Quantstamp

Ensuro 3

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Insurance
Timeline	2025-11-24 through 2025-12-03
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Ensuro Docs
Source Code	• https://github.com/ensuro/ensuro #637ee48
Auditors	• Andy Lin Senior Auditing Engineer • Leonardo Passos Senior Research Engineer • Ed Zulkoski Senior Auditing Engineer

Documentation quality	High
Test quality	High
Total Findings	13 Fixed: 8 Acknowledged: 5
High severity findings	0
Medium severity findings	2 Fixed: 1 Acknowledged: 1
Low severity findings	9 Fixed: 7 Acknowledged: 2
Undetermined severity findings	0
Informational findings	2 Acknowledged: 2

Summary of Findings

We audited the Ensuro protocol, a decentralized on-chain insurance marketplace that connects liquidity providers with insurance product creators through a capital-tranching system. The protocol enables liquidity providers to deposit stablecoins into junior or senior risk tranches that earn yield from underwriting insurance products. Policyholders receive coverage represented as ERC-721 NFTs, with automated payout settlement handled through modular `RiskModule` components.

The protocol is built around several core contracts. The `PolicyPool` acts as the central component that links liquidity providers and customers taking out insurance policies. `RiskModule` contracts allow end-users to create new policies. `PremiumsAccount` stores premiums collected from existing policies and is used to pay out claims. When premiums are not sufficient, the `PremiumsAccount` can borrow from `EToken` liquidity, which represents funds supplied by liquidity providers. These ETokens are rebasing ERC20 tokens whose value may temporarily decrease if they are used to cover policy payouts.

The codebase shows strong engineering practices with clear separation of concerns across modules such as `PolicyPool`, `EToken`, `PremiumsAccount`, and `RiskModule`. The design is thoughtful and supported by good documentation. Test coverage is high overall at more than 85 percent branch coverage, but there are still several non-trivial functions in `PremiumsAccount.sol` that remain untested and should be covered.

During our security review, we identified some issues. We recommend that the development team address all identified findings before deploying the protocol to production.

Fix Review Update: The team has fixed or acknowledged all issues.

ID	DESCRIPTION	SEVERITY	STATUS
ENSR-1	Pre-Loss Withdrawal Allows LPs to Escape Yield Vault Losses	• Medium	Fixed
ENSR-2	Yield Vault Losses Can Block Vault Migration and Correct Accounting	• Medium	Acknowledged
ENSR-3	`SCR` Manipulation Can Halt or Severely Throttle Liquidity Provider Withdrawals	• Low	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
ENSR-4	Unlimited ERC-20 Approvals Enable Malicious Yield Vault to Drain Funds	• Low ⓘ	Fixed
ENSR-5	`UUPS` Upgrade Can Change the Pool's Underlying Currency, Corrupting Economic Invariants	• Low ⓘ	Fixed
ENSR-6	`changeComponentStatus()` Allows Setting Inactive, Bricking Component and Causing Fund Lockup	• Low ⓘ	Fixed
ENSR-7	`setExposureLimit()` Uses Strict Inequality, Preventing Limit Equal to Current Exposure	• Low ⓘ	Fixed
ENSR-8	Senior Loan Limit Check Uses Strict Inequality, Inconsistent with Junior Logic	• Low ⓘ	Fixed
ENSR-9	Precision Loss in Cost of Capital Calculation Due to Premature Division	• Low ⓘ	Fixed
ENSR-10	Signature Replay Across Riskmodules Due to Missing Riskmodule Address in Signed Data	• Low ⓘ	Fixed
ENSR-11	Payout-worthy incident may not be paid if it occurs near the policy expiration	• Low ⓘ	Acknowledged
ENSR-12	Use of Centralized Stablecoins May Allow External Freezing of Protocol Funds	• Informational ⓘ	Acknowledged
ENSR-13	Missing Zero-Address Check for Optional Senior Etoken in `_borrowFromEtk()`	• Informational ⓘ	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Also, the role access configuration on the proxy layer is outside the scope of this audit. We assume that all configurations match the document the team shared with us.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following

    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

This audit scope includes the smart contracts excluding the mock contracts.

Files Included

Files: `contracts/*`

Files Excluded

Files: `contracts/mocks/*`

Operational Considerations

1. The contract architecture moves access control to the proxy layer using the `AccessManagedProxy` contract. This requires the deployment process to correctly assign the corresponding roles to the appropriate functions. In this audit, we assume that access control will be properly configured during deployment.
2. The contract system is upgradeable. We assume the `UPGRADE_ROLE` is securely guarded and that contract upgrades will not be malicious.
3. In `ETKLib`, the `ScaledAmount` struct includes a `lastUpdate` field of type `uint32`. This data type can represent timestamps only up to the year 2106, so the protocol will require an upgrade before that time.

Key Actors And Their Capabilities

Ensuro uses `AccessManagedProxy` together with a shared `IAccessManager` instance to centralize permissions. Roles are assigned at the proxy layer and gate all state-changing calls into the core components, while most view and pure functions are open to the `PUBLIC_ROLE` for observability. The following is a summary of the roles provided in the documentation from the team (see: doc). Note that the concrete role setup is outside the scope of this audit, as we did not review the actual configuration. We assume they configured the access as specified in the document.

- **Public Users (** `PUBLIC_ROLE` **)**: Any address allowed to interact with user-facing flows: deposit/withdraw LP capital via `PolicyPool` and `EToken`, transfer/approve NFTs and ERC20s (including `Cooler` withdrawals), and trigger certain PremiumsAccount actions that already have strong in-contract validation.
- **Protocol Governance (** `ADMIN_ROLE` **,** `LEVEL1_ROLE` **,** `LEVEL2_ROLE` **,** `UPGRADE_ROLE` **)**: Controls initialization, structural configuration, and upgrades. `LEVEL1_ROLE` manages components and key addresses (adding/removing RiskModules, setting treasury, yield vaults, whitelists, etc.), `LEVEL2_ROLE` tunes sensitive risk/credit parameters (exposure limits, loan limits, deficit ratios, CoC parameters, cooler cooldowns), `ADMIN_ROLE` is used for one-time initialization, and `UPGRADE_ROLE` can change implementations via `upgradeToAndCall` on all major proxies.
- **Safety Guardians (** `GUARDIAN_ROLE` **,** `CHANGE_STATUS_ROLE` **)**: Can pause/unpause the `PolicyPool` and change component statuses (active/deprecated/suspended), effectively controlling whether RiskModules, PremiumsAccount, and other components can be used while preserving funds and existing positions.
- **Core System Components (** `POLICY_POOL_ROLE` **,** `PREMIUMS_ACCOUNT_ROLE` **,** `COOLER_ROLE` **)**: Machine-to-machine roles used when components call each other. `POLICY_POOL_ROLE` allows the `PolicyPool` to drive policy lifecycle callbacks on `PremiumsAccount` and to move capital in/out of `EToken`; `PREMIUMS_ACCOUNT_ROLE` allows PremiumsAccount to borrow/lock/unlock funds from eTokens; `COOLER_ROLE` allows the `Cooler` contract to rebalance and redistribute eToken liquidity during withdrawals.
- **Yield and Treasury Operators (** `REBALANCER_ROLE` **,** `WITHDRAW_PREMIUMS_ROLE` **,** `REPAY_LOAN_ROLE` **)**: Operational roles that move idle funds into and out of external yield vaults, withdraw earned premiums to treasury, and repay internal loans. They do not control policy lifecycle directly but materially affect liquidity, solvency, and profit extraction.
- **Underwriters / Product Operators (** `POLICY_CREATOR_ROLE` **,** `CANCEL_POLICY_ROLE` **,** `REPLACER_ROLE` **,** `RESOLVER_ROLE` **)**: Per-RiskModule roles that create, cancel, replace, and resolve policies. These actors define pricing parameters off-chain, decide which risks are underwritten, and determine claim outcomes, subject to PolicyPool-level minima and accounting.
- **LP Whitelisting Operators (** `WHITELIST_ROLE` **)**: Manage investor access via `LPManualWhitelist`, deciding which addresses can act as LPs and under what default conditions.

Overall, governance and operational roles (`LEVEL1` / `LEVEL2` / `UPGRADE` / `REBALANCER` /etc.) have broad systemic impact over risk, capital flows, and upgradeability, while underwriter roles control product-specific policy lifecycles, and `PUBLIC_ROLE` remains limited to user-facing capital and policy interactions that are protected by in-contract checks.

