices. For example:

```
if (price <= 0) {
    revert InvalidPrice();
}
```

Such checks ensure that:

- Negative values do not get misinterpreted as extremely large positive numbers.
- Zero values are handled properly rather than silently causing reverts later in the Oracle Engine.

Client's Commentary

> We'll add explicit checks for negative and zero prices in oracle providers.

M-2	E-Mode Misconfiguration Causing Inaccurate Collateral Accounting
Severity	Medium
Status	Fixed in 8eec3307

Description

This issue has been identified within the `supportEMode` function of the `RiskEngineModule` contract.

An asset may be removed from an existing E-Mode category while still retaining its elevated collateral factor. In this scenario, if a user remains in E-Mode, the removed asset continues to be valued with a higher collateral factor instead of returning to its true collateral value. The same issue applies to borrowed assets and their liquidation thresholds. This mismatch can yield inaccurate liquidity calculations and potentially lead to bad debt for the protocol.

The issue is classified as **medium** severity because it can expose the protocol to significant financial risk through inaccurate collateral accounting.

Recommendation

We recommend prohibiting the removal of asset permissions from an existing E-Mode category. If adjustments are necessary, consider creating a new E-Mode category instead. Alternatively, implement robust checks in `_getCollateralFactor` and `_getLiquidationThreshold` to verify that if an asset is no longer permitted in a particular E-Mode, the default collateral or liquidation thresholds are applied.

Client's Commentary

> We will implement the alternative solution by ensuring that when an asset is removed as collateral, the default risk parameters are used in collateral calculations.

M-3	Excessively High Borrow Rate Processed as Zero
Severity	Medium
Status	Fixed in 8eec3307

Description

This issue has been identified within the `accrueInterest` function, which checks whether the computed borrow rate exceeds `borrowRateMaxMantissa`. If it does, the function updates `accrualBlockTimestamp` and returns early:

```
if (IInterestRateModel(address(this)).getBorrowRate(
    getCash(), snapshot.totalBorrow, snapshot.totalReserve
) > $.borrowRateMaxMantissa) {
    $.accrualBlockTimestamp = currentBlockTimestamp;
    return;
}
```

This effectively treats any excessively high borrow rate as zero interest accrual without notifying users or developers. Consequently, if the interest rate model malfunctions or market conditions become abnormal, the protocol silently applies no interest instead of alerting stakeholders. Additionally, `borrowRateMaxMantissa` is only set in the constructor and may not align with changing interest rate models over time.

The issue is classified as **medium** severity because it leads to inaccurate financial calculations and hinders prompt detection of anomalous or erroneous borrow rate conditions.

Recommendation

We recommend reverting the transaction if the computed borrow rate exceeds the maximum threshold, prompting immediate investigation rather than silently ignoring it. For example:

```
if (IInterestRateModel(address(this)).getBorrowRate(
    getCash(), snapshot.totalBorrow, snapshot.totalReserve
) > $.borrowRateMaxMantissa) {
    revert PTokenError.ExcessiveBorrowRate();
}
```

Additionally, consider making `borrowRateMaxMantissa` configurable by an authorized administrator so that it remains aligned with the current interest rate model.

Client's Commentary

> We'll make borrowRateMaxMantissa configurable and revert transactions exceeding the threshold, with existing freshness check to clearly indicate `accrueInterest` issue.

2.4 Low

L-1	Oracle Ownership Vulnerability During Deployment
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified within the deployment flow of `OracleEngine`, `ChainlinkOracleProvider`, and `PythOracleProvider` contracts.

Because of the Cannon execution flow, there is a brief moment where ownership is not fully restricted. During this time, a malicious actor could attempt to acquire ownership of any of the oracle contracts. While such an attempt would likely revert the deployment and be obvious, if the upgrade process continues despite a partial failure, the attacker could manipulate oracle prices or disrupt protocol operations.

The issue is classified as **low** severity because it occurs only during a short deployment or upgrade window. However, if overlooked, it may have significant consequences (e.g., price manipulation).

Recommendation

We recommend locking down the ownership of oracle contracts immediately upon deployment to prevent any unauthorized ownership claims. Additionally, perform any upgrade processes in a single atomic transaction, ensuring that a partial failure reverts the entire upgrade rather than leaving the system in a vulnerable state.

Client's Commentary

> We will implement immediate ownership locking for oracle provider contracts during deployment to prevent unauthorized ownership claims. however we already use atomic deployment for oracle engine using factory.

L-2	Missing Zero-Address Checks in Key Functions
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified within the `setRiskEngine()`, `setOracle()`, `transfer()`, `transferFrom()`, `redeem()`, and `withdraw()` functions.

- **`setRiskEngine()` and `setOracle()`**
  These functions do not validate that the new addresses provided are non-zero. Using `address(0)` for critical protocol components could cause undefined behavior or reverts during execution (e.g., calls to a nonexistent RiskEngine or Oracle).