Findings

ENSR-1
Pre-Loss Withdrawal Allows LPs to Escape Yield Vault Losses

● Medium ⓘ Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `5eaf2c33d34c6d9dddca8af4fff1254072cc4981`.
The client provided the following explanation:

```
We changed the code to record earnings/losses before any LP withdrawal. It's not done on the _deinvest
method itself because some calculations performed prior to calling it depend on the totalSupply which
is updated by the recording of losses. A comment was added in that method documentation to encourage
future users to update accounting before calling it.
  We also added a call to  recordEarnings()  in two methods of the PremiumsAccount contract that also
might be affected by non recorded losses / earnings: repayLoans and withdrawWonPremiums.
```

We verified that this is fixed by forcing `recordEarnings()` before deinvestment.

File(s) affected: `contracts/Reserve.sol`

Description: When an integrated ERC4626 yield vault suffers a loss, the `_deinvest()` function only records earnings when `amount > _invested` but NEVER records losses. The loss remains unrecorded until someone explicitly calls `recordEarnings()`. During this window, liquidity providers (LPs) can withdraw their funds at the pre-loss value, effectively transferring their share of losses to remaining LPs.

The vulnerable code in `_deinvest()`:

```
function _deinvest(IERC4626 yieldVault_, uint256 amount) internal {
  yieldVault_.withdraw(amount, address(this), address(this));
  if (amount > _invested) {
    // If deinvests more than was already invested, then there's an earning and we have to record it.
    _yieldEarnings(int256(amount - _invested));
    _invested = 0;
  } else {
    _invested -= amount;  // Loss never recorded, just decrements _invested
  }
}
```

The issue is that when the vault has suffered a loss, `amount` withdrawn < value of shares before loss, and the function just decrements `_invested` without calling `_yieldEarnings()` with a negative value. The loss is "hidden" until `recordEarnings()` is called, allowing early withdrawers to get their shares valued at pre-loss rates.

The public `withdrawFromYieldVault()` function calls `_deinvest()` directly, allowing anyone to trigger withdrawals that bypass loss socialization. Additionally, normal LP withdrawals via `_transferTo()` also call `_deinvest()`, creating a "first-to-exit wins" race condition where sophisticated LPs can monitor the yield vault's performance and withdraw before losses are socialized.

Exploit Scenario:

1. EToken has 10M USDC, with 9M invested in an ERC4626 yield vault
2. The yield vault suffers a 10% loss (vault shares now worth 8.1M instead of 9M)
3. Before anyone calls `recordEarnings()`, sophisticated LP Alice monitors the vault and detects the loss
4. Alice calls `withdraw()` on the EToken to redeem 1M of her eToken shares
5. The withdrawal triggers `_transferTo()` → `_deinvest()` with the loss still unrecorded
6. Alice receives 1M USDC at the pre-loss valuation (should have received ~910K)
7. Alice has successfully extracted ~90K more than her fair share
8. Later, when `recordEarnings()` is called, the 900K loss is socialized among remaining LPs
9. The remaining LPs absorb not just their proportional loss but also Alice's escaped loss

This is a critical value extraction vulnerability where early withdrawers can front-run loss socialization.

Recommendation: Modify `_deinvest()` to ALWAYS sync PnL before deinvesting, similar to how `recordEarnings()` works:

```
function _deinvest(IERC4626 yieldVault_, uint256 amount) internal {
  // ALWAYS sync PnL before any deinvestment
  uint256 currentValue = yieldVault_.convertToAssets(yieldVault_.balanceOf(address(this)));
  int256 unrealizedPnL = int256(currentValue) - int256(_invested);
  if (unrealizedPnL != 0) {
    _invested = currentValue;
    _yieldEarnings(unrealizedPnL);  // Record gains OR losses
  }

  // Now perform deinvestment (with _invested reflecting current reality)
  yieldVault_.withdraw(amount, address(this), address(this));
  _invested -= amount;
}
```

This ensures that all PnL (both gains and losses) is socialized immediately before any withdrawal, preventing the race condition.

ENSR-2

Yield Vault Losses Can Block Vault Migration and Correct Accounting

● Medium ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
This situation is highly unlikely because the protocol invests only in conservative, well-established
instruments (e.g., tokenized T-bills or over-collateralized lending on Aave/Compound) and the Ensuro
team continually monitors asset management performance. If such an extreme event were to occur,
governance can manually provide a grant to recapitalize the premiums account and immediately change or
remove the yield vault to restore accounting and operations. Introducing an explicit on-chain
"recovery" state would add complexity for a very rare edge case while offering limited additional
value, so we consider the current monitoring plus manual recapitalization pathway the more pragmatic
mitigation.
```

Description: Large yield vault losses greater than the configured max deficit can prevent the protocol from successfully recording those losses (via `recordEarnings` or `setYieldVault`), effectively blocking vault migration and accurate accounting.

When replacing the yield vault via `setYieldVault()`, the contract first attempts to deinvest all funds from the existing vault, then records earnings/losses using `_yieldEarnings(int256(deinvested) - int256(_invested))`.

If the vault has suffered severe losses (e.g., a hack or malfunction), deinvested may be significantly below `_invested`, resulting in a large negative earnings value. This negative value flows into PremiumsAccount's `_yieldEarnings`, which ultimately calls:

```
uint256 excess = _payFromPremiums(-earnings);
require(excess == 0, LossesCannotExceedMaxDeficit(...));
```

The failure condition occurs when `loss > maxDeficit = activePurePremiums * deficitRatio`. In that case, `LossesCannotExceedMaxDeficit` is thrown.

Recommendation: Once a large vault loss is detected:

1. Enter a "recovery" state:
2. No new RM exposure.
3. Possibly stricter withdrawal limits.
4. Require governance to either recapitalize (via `receiveGrant`) or write off / restructure deficits.
5. Keep this state machine explicit on-chain so LPs and integrators can detect it.

ENSR-3

SCR Manipulation Can Halt or Severely Throttle Liquidity Provider Withdrawals

● Low ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
The protocol already enforces a per-risk-module max-exposure in the PolicyPool that effectively caps
total SCR, and the signer component can impose a per-policy max SCR. Leaving policy pricing limits and
per-policy parameters off-chain was an explicit design decision for this version to retain pricing
flexibility; if a future product (or further decentralization of the protocols) requires on-chain
enforcement, those constraints can be codified by implementing a new underwriter implementation to
enforce per-policy caps or other kind of per-policy restrictions.
```

Description: The `EToken` withdrawal logic allows liquidity providers to withdraw only up to `totalWithdrawable = totalSupply − SCR * liquidityRequirement`. Because `SCR` can be increased arbitrarily through `lockScr()` when new policies are created, a privileged actor (e.g., a risk module, policy pool, or governance) can raise SCR up to the maximum allowed (`maxUtilizationRate * totalSupply`). When `liquidityRequirement * maxUtilizationRate ≥ 1`, the system can be pushed into a state where `totalWithdrawable == 0`.

This effectively **freezes all withdrawals** for liquidity providers, even when the protocol is fully solvent.

Further, the protocol does **not** enforce any on-chain relationship between:

1. SCR locked
2. premiums collected
3. CoC (Capital Cost)
4. actual capital available in the PremiumsAccount

As a result, a misconfigured or malicious module can cheaply lock extremely large SCR amounts (even with minimal or zero CoC) and effectively trap LP liquidity.

Recommendation: Introduce explicit on-chain constraints preventing SCR inflation from eliminating LP withdrawal capacity. Potential mitigations include:

1. **Per-Policy SCR Caps:**
   Guarantee SCR is proportionate to premium or CoC; forbid policies whose SCR exceeds economically sound bounds.
2. **Epoch-Based SCR Growth Limits:**
   Restrict how fast total SCR can grow per block/day/epoch, preventing sudden liquidity freezes.

ENSR-4

Unlimited ERC-20 Approvals Enable Malicious Yield Vault to Drain Funds

• Low ⓘ Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `7d9180df29bc238ceb194f9e05dbd1837c152b3e`.
The client provided the following explanation:

```
We changed the code, removing the infinite approval. Now we approve only before doing deposit.
```

We verified that the team fixed the issue by approving allowances on demand. However, some tokens (e.g., USDT) require the allowance to be set to zero before being updated to a new value. Operations should ensure that such tokens are not integrated in the future.

Description: When a new yield vault is configured using `setYieldVault`, the `Reserve` contract executes

```
if (address(newYieldVault) != address(0)) {
    asset.approve(address(newYieldVault), type(uint256).max);
}
```

This grants the new yield vault contract an unlimited allowance to pull assets directly from the `Reserve`. If the yield vault is malicious, compromised, or upgraded to a malicious implementation, it can call `transferFrom()` to drain all assets held by the `PremiumsAccount` or `EToken` reserves.

Because the approval amount is effectively infinite and not reset during later interactions, the `Reserve` permanently exposes its entire balance to any malicious behavior of the yield vault.

This is especially dangerous because:

- Yield vaults are external contracts not controlled by the protocol.
- ERC-4626 vaults may be upgradeable, enabling post-deployment compromise.
- This affects all reserves: `PremiumsAccount` balances, `EToken` liquidity, LP funds, and SCR capital backing policies.

Thus, a malicious or compromised vault can fully liquidate protocol funds without violating any protocol checks, bypassing solvency logic, SCR rules, or withdrawal limitations.

Recommendation: Avoid granting unlimited (`type(uint256).max`) allowances to the yield vault. Instead:

- Approve only the minimum amount necessary per operation or up to a conservative spending cap, and reset allowances to zero when not in use; and
- Optionally, complement this with operational controls such as off-chain monitoring and scheduled allowance reductions to approximate time-limited approvals.

ENSR-5

UUPS Upgrade Can Change the Pool's Underlying Currency, Corrupting Economic Invariants

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `a6330f1d1e74b70519f3dd86ed312953fedae97c`.
The client provided the following explanation:

```
We restored (it was present in Ensuro 2.0) the validation, now it reverts if an upgrade changes the
currency.
```

We verified that the team fixed the issue as recommended.

File(s) affected: `contracts/PolicyPool.sol`

Description: The `PolicyPool` contract uses a UUPS proxy pattern, but the core economic parameter — the underlying ERC20 currency — is stored as an *immutable variable in the implementation contract*, not in proxy storage.

When upgrading via UUPS, the proxy simply delegates calls to the new implementation; thus, deploying a new implementation with a different immutable `_currency` causes the entire protocol to silently switch to a new underlying asset.

This produces several severe failures:

Existing assets become stranded
All USDC (or other old currency) held in the pool, eTokens, or PremiumsAccount remains in those contracts, but the upgraded implementation no longer references that token. All future operations use the *new* ERC20 instead, making old assets inaccessible except through custom rescue code.

Units of account become inconsistent
All storage variables (premium balances, SCR, exposures, surplus, deficit, etc.) were accumulated assuming the old currency. After upgrade, they are interpreted as amounts denominated in the *new* currency, breaking solvency assumptions.

Liquidity provider and policyholder balances become economically meaningless
LPs who deposited Token A may be forced to withdraw Token B after upgrade, with potentially different value, decimals, or liquidity.
Policy payouts, premiums, and SCR are now implicitly priced in a new unit, with no conversion or reconciliation.

The entire economic model (SCR, exposure, solvency, CoC, utilization) collapses
Even if the upgrade is unintentional, the math becomes meaningless because:

- stored values were calibrated to the old token,
- accounting and solvency are now interpreted in a new asset,
- and no migration / conversion is performed.

This creates a catastrophic mismatch that can lead to silent insolvency or loss of user funds.

Recommendation: In `PolicyPool._authorizeUpgrade(newImpl)`, ensure that the current currency matches `newImpl._currency`.

ENSR-6

`changeComponentStatus()` Allows Setting Inactive, Bricking Component and Causing Fund Lockup

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `de64e9dfa859e9acbeadccb021ee4e5c52b54566`.
The client provided the following explanation:

```
Fixed, now the only way to change the status to inactive is with removeComponent()
```

We verified that the team fixed the issue as recommended.

File(s) affected: `contracts/PolicyPool.sol`

Description: The `changeComponentStatus()` function is intended to transition components between `active`, `deprecated`, and `suspended` states. The `inactive` status is reserved to mean "not present in the pool" and should only be achievable through `removeComponent()`, which performs critical safety checks before removal.

However, `changeComponentStatus()` only validates that the current status is not `inactive` (line 464) but fails to validate that `newStatus` is not `inactive`. This allows a privileged caller to set any component to `inactive` status, bypassing all safety checks that `removeComponent()` enforces:

1. **For EToken**: No verification that `totalSupply() == 0`, allowing LP funds to remain locked
2. **For RiskModule**: No verification that `activeExposure == 0`, leaving active policies unresolvable
3. **For PremiumsAccount**: No verification that `purePremiums() == 0` and no cleanup of borrower relationships via `removeBorrower()`

Additionally, unlike `removeComponent()` which deletes the component from the `_components` mapping (line 450), `changeComponentStatus()` leaves the component in the mapping with `inactive` status, creating an inconsistent state.

Exploit Scenario:
1. PolicyPool has an active EToken with 1M USDC in LP deposits (`totalSupply = 1M`)
2. Governance accidentally calls `changeComponentStatus(eToken, ComponentStatus.inactive)` instead of following the proper deprecation flow
3. The component is immediately set to `inactive` status
4. All EToken operations now fail with `ComponentNotFoundOrNotActive` error
5. LPs cannot withdraw their 1M USDC as `withdraw()` requires component to be active or deprecated (line 541)
6. Funds are permanently locked unless governance can somehow restore the component status
7. For PremiumsAccount, the borrower relationships are not cleaned up, potentially blocking eToken operations

Recommendation: Add validation to prevent setting `newStatus` to `inactive` in the `changeComponentStatus()` function.

ENSR-7

`setExposureLimit()` Uses Strict Inequality, Preventing Limit Equal to Current Exposure

• Low Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `3336a62cd4f08d47c6ba742ee6208dc9273a381e` .
The client provided the following explanation:

```
Fixed and test-case added
```

We verified that the team fixed the issue as recommended.

File(s) affected: `contracts/PolicyPool.sol`

Description: The `setExposureLimit()` function contains an inconsistency in its validation logic compared to the `_changeExposure()` function. When setting a new exposure limit, line 779 uses strict inequality (`<`) to validate that the current active exposure is below the new limit:

```
require(exposure.active < newLimit128, ExposureLimitExceeded(exposure.active, newLimit128));
```

However, when creating new policies and checking if the exposure increase is allowed, `_changeExposure()` uses `<=` (line 770):

```
require(exposure.active <= exposure.limit, ExposureLimitExceeded(exposure.active, exposure.limit));
```

This inconsistency means:

- **Policy creation allows**: `active exposure == limit` (using `<=`)
- **Setting limit forbids**: `new limit == active exposure` (using `<`)

This creates an unnecessary edge case where governance cannot set the exposure limit to exactly match the current active exposure, even though this would be a valid and reasonable configuration. For example, if a RiskModule currently has exactly 1M in active exposure, governance cannot set the limit to 1M - they must set it to at least 1M + 1 wei.

Recommendation: Change the validation in `setExposureLimit()` from strict inequality to `<=` to align with `_changeExposure()` :

```
function setExposureLimit(IRiskModule rm, uint256 newLimit) external {
  Exposure storage exposure = _exposureByRm[rm];
  uint128 newLimit128 = newLimit.toUint128();
  require(exposure.active <= newLimit128, ExposureLimitExceeded(exposure.active, newLimit128)); // Change
< to <=
  emit ExposureLimitChanged(rm, exposure.limit, newLimit128);
  exposure.limit = newLimit128;
}
```

This allows governance to set the limit equal to the current active exposure, which is consistent with the policy creation logic and represents a valid operational state.

ENSR-8

Senior Loan Limit Check Uses Strict Inequality, Inconsistent with Junior Logic

• Low Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `56a909418468d1d2807d723c9d6674eb5e57a54f`.
The client provided the following explanation:

```
Fixed. We also created a value to store  _juniorEtk.getLoan(address(this)) , instead of calling it
several times. This improves code readability, size, and gas efficiency.
```

We verified that the team fixed the issue as recommended.

File(s) affected: `contracts/PremiumsAccount.sol`

Description: The `_borrowFromEtk()` function contains an inconsistency in how it validates loan limits for junior vs senior eTokens.