- **`transfer()` and `transferFrom()`**
  These functions do not validate that the destination address `dst` is non-zero, violating the ERC20 standard and potentially resulting in irreversible token burns.

- **`redeem()` and `withdraw()`**
  These functions do not validate that the `receiver` address is non-zero. Assigning `address(0)` here may cause an unintended token burn or a loss of funds.

Recommendation

We recommend adding zero-address checks in the relevant functions to prevent accidental or malicious misconfiguration.

Client's Commentary

> We will implement the suggested enhancement.

L-3	Missing Range Validation for Key Parameters
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified in several parameter setters and configuration functions—such as `setProtocolSeizeShare()`, `setCloseFactor()`, `configureMarket()`, `configureEMode()`, and the handling of `borrowRateMaxMantissa_`. None of these parameters are currently constrained by explicit interval checks.

Specific risks include:

- **`setProtocolSeizeShare()`**: Failing to ensure that `reserveFactorMantissa + protocolSeizeShareMantissa` remains under 1e18 could result in seizing more collateral than exists.
- **`setCloseFactor()`**: The `closeFactor` could be set to an extreme value above 1e18, breaking normal liquidation assumptions.
- **`configureMarket()`, `configureEMode()`**: The `liquidationIncentiveMantissa` may be set to an unreasonably high or low value, distorting the liquidation process.

Recommendation

We recommend introducing explicit validation in each function to ensure these parameters remain within sensible intervals.

Client's Commentary

> We will implement the suggested enhancement.

L-4	RiskEngine Error Checks Using Numerical Constant Instead of `NO_ERROR`
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified across multiple checks in the RiskEngine-related code where numerical comparisons against `0` are used rather than the enumerated value `RiskEngineError.Error.NO_ERROR`.

For example:

```
if (allowed != 0) { ... }
```

Replacing such comparisons with:

```
if (allowed != RiskEngineError.Error.NO_ERROR) { ... }
```

improves code clarity and reduces confusion for developers and auditors.

Recommendation

We recommend replacing numeric comparisons with the more explicit enumerated constant to make the code self-documenting and less error-prone.

Client's Commentary

> We will implement the suggested enhancement.

L-5	Misleading or Incorrect Comments
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified in both `PTokenModule` and `RiskEngineModule` due to comments that do not accurately reflect the underlying code logic:

- **`PTokenModule`**
  - `_getReserveShares(...)` is described as retrieving a "block number" but actually handles the distribution of reserves.
  - `_getBlockTimestamp()` also claims to retrieve the "block number" but returns `block.timestamp`.
- **`RiskEngineModule`**
  - `exitMarket(...)` has a line comment that says:

    ```
    /* Return true if the sender is not already 'in' the market as collateral */
    ```

    However, the function does not return a boolean—it checks membership and exits early if the user is not already in the market.

Such inconsistencies can mislead developers or auditors who rely on comments to understand the code.

Recommendation

We recommend updating or removing incorrect comments to ensure that they accurately describe the functions' purposes.

Client's Commentary

> We will implement the suggested enhancement on Natspec.

L-6	Individually Changeable `RiskEngine` per `pToken` Leading to Inconsistency
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified in the architecture allowing `riskEngine` to be changed individually for each `pToken`. There is no robust mechanism to ensure a coordinated migration for all `pToken`s simultaneously.

Potential issues include:

- **Incorrect risk calculations**: Different `RiskEngine`s may use varying collateral or liquidity logic, leading to inconsistent borrow allowances and liquidation criteria.
- **Inconsistent user state**: If some `pToken`s migrate to a new `RiskEngine` while others remain on the old one, the system may fail to recognize a user's overall collateral or debt, allowing an overborrow or blocking legitimate actions.

The issue is classified as **low** severity because it requires a particular combination of conditions (e.g., switching or adding a new `RiskEngine`) to become exploitable, but it can still lead to inconsistent accounting and potential exploits.

Recommendation

We recommend removing the external `setRiskEngine` function from the `PTokenModule` so that the `RiskEngine` cannot be changed arbitrarily per `pToken`. If a migration is unavoidable, consider making the `RiskEngine` reference immutable within the Beacon implementation to ensure consistency across all `pTokens`.

Client's Commentary

> We will implement the suggested enhancement.

L-7	Inefficient Struct Usage in `ExponentialNoError` Library
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified within the `ExponentialNoError` library, which uses `struct` types (`Exp` and `Double`) to store fixed-precision decimals. Because structs must reside in memory, this design may be more cumbersome and less gas-efficient than user-defined value types, for example:

```
type Exp is uint256;
type Double is uint256;
```

Using these can reduce memory usage, simplify code, and potentially lower gas costs while improving readability.

Recommendation

We recommend refactoring the `ExponentialNoError` library to leverage user-defined value types. This approach can improve efficiency and maintainability without sacrificing correctness.