For the **junior eToken**, it uses `<=` when checking if a loan can be taken:

```
if (_juniorEtk.getLoan(address(this)) + borrow <= jrLoanLimit())
```

For the **senior eToken**, it uses strict inequality `<`:

```
if (_seniorEtk.getLoan(address(this)) + left < srLoanLimit())
```

This inconsistency means:

- **Junior eToken allows**: Borrowing up to and including the exact limit (`loan + borrow == limit`)
- **Senior eToken forbids**: Borrowing exactly to the limit (`loan + borrow == limit`)

This creates an unnecessary edge case where the senior eToken cannot be utilized to its full capacity. If the senior loan limit is 1M and current loan is 0, the PremiumsAccount can only borrow up to 999,999 wei less than 1M, leaving 1 wei of borrowing capacity perpetually unused.

Recommendation: Change the senior eToken validation from strict inequality to `<=` to align with junior eToken logic:

```
if (left != 0 && address(_seniorEtk) != address(0)) {
  // Consume Senior Pool only up to SCR
  if (_seniorEtk.getLoan(address(this)) + left <= srLoanLimit()) {  // Change < to <=
    left = _seniorEtk.internalLoan(left, receiver);
  }
}
require(left == 0, CannotBeBorrowed(left));
```

This allows borrowing exactly to the senior loan limit, maximizing capital efficiency and maintaining consistency across both eToken tiers.

ENSR-9
Precision Loss in Cost of Capital Calculation Due to Premature Division

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `8655db3dabed38905533e73b8489dfcbe629aa76`.
The client provided the following explanation:

```
Fixed. We also applied the same recommendation (divide later) to  jrInterestRate()  and
 srInterestRate()  in the same library.
```

We verified that the team fixed the issue as recommended.

File(s) affected: `contracts/Policy.sol`

Description: The Cost of Capital (CoC) calculation for both junior and senior tranches performs integer division before passing the result to `mulDiv()`, causing systematic precision loss that undercharges premiums. The expression `(rmParams.jrRoc * (expiration - start)) / SECONDS_PER_YEAR` performs integer division that truncates the result. This truncated value is then used in the `mulDiv()` operation, meaning the precision loss has already occurred and cannot be recovered.

For a 1-day policy with 5% annual junior return rate and 10,000 token SCR, the current implementation calculates `(0.05e18 * 86400) / 31536000 = 4.32e21 / 31536000 = 136986301369863` (truncated, losing ~13 wei), then `10000e18.mulDiv(136986301369863, 1e18) = 1369863013698630000` with ~0.01% precision already lost. A correct implementation maintaining full precision would calculate `10000e18.mulDiv(0.05e18 * 86400, 31536000 * 1e18)` without intermediate truncation.

The impact is small but systematic across all policies: 1 day policies experience `~0.01%` loss in CoC, while 30 day policies experience `~0.001%` loss, with shorter duration policies facing greater relative impact. Since `totalPremium = purePremium + ensuroCommission + totalCoc` (line 123), this CoC undercharging directly reduces the minimum premium required, systematically favoring policyholders at the expense of liquidity providers. For a protocol with $100M TVL and 10% annual rates, the estimated cumulative loss could be $1K-$10K per year in LP revenue.

Code reference:

```
// Calculate CoCs
minPremium.jrCoc = minPremium.jrScr.mulDiv(
    (rmParams.jrRoc * (expiration - start)) / SECONDS_PER_YEAR,  // Division truncates here
    WAD
);
minPremium.srCoc = minPremium.srScr.mulDiv(
    (rmParams.srRoc * (expiration - start)) / SECONDS_PER_YEAR,  // Same issue
    WAD
);
```

Where `SECONDS_PER_YEAR = 365 days = 31536000` (line 16) and `WAD = 1e18` (line 15).

Recommendation: Refactor the CoC calculation to maintain full precision by moving the division into the `mulDiv()` denominator:

```
// Calculate CoCs - maintain full precision
minPremium.jrCoc = minPremium.jrScr.mulDiv(
    rmParams.jrRoc * (expiration - start),
    SECONDS_PER_YEAR * WAD
);
minPremium.srCoc = minPremium.srScr.mulDiv(
    rmParams.srRoc * (expiration - start),
    SECONDS_PER_YEAR * WAD
);
```

ENSR-10
Signature Replay Across Riskmodules Due to Missing Riskmodule Address in Signed Data

• Low ⓘ Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `5612768d7404365e8c02c201bf189b514c85e724`.
The client provided the following explanation:

```
Fixed. We added the riskModule by changing the internalId field (96 bits) for a policyId field (256
bits) that includes both the riskModule address (in the first 160 bits) and the internalId (last 96
bits). It was better doing it that way, for gas efficiency and also because on the client side we
already have the policyId.
  As part of the change, we moved the functions related to assembling the policyid, or extracting the
riskModule or the internalId to the Policy library (they were in the PolicyPool contract), so the logic
is encapsulated there.
```

We verified that the issue is resolved by encoding `rm` into `policyId` (`rm+internalId`) and checking `Policy.extractRiskModule(policyId) == rm` in all signed underwriter flows, which cryptographically ties each signature to a single `RiskModule`.

File(s) affected: `contracts/underwriters/FullSignedUW.sol`

Description: The `FullSignedUW._checkSignature()` function verifies that a signer has the required role on a target RiskModule, but the signed data itself does not include the RiskModule address. This creates a potential for signature replay across different RiskModules when the same signer is authorized on multiple products. The signed data for a new policy includes `abi.encode(payout, premium, lossProb, expiration, internalId, params)` but omits the RiskModule address, meaning a signature created for RiskModule X could be replayed on RiskModule Y if the signer has roles on both.

The vulnerability requires specific conditions to be exploitable: (1) the same signer must be granted roles on multiple RiskModules (an operational choice), (2) the pricing parameters must be applicable to both products, (3) the `internalId` must not have been used on the target `RiskModule` yet, and (4) the mispriced policy must create economic advantage for the attacker. While `_checkSignature()` validates that `AccessManagedProxy(payable(rm)).ACCESS_MANAGER().canCall(signer, rm, requiredRole)` returns true, this only confirms the signer has the required role on the target RiskModule but does not prevent reuse of the same signed pricing data.

The impact is limited by several mitigations: different products typically use different signers as part of operational security, wrong pricing may not always favor the attacker (e.g., if flight insurance pricing is higher than auto insurance pricing), internal ID namespaces are per-RiskModule (reusing an already-used ID causes revert), and `PolicyPool` still enforces that premium meets minimum requirements. However, if these conditions align—same signer on multiple `RiskModules` with pricing that favors the policyholder on the wrong product—an attacker could observe a transaction in the mempool for `RiskModule` X, copy the `inputData` (pricing + signature), and submit it to `RiskModule` Y before the original transaction confirms, creating a mispriced policy.

Recommendation: Include the `RiskModule` address in the signed data to cryptographically bind each signature to a specific RiskModule:

```
function _checkSignature(address rm, bytes calldata inputData, uint256 inputSize, bytes4 requiredRole)
internal view {
  // Check length (now includes 32 bytes for address)
  uint256 inputLength = inputData.length;
  if (inputLength != (inputSize + 32 + SIGNATURE_SIZE)) revert InvalidInputSize(inputLength, inputSize +
32 + SIGNATURE_SIZE);

  // Decode and verify RM address is bound to signature
  address signedRm = abi.decode(inputData[0:32], (address));
  require(signedRm == rm, "RiskModule address mismatch");

  // Recover signer (hash includes the RM address now)
  bytes32 inputHash = MessageHashUtils.toEthSignedMessageHash(inputData[0:inputSize+32]);
  address signer = ECDSA.recover(inputHash, inputData[inputSize+32:inputLength]);

  // Check it has the permission in the RM
  (bool immediate, ) = AccessManagedProxy(payable(rm)).ACCESS_MANAGER().canCall(signer, rm,
requiredRole);
  require(immediate, UnauthorizedSigner(signer, requiredRole));
}
```

ENSR-11

Payout-worthy incident may not be paid if it occurs near the policy expiration

● Low ⓘ Acknowledged

Update

During the audit, we checked this with the team, and it appears to be intended behavior. If the team misses the expiration time, they will create a new policy with a short period to resolve it. However, there is a minor risk if the eTokens do not have sufficient liquidity at that time. Due to this, we mark this issue as acknowledged.

The following is the original statement from the team:

```
 expirePolicy  is intentionally permissionless, mainly as a guarantee for the LPs that their money will
eventually be unlocked.

That's a clear invariant — no payout can happen after expiration.

When we define the expiration time, we typically consider oracle delay and other possible operative
delays.

Given how much control we typically have on the policy creation and resolution, in case we miss the
expiration date, we can always create a new short policy and resolve it (unless the LPs removed all
their capital).
```

File(s) affected: `PolicyPool`

Description: In `_resolvePolicy()`, a policy can only be resolved if `payout == 0 || policy.expiration > block.timestamp`. Suppose an incident occurs one hour before the policy expires. Since the protocol is reliant upon signatures from privileged roles or oracles to determine the validity of a payout, there may not be enough time to acquire signatures and submit a transaction.

Recommendation: Since the protocol relies upon resolving expired policies in order to unlock capital for liquidity providers, it is not clear if this would be easily fixed at the contract level, and may be more easily resolved off-chain. In any case, this scenario should be considered for potential dispute-resolution.

ENSR-12
Use of Centralized Stablecoins May Allow External Freezing of Protocol Funds

● Informational ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We know about this and I think our users too, since we always make clear we operate in a highly
regulated environment where these discretional freeze might happen. We will add a section in our
documentation to stress this. Our near term plans are working only with USDC, for regulatory /
compliance reasons.
```

Description: The PolicyPool architecture allows any ERC-20 token to be used as the base currency.
If a **centralized stablecoin** (e.g., USDC, USDP, GUSD, or any token with an on-chain **blacklist**, **freeze**, or **seizure** mechanism) is used, the protocol becomes exposed to an external dependency risk.

Such tokens enable their issuer or regulator to **freeze balances at specific addresses**, including smart contracts. If *any* Ensuro contract (PolicyPool, PremiumsAccount, EToken, Reserve, Cooler, or associated vaults) is frozen:

- Transfers revert, blocking **withdrawals**, **premium refunds**, and **claim payouts**.
- Internal operations such as SCR locking/unlocking, ETK loan repayment, and liquidity rebalancing may fail.
- Liquidity providers may be unable to exit their positions.
- The system may become **partially or completely insolvent** despite being fully collateralized.

Freezing of *user* addresses also prevents the protocol from delivering payouts or withdrawals, creating contractual inconsistencies.

This introduces a **centralized kill-switch** that can halt or compromise the entire protocol.

Recommendation: If centralized stablecoins must be supported:
- Clearly document the inherent freezing risk.
- Implement monitoring to detect unexpected freezes.
- Provide emergency procedures or fallback behavior for frozen states.
- Consider support for multiple currencies to reduce single-token systemic risk.

ENSR-13
Missing Zero-Address Check for Optional Senior Etoken in `_borrowFromEtk()`

● Informational ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Check comment at
https://github.com/ensuro/ensuro/blob/7d9180df29bc238ceb194f9e05dbd1837c152b3e/contracts/PremiumsAccoun
t.sol#L434. If _seniorEtk == address(0) it will revert (and it's fine, since there's amount left).
Also, in practice we almost always have seniorEtk. So, for gas efficiency and code readability, we
prefer to keep it as it is. The only effect is it will revert without message, instead of reverting
with the CannotBeBorrowed custom error.
```

File(s) affected: `contracts/PremiumsAccount.sol`

Description: The `_borrowFromEtk()` function attempts to borrow funds from the senior eToken when the junior eToken cannot fulfill the full loan amount. However, the line `if (_seniorEtk.getLoan(address(this)) + left < srLoanLimit())` directly calls methods on `_seniorEtk` without checking if it is `address(0)`.

The senior eToken is an optional component in the PremiumsAccount design, as evidenced by:

- The upgrade validation that allows `address(_seniorEtk) == address(0)`
- The `repayLoans()` function explicitly checking `address(_seniorEtk) != address(0)` before using it

When `_seniorEtk` is `address(0)`, the call will fail with a low-level ABI decoding error rather than the more informative `CannotBeBorrowed` error that should be raised by `require(left == 0, CannotBeBorrowed(left))`.

Recommendation: Add a zero-address check before attempting to use the senior eToken, similar to the pattern used in `repayLoans()`:

```
if (left != 0 && address(_seniorEtk) != address(0)) {  // Add address(0) check
  // Consume Senior Pool only up to SCR
```

```
    if (_seniorEtk.getLoan(address(this)) + left < srLoanLimit()) {
      left = _seniorEtk.internalLoan(left, receiver);
    }
  }
  require(left == 0, CannotBeBorrowed(left));
```

This ensures that when the senior eToken is not configured, the function will properly revert with the `CannotBeBorrowed` error, providing clear feedback about the borrowing failure.

Auditor Suggestions

S1 Unresolved Todo

Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `522b40233452aafdcff0edb8d586b98fb4585244`.
The client provided the following explanation:

```
It was fixed before the audit report. TO DO removed.
```

File(s) affected: `PolicyPool.sol`

Description: There is an unresolved TODO in `PolicyPool.changeComponentStatus()`.

Recommendation: Either update the code or remove the TODO.

S2 Mismatch Between Inline Comments and Code

Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `65fed9d52ad3383720d49109ddc929db07a948de`.
The client provided the following explanation:

```
Fixed. Many of the comments were also fixed in a pre-report commit
522b40233452aafdcff0edb8d586b98fb4585244, that fixed documentation across all the contracts.
```

File(s) affected: `ETKLib.sol`, `Reserve.sol`, `PolicyPool.sol`, `PremiumsAccount.sol`, `EToken.sol`, `IEToken.sol`

Description: There are several inline comments that may be out-of-date or incorrect:

1. In `ETKLib.sol`, the comment block for `_mulDivCeil()` is the same as `_mulDiv()` above. It should additionally describe that it is "rounding up." Similarly, both `grow()` and `add()` have the same comment block.
2. In `ETKLib.sol`, the `toScaled*` function names do not seem to match the description. These functions "un-apply" the scale, so it may be more appropriate to be called `fromScaled*`. Similarly `EToken.scaledTotalSupply()` returns the unscaled total supply.
3. The comment above `Reserve.withdrawFromYieldVault()` should state "...and yieldVault().maxWithdraw() **>=** amount".
4. `ETKLib.add()`: The formulas in the comments of the two `add()` functions should be corrected. Instead of `newScale = scale * (1 + factor)`, it should be `newScale = scale + factor`.
5. In `PolicyPool.sol`, the two functions `_notifyReplacement()` and `_notifyCancellation()` have comments stating that they "never revert", however both have revert statements.
6. In the comment block of `PremiumsAccount._yieldEarnings()`, the comment "If positive, repays the loans and accumulates the rest in the surplus" appears to relate to an older version of the code.
7. In the comment block of `EToken.initialize()`, it is mentioned that both `maxUtilizationRate_` and `internalLoanInterestRate_` are in RAY. However, there is no usage of RAY elsewhere and the tests suggest that they should be in WAD (as the tests use the `_W()` function).
8. In `IEToken.sol`, the comment on `withdraw()`: "If the asked `amount` can't be withdrawn, it withdraws as much as possible" is not correct, since the function reverts if `amount > maxWithdraw`.
9. `Reserve.withdrawFromYieldVault()`: In the code comment, the part `Requires yieldVault().maxWithdraw() <= amount` is incorrect. It should be `amount <= yieldVault().maxWithdraw()`. Also, note that this is not explicitly validated but is likely derived from the nature of `yieldVault_.withdraw()` in the `_deinvest()` call. Additionally, using `type(uint256).max` as the amount input is also allowed.

Recommendation: Ensure that the inline comments align with the code.

S3 Interest Rate Optimization

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
It should be quite rare for a premiums account to have loans from the Sr. Pool. Furthermore,
implementing such a ""refinancing"" mechanism would imply repaying Sr. LP losses using Jr. LP funds and
generating a loss for Jr. LPs.
  Typically the interest rate paid to Junior LPs is higher than the one paid to Senior LPs, since they
take more risk.
  The way it is designed is to take as much debt with the junior LPs as possible and only then take
debt with senior LPs. For the repayment the order is the opposite, it pays the senior debt first and
only when fully repaid, repays the junior debt.
```

File(s) affected: `PremiumsAccount.sol`

Description: Suppose a `PremiumsAccount` has an existing loan from the senior EToken with a relatively high interest rate. Later, the junior EToken has more liquidity at a lower interest rate. It may be beneficial for the `PremiumsAccount` to be able to "refinance" by taking a loan from the junior to pay the senior.

Recommendation: It is not clear if this scenario would happen often (if at all). If so, consider periodically checking the junior EToken for liquidity if a loan exists to the senior EToken.

S4 General Suggestions and Best Practices

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
1. Validation not added to avoid increasing gas-cost in the usual case of amount != 0, the edge case
   will fail anyway
2. Although it's not explicitly checked, if expiration < start the premium calculation done on
   initialize would cause a revert, in lines like: minPremium.jrCoc =
   minPremium.jrScr.mulDiv(rmParams.jrRoc * (expiration - start), WAD * SECONDS_PER_YEAR)
```

File(s) affected: `contracts/ETKLib.sol`, `contracts/Policy.sol`, `contracts/Reserve.sol`

Description: The following is a list of general suggestions we have for this audit:

1. `ETKLib.discreteChange()`: Consider returning early when `scaledAmount.amount == 0` to avoid potential division by zero. It can return a `ScaledAmount` struct with the following fields: `amount: 0`, `scale: scaledAmount.scale`, and `lastUpdate: uint32(block.timestamp)`.
2. `Policy.initialize()`: Consider adding validation to ensure that `expiration > start`.

Recommendation: Consider applying the suggestions in the description section.

S5 Lack of Independent Borrower Whitelist Allows Full Compromise if `PolicyPool` Is Breached

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We think it's reasonable to accept PolicyPool as a single point of failure, since it's a key component
of our architecture. As such, we take special care in the upgrade configuration, with 4-day time lock
and monitoring of it.
  Anyway, if we would like to implement an additional whitelisting in the eTokens, we would be able to
do it without changing the eToken's code. With the AccessManagedProxy pattern we are using, we can add
a proxy-level access validation to those methods, just with configuration. We probably won't do it for
 lockScr  and  unlockScr , for gas-efficiency reasons, but we can do it for  internalLoan  since that
method isn't called so frequently.
  But if PolicyPool were compromised, a hacker could also produce severe damage (and economic profit
for him) by changing the ownership of the policies and resolving them, draining all the exposure.
```

Description: The `EToken` contract relies entirely on the `PolicyPool` to determine which addresses may become borrowers via `addBorrower()`. There is **no secondary authorization layer** within `EToken` itself.
Suppose the `PolicyPool` (or its upgrade authority) is compromised. In that case, an attacker can register arbitrary borrower addresses and exploit privileged functions such as `lockScr`, `unlockScr`, or `internalLoan`, potentially manipulating SCR or draining liquidity.

Because `EToken` exposes sensitive state-changing operations to borrower contracts, the absence of an independent borrower whitelist makes the system vulnerable to a **single point of failure** at the `PolicyPool` level.

Recommendation: Introduce an optional borrower whitelist inside the `EToken` contract. Borrower actions should require:

- `loan.lastUpdate != 0` (existing borrower tracking), and
- `borrowerWhitelist[msg.sender] == true`.

This provides an additional layer of defense: even if the `PolicyPool` is compromised, only pre-approved borrowers, typically `PremiumsAccounts`, can access `EToken` borrowing functionality, substantially reducing risk.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not pose an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Test Suite Results

We ran the tests using `npx hardhat test`.

Fix Review Update: during the fix review, we reran the tests on commit `5612768` using `USE_CUSTOM_ERRORS=Y npx hardhat python-coverage`.

```
Instrumenting for coverage...
=============================

> Cooler.sol
> ETKLib.sol
> EToken.sol
> interfaces/ICooler.sol
> interfaces/IEToken.sol
> interfaces/ILPWhitelist.sol
> interfaces/IPolicyHolder.sol
> interfaces/IPolicyPool.sol
> interfaces/IPolicyPoolComponent.sol
> interfaces/IPremiumsAccount.sol
> interfaces/IRiskModule.sol
> interfaces/IUnderwriter.sol
> LPManualWhitelist.sol
> Policy.sol
> PolicyPool.sol
> PolicyPoolComponent.sol
> PremiumsAccount.sol
> Reserve.sol
> RiskModule.sol
> underwriters/FullSignedUW.sol
> underwriters/FullTrustedUW.sol
```

```
Coverage skipped for:
=====================

> mocks/ForwardProxy.sol
> mocks/InterfaceIdCalculator.sol
> mocks/PolicyHolderMock.sol
> mocks/PolicyPoolComponentMock.sol
> mocks/PolicyPoolMock.sol
> mocks/ReserveMock.sol
> mocks/RiskModuleMock.sol

Compilation:
============

Nothing to compile

Network Info
============
> HardhatEVM: v2.26.3
> network:    hardhat

============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-8.4.1, pluggy-1.6.0
rootdir: /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full
configfile: pyproject.toml
testpaths: tests
plugins: timeout-2.4.0, cov-6.2.1
collected 130 items

tests/test_etoken.py ..........................s.                        [ 21%]
tests/test_policypool.py ....ss......................................... [ 57%]
.                                                                        [ 58%]
tests/test_premiumsaccount.py ....................................       [ 86%]
tests/test_riskmodule.py .................                               [100%]

=============================== warnings summary ===============================
.venv/lib/python3.12/site-packages/websockets/legacy/__init__.py:6
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/websockets/legacy/__init__.py:6: DeprecationWarning: websockets.legacy is deprecated; see
https://websockets.readthedocs.io/en/stable/howto/upgrade.html for upgrade instructions
    warnings.warn(  # deprecated in 14.0 - 2024-11-09

tests/test_policypool.py::test_expire_policy[prototype]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_expire_policy[prototype] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
    warnings.warn(

tests/test_policypool.py::test_expire_policy[ethereum]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_expire_policy[ethereum] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
    warnings.warn(

tests/test_policypool.py::test_expire_policies_in_batch[prototype]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_expire_policies_in_batch[prototype] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
    warnings.warn(

tests/test_policypool.py::test_expire_policies_in_batch[ethereum]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_expire_policies_in_batch[ethereum] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
```

```
    warnings.warn(

tests/test_policypool.py::test_replace_policy[prototype]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_replace_policy[prototype] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
    warnings.warn(

tests/test_policypool.py::test_replace_policy[ethereum]
  /home/appuser/workspace/projects/Ensuro/ensuro-ensuro-637ee48-github~full/.venv/lib/python3.12/site-
packages/_pytest/python.py:161: PytestReturnNotNoneWarning: Test functions should return None, but
tests/test_policypool.py::test_replace_policy[ethereum] returned <class 'dict'>.
  Did you mean to use `assert` instead of `return`?
  See https://docs.pytest.org/en/stable/how-to/assert.html#return-not-none for more information.
    warnings.warn(

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============ 127 passed, 3 skipped, 7 warnings in 467.76s (0:07:47) ============
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.



  Test add, remove and change status of PolicyPool components
    ✔ Change status and remove eToken (560ms)
    ✔ Change status and remove RiskModule (1294ms)
    ✔ Change status and remove PremiumsAccount (1307ms)

  Constructor validations
    ✔ Checks PolicyPool constructor validations (606ms)
    ✔ Checks EToken constructor validations (183ms)
    ✔ Checks PremiumsAccount constructor validations (119ms)
    ✔ Checks TrustfulRiskModule constructor validations (1451ms)
    ✔ Checks LPManualWhitelist constructor validations (53ms)

  Cooler
    ✔ Checks cooler cannot be initialized twice (1093ms)
    ✔ Has the right name and symbol
    ✔ It can change the cooldown period for a given eToken and each ETK has its own cooldown period
(429ms)
    ✔ It doesn't allow immediate withdrawals if the cooler is active and cooldown > 0 (376ms)
    ✔ It checks withdrawal schedule inputs (when >= cooldownPeriod, amount > 0, etk active, allowance)
(309ms)
    ✔ Checks withdrawal can't be executed before deadline or twice (367ms)
    ✔ Checks the earnings are redistributed to the rest of the LPs - YV Version (672ms)
    ✔ Checks the losses do impact the withdrawal result and funds can't be locked when scheduled (669ms)
    ✔ Checks it works well when scheduled infinite withdrawals exceed totalSupply (739ms)
    ✔ Checks withdrawals can be requested with permit (897ms)
    ✔ Checks the owner of the NFT receives the money (548ms)

  Etoken
    ✔ Refuses transfers to null address (814ms)
    ✔ Checks user balance
    ✔ Returns the available funds (80ms)
    ✔ Only allows PolicyPool to add new borrowers
    ✔ Can add new borrowers only once (864ms)
    ✔ Only allows PolicyPool to remove borrowers
    ✔ Can remove existing borrowers (842ms)
    ✔ Only can take loan on existing borrowers (176ms)
    ✔ Checks only borrower can lock and unlock capital (52ms)
    ✔ Only can repay loan on existing borrowers (216ms)
    ✔ Validates the parameter changes (199ms)
    ✔ Can remove the whitelist (973ms)
    ✔ Checks the whitelist belongs to the same pool (495ms)
    ✔ Can change the cooler and emits event (318ms)
    ✔ Checks the cooler belongs to the same pool (709ms)
    ✔ Only allows PolicyPool to call deposit
    ✔ Only allows PolicyPool to call withdraw
    ✔ Checks tokenInterestRate is zero when TS is zero (99ms)
    ✔ Can deposit to a different receiver (89ms)
    ✔ Can withdraw to a different receiver (252ms)
    ✔ Can withdraw using EIP-2612 approval (184ms)
    ✔ Can deposit to a different receiver - Whitelist version (1426ms)
    ✔ Can withdraw to a different receiver - Whitelist version (460ms)
    ✔ Allows setting whitelist to null (47ms)
    ✔ Checks funds can be unlocked with refund of CoC (945ms)
    ✔ Checks totalWithdrawable is zero when SCR > totalSupply (140ms)
    ✔ Can assign a yieldVault and rebalance funds there (337ms)
    ✔ LP cannot exit the pool before yieldVault losses are recorded (335ms)
    ✔ Can combines returns from locked SCR and from YV (489ms)
    ✔ Checks loans can be paid from the yieldVault if needed (333ms)
    ✔ Checks asset earning when totalSupply() == 0 (483ms)

  Test Implementation contracts can't be initialized
    ✔ Does not allow initialize PolicyPool implementation (134ms)
    ✔ Does not allow initialize EToken implementation (512ms)
    ✔ Does not allow initialize PremiumsAccount implementation (73ms)
    ✔ Does not allow initialize RiskModule implementation (868ms)
    ✔ Does not allow initialize LPManualWhitelist implementation

  Test Initialize contracts
    ✔ Does not allow reinitializing PolicyPool
    ✔ Does not allow reinitializing Etoken
    ✔ Does not allow reinitializing PremiumsAccount
    ✔ Does not allow reinitializing RiskModule
    ✔ Does not allow reinitializing Whitelist (174ms)

  Multiple policy expirations
Total gas used by individual expiration of 100 policies: 9469800n
Avg gas per policy expiration: 94698n
    ✔ Measure the gas cost of single policy expiration (20034ms)
Total gas used by multiple expiration of 100 policies: 3575181n
Avg gas per policy expiration: 35751n
    ✔ Measure the gas cost of multiple policy expiration (7139ms)
Total gas used by chunked expiration of 100 policies with chunksize=10: 4248570n
Avg gas per policy chunk: 424857n
Avg gas per policy expiration: 42485n
    ✔ Measure the gas cost of chunked multiple policy expiration (7260ms)
    - Measure what's the max number of policies that can be expired without exceeding the max gas
    ✔ Actually expires the policies (10243ms)
    ✔ Refuses to expire unexpired policies (301ms)

  Test pause, unpause and upgrade contracts
    ✔ Pause and Unpause PolicyPool (1074ms)
    ✔ Pause/Unpause and resolve policy with full payout (715ms)
    ✔ Pause/Unpause and expire/cancel policy (425ms)

  PoliyHolder policy creation handling
    ✔ Receiving with a functioning holder contract succeeds and executes the handler code (1675ms)
    ✔ Receiving with a holder that fails reverts the transaction (50ms)
    ✔ Receiving with a holder that fails empty reverts the transaction (49ms)
    ✔ Receiving with a holder that returns a bad value reverts the transaction (45ms)

  PolicyHolder resolution handling
    ✔ Resolving with a functioning holder contract succeeds and executes the handler code (371ms)
    ✔ Resolving with a holder that fails reverts the transaction (359ms)
    ✔ Resolving with a holder that fails empty reverts the transaction (355ms)
    ✔ Resolving with a holder that returns a bad value reverts the transaction (388ms)
    ✔ Resolving with a holder that doesn't implement the interface suceeds without executing the handling
code (386ms)
    ✔ Resolving with a holder that doesn't implement ERC165 succeeds without executing the handling code
(374ms)

  PolicyHolder replacement handling
    ✔ Replacing with a functioning holder contract succeeds and executes the handler code (418ms)
    ✔ Replacing with a functioning holder contract succeeds even if it spends a lot of gas (363ms)
    ✔ Replacing with a failing holder contract fails (572ms)

  PolicyHolder cancelation handling
    ✔ Canceling with a functioning holder contract succeeds and executes the handler code (294ms)
    ✔ Cancelling with a functioning holder contract succeeds even if it spends a lot of gas (339ms)
```

```
      ✔ Cancelling with a failing holder contract fails (503ms)

    PolicyHolder expiration handling
      ✔ Expiring with a functioning holder contract succeeds and executes the handler code (272ms)
      ✔ Expiring with a holder that reverts succeeds but doesn't execute the handler code (230ms)
      ✔ Expiring with a holder that spends a lot of gas succeeds but doesn't execute the handler code
(248ms)
      ✔ Expiring with a holder that spends few gas succeeds and executes the handler code (low gas
version)) (246ms)
      ✔ Expiring with a holder that reverts empty succeeds but doesn't execute the handler code (454ms)
      ✔ Expiring with a holder that returns a bad value succeeds (282ms)
      ✔ Expiring with a holder that doesn't implement the interface succeeds (264ms)
      ✔ Expiring with a holder that doesn't implement ERC165 succeeds (229ms)

    PolicyPool contract
      ✔ can change the treasury (659ms)
      ✔ can't change the treasury without permission, not even using multicall (266ms)
      ✔ can add components (425ms)
      ✔ Does not allow adding an existing component (404ms)
      ✔ Does not allow adding different kind of component (1086ms)
      ✔ Does not allow adding a component that belongs to a different pool (683ms)
      ✔ Adds the PA as borrower on the jr etoken (861ms)
      ✔ Removes the PA as borrower from the jr etoken on PremiumsAccount removal (1149ms)
      ✔ Adds the PA as borrower on the sr etoken (1010ms)
      ✔ Removes the PA as borrower from the sr etoken on PremiumsAccount removal (980ms)
      ✔ Does not allow suspending unknown components (329ms)
      ✔ Only allows riskmodule to create policies
      ✔ Fails if doing deposits to ZeroAddress receiver (538ms)
      ✔ Can deposit with permit, even if front-runned (261ms)
      ✔ Fails if doing withdrawals to ZeroAddress receiver, otherwise sends the money to receiver (279ms)
      ✔ Can do withdrawals on behalf of other user, if I have allowance (296ms)
      ✔ Can't change the exposure limit to something lower than the active exposure (1700ms)
      ✔ Can change the exposure limit to exactly equal the active exposure (61ms)
      ✔ Can't have two policies with the same internalId
      ✔ Only allows to resolve a policy once (172ms)
      ✔ Only allows riskmodule to resolve/cancel unexpired policies
      ✔ Does not allow a bigger payout than the one setup in the policy
      ✔ Can't expire policies when the pool is paused
      ✔ Can't replace resolved policies (157ms)
      ✔ Only RM can replace policies
      ✔ Rejects replace policy if the pool is paused
      ✔ Rejects cancel policy if the pool is paused
      ✔ Components must be active to replace policies (96ms)
      ✔ Components must be active or deprecated to cancel policies (165ms)
      ✔ Does not allow to replace expired policies
      ✔ Does not allow to cancel expired policies
      ✔ Does not allow to replace with expired policy
      ✔ Must revert if new policy premiums components are lower than old policy (423ms)
      ✔ Must revert if new policy have different start date (141ms)
      ✔ Should accept changes in the interest rate - Longer policy (490ms)
      ✔ Should accept changes in the interest rate - Shorter policy (529ms)
      ✔ Should accept changes in the payout and locked capital (829ms)
      ✔ Must revert if cancelation refunds exceed premium components (350ms)
      ✔ Can cancel policies giving zero refund (88ms)
      ✔ Can cancel policies and transfers refund to customer (557ms)
      ✔ Can cancel policies and refund pure premium even if purePremiums are exhausted (719ms)
      ✔ Allows borrowing exactly up to srLoanLimit (958ms)
      ✔ Only PolicyPool can call PA policyReplaced (614ms)
      ✔ Replacement policy must have a new unique internalId (38ms)
      ✔ Only PolicyPool can call PA policyCancelled (590ms)
      ✔ Can change the baseURI and after the change the tokenURI works (66ms)
      ✔ Initialize PolicyPool without name and symbol fails (455ms)

    Policy initialize
      ✔ Does not allow premium greater than payout
      ✔ Correctly collateralizes with jr etoken
      ✔ Correctly computes jr and sr CoC

    PremiumsAccount
      ✔ Checks the yieldVault is initialized as null and fails if invalid YV is set (1410ms)
      ✔ Checks the funds can be sent to the yieldVault and earnings properly recorded (1590ms)
      ✔ Updates accounting before withdrawal of won premiums (178ms)
```

```
Reserve base contract
  ✔ Checks the yieldVault is initialized as null and fails if invalid YV is set (583ms)
  ✔ Checks setYieldVault fails if receives a yieldVault with a different currency (112ms)
  ✔ Checks rebalance methods and recordEarnings fail if yieldVault not set
  ✔ Checks deposits into the YV and earnings recorded (794ms)
  ✔ Checks manual withdrawals from the YV (204ms)
  ✔ Checks funds are deinvested with YV is replaced (183ms)
  ✔ Checks funds are deinvested with YV is replaced - With unrecorded earnings (186ms)
  ✔ Checks funds are deinvested with YV is replaced - With failing vault (331ms)
  ✔ Checks cash outs from the YV (_transferTo) (413ms)
  ✔ Checks _transferTo works well with borderline input (216ms)
  ✔ Checks no redeem is called if YC without assets is disconnected

RiskModule contract
  ✔ It can change the partner wallet (1341ms)
  ✔ It can change the underwriter (58ms)
  ✔ The payer is always the caller and fails with ERC20 error if spending not approved (134ms)
  ✔ Does not allow wallet with zero address (169ms)
  ✔ If MaxUint256 is sent as premium, it's created with minimum premium (200ms)
  ✔ Can create several policies with a single call (764ms)
  ✔ Can create a lot of policies with a single call (12453ms)
  ✔ Can create policies using FullSignedUW (439ms)
  ✔ Fails if expiration is in the past (43ms)
  ✔ Fails if customer is ZeroAddress (44ms)
  ✔ Does not allow another payer - Even with old allowances (195ms)
  ✔ Reverts if new policy is lower than previous (671ms)
  ✔ It reverts if the replacement premium >= payout (49ms)
  ✔ Reverts if new policy is lower than previous with premium == MaxUint256 (180ms)
  ✔ Reverts if new policy has expiration in the past (48ms)
  ✔ It reverts if _activeExposure > exposureLimit (194ms)
  ✔ Rejects replace policy if the pool is paused (131ms)
  ✔ Should emit PolicyReplaced when policy is replaced (294ms)
  1) It can cancel a policy with pure premium and non accrued CoC
  ✔ It can cancel a policy with pure premium and custom CoC (230ms)

Storage Gaps
  ✔ EToken has a proper storage gap (365ms)
  ✔ LPManualWhitelist has a proper storage gap (792ms)
  ✔ PolicyPool has a proper storage gap (301ms)
  ✔ PolicyPoolComponent has a proper storage gap (368ms)
  ✔ PremiumsAccount has a proper storage gap (290ms)
  ✔ Reserve has a proper storage gap (302ms)
  ✔ RiskModule has a proper storage gap (327ms)

Supports interface implementation
  ✔ Checks PolicyPool supported interfaces (420ms)
  ✔ Checks EToken supported interfaces (84ms)
  ✔ Checks PremiumsAccount supported interfaces (660ms)
  ✔ Checks RiskModule supported interfaces (56ms)
  ✔ Checks LPManualWhitelist supported interfaces (42ms)
  ✔ Checks Cooler supported interfaces (61ms)

FullTrustedUW
  ✔ Checks it decodes the same input for priceNewPolicy (1622ms)
  ✔ Checks it decodes the same input for pricePolicyReplacement (2123ms)
  ✔ Checks it decodes the same input for pricePolicyCancellation (1512ms)
  ✔ Checks pricePolicyCancellation extracts the accrued interest when receices MaxUint256 (324ms)

FullSignedUW
  ✔ Checks it decodes the same input for priceNewPolicy (1823ms)
  ✔ Checks it decodes the same input for pricePolicyReplacement (773ms)
  ✔ Checks it decodes the same input for pricePolicyCancellation (546ms)
  ✔ Checks pricePolicyCancellation extracts the accrued interest when receices MaxUint256 (546ms)

Test Upgrade contracts
  ✔ Should be able to upgrade PolicyPool (1118ms)
  ✔ Shouldn't be able to upgrade PolicyPool changing the Currency (72ms)
  ✔ Should be able to upgrade EToken (76ms)
  ✔ Can upgrade EToken with componentRole (74ms)
  ✔ Should not be able to upgrade EToken with different pool (335ms)
  ✔ Should be able to upgrade PremiumsAccount contract (71ms)
```

```
    ✔ Can upgrade PremiumsAccount with componentRole (70ms)
    ✔ Should not be able to upgrade PremiumsAccount with different pool or jrEtk (805ms)
    ✔ Should be able to deploy PremiumsAccount without eTokens and upgrade to have them (1849ms)
    ✔ Should be able to upgrade RiskModule contract (1381ms)
    ✔ Can upgrade RiskModule with componentRole
    ✔ Should not be able to upgrade RiskModule with different pool or PremiumsAccount (924ms)
    ✔ Should be able to upgrade Whitelist (608ms)
    ✔ Can upgrade Whitelist with componentRole


  210 passing (3m)
  1 pending
  1 failing

  1) RiskModule contract
       It can cancel a policy with pure premium and non accrued CoC:
     AssertionError: expected 22512 to be close to 22567 +/- 50
      at Context.<anonymous> (test/test-risk-module.js:493:37)
```

Code Coverage

We ran the coverage using `npx hardhat coverage`, and the codebase shows strong test coverage. Line coverage is over 90%, and branch coverage is around 86%.

Fix Review Update: During the fix review, we reran the coverage on commit `5612768` using `USE_CUSTOM_ERRORS=Y npx hardhat python-coverage`, which also includes coverage from the Python tests. Overall, the coverage is very high, with branch coverage now at 96.65%.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	99.69	96.54	100	100	
Cooler.sol	100	96.15	100	100	
ETKLib.sol	100	93.75	100	100	
EToken.sol	100	99.15	100	100	
LPManualWhitelist.sol	100	86.36	100	100	
Policy.sol	100	90	100	100	
PolicyPool.sol	100	98.61	100	100	
PolicyPoolComponent.sol	100	90	100	100	
PremiumsAccount.sol	98.32	95.45	100	100	
Reserve.sol	100	97.37	100	100	
RiskModule.sol	100	92.31	100	100	
contracts/interfaces/	100	100	100	100	
ICooler.sol	100	100	100	100	
IEToken.sol	100	100	100	100	
ILPWhitelist.sol	100	100	100	100	
IPolicyHolder.sol	100	100	100	100	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
IPolicyPool.sol	100	100	100	100	
IPolicyPoolComponent.sol	100	100	100	100	
IPremiumsAccount.sol	100	100	100	100	
IRiskModule.sol	100	100	100	100	
IUnderwriter.sol	100	100	100	100	
contracts/underwriters/	100	100	100	100	
FullSignedUW.sol	100	100	100	100	
FullTrustedUW.sol	100	100	100	100	
All files	99.7	96.65	100	100	

Changelog

- 2025-12-03 - Initial report
- 2025-12-16 - Fix review report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

Quantstamp