Client's Commentary

> We will implement the suggested enhancement.

L-8	Use `require` Instead of `if (condition) revert`
Severity	Low
Status	Fixed in 8eec3307

Description

This finding has been identified in areas where the contract uses `if (condition) revert Error();` instead of `require(condition, Error());`. As of Solidity 0.8.26, `require` statements can accept custom error messages, making them more concise and consistent with typical revert patterns.

While the current usage does not introduce a direct vulnerability, adopting `require` for simple checks can enhance code readability and alignment with Solidity conventions.

Recommendation

We recommend replacing `if (condition) revert Error();` patterns with `require(!condition, Error());` statements wherever possible. This change maintains clear error messages and offers a more idiomatic revert pattern.

Client's Commentary

> We will implement the suggested enhancement.

L-9	Confusing Error Handling in `mintFresh` and `redeemFresh` Functions
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified in the `mintFresh` and `redeemFresh` functions. Both enforce that only one of two parameters (`mintTokensIn` and `mintAmountIn` for `mintFresh`; `redeemTokensIn` and `redeemAmountIn` for `redeemFresh`) can be non-zero at a time. If both are provided as non-zero, the code reverts with `CommonError.ZeroValue()` — an error name that does not accurately describe the situation.

Because `ZeroValue` suggests that a parameter was unexpectedly zero, it can confuse developers attempting to diagnose a revert. In reality, the revert is caused by providing two non-zero parameters simultaneously.

Recommendation

We recommend replacing the `ZeroValue` error with a more descriptive name—for example, `NonZeroValuesNotAllowed` or `OnlyOneInputAllowed`. This ensures the revert message accurately describes why the check fails and simplifies debugging.

Client's Commentary

> We will implement the suggested enhancement.

L-10	Risk of Storage Slot Overlap Leading to Unpredictable Behavior
Severity	Low
Status	Fixed in 8eec3307

Description

This issue has been identified due to the shared use of the same storage address (`_SLOT_PTOKEN_STORAGE`) for both transient reentrancy guard data and persistent protocol data. While currently functional, any future refactoring that merges these two data sets into a single storage layout may cause overlap of the reentrancy guard flag with other state variables, leading to unpredictable or corrupted behavior.

Recommendation

As a precautionary measure, we recommend assigning a dedicated storage slot for the reentrancy guard flag to ensure clear separation from other storage variables. This isolation helps protect against unintentional overlap during upgrades or refactoring.

Client's Commentary

> We will implement the suggested enhancement.

L-11	Missing EIP-165 Interface Compliance Checks
Severity	Low
Status	Acknowledged

Description

This issue has been identified in functions that accept addresses of external contracts—such as `setOracle(address newOracle)` and `supportMarket(IPToken pToken)` — without verifying EIP-165 interface support.

EIP-165 allows a contract to declare which interfaces it implements. Verifying that a provided address actually supports the required interface (for example, `IOracleEngine` or `IPToken`) helps ensure it behaves correctly when called by the protocol. If no check is done, an invalid address could be set, leading to unexpected behavior or reverts.

Recommendation

We recommend implementing EIP-165 checks (e.g., via `IERC165(newOracle).supportsInterface(type(IOracleEngine).interfaceId)`) in functions that accept external contract addresses. This ensures the protocol only interacts with contracts that implement the expected interfaces.

Client's Commentary

> Given that we use separate interfaces due to the proxy pattern, we do not see a need to implement EIP-165 for contracts.

L-12	Undescriptive Errors on Arithmetic Underflow
Severity	Low
Status	Acknowledged

Description

This issue has been identified in arithmetic operations susceptible to underflow. While Solidity 0.8.x automatically reverts on underflow, the revert message does not indicate the specific cause or location. This obscures debugging.

Examples include:

- `PTokenModule._transferTokens`:

```
uint256 srcTokensNew = $.accountTokens[src] - tokens;
```

- `PTokenModule.redeemFresh`:

```
$.totalSupply = $.totalSupply - redeemTokens;
$.accountTokens[onBehalfOf] = $.accountTokens[onBehalfOf] - redeemTokens;
```

- `PTokenModule.repayBorrowFresh`:

```
uint256 accountBorrowsNew = accountBorrowsPrev - actualRepayAmount;
uint256 totalBorrowsNew = $.totalBorrows - actualRepayAmount;
```

If the subtracted amount exceeds the current balance or supply, it causes an underflow revert with no clear error message.

Recommendation

We recommend adding explicit `require` statements before these subtraction operations. For example:

```
require($.accountTokens[src] >= tokens, PTokenError.InsufficientBalance());
```

Such validation ensures that any underflow triggers a descriptive revert message, improving debuggability and user experience.

Client's Commentary

> Given the contract size limit, we acknowledge this issue but see no need to implement it.

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes