# IMMUNEFI AUDIT

Immunefi / Hoenn

DATE June 11, 2025

AUDITOR **Neumo**, Security Researcher

REPORT BY **gmhacker**, Immunefi Head of Security

# ABOUT IMMUNEFI

Immunefi is the leading onchain security platform, having directly prevented hacks worth more than $25 billion USD. Immunefi security researchers have earned over $120M USD for responsibly disclosing over 4,000 web2 and web3 vulnerabilities, more than the rest of the industry combined.

Through Magnus, Immunefi delivers a comprehensive suite of best-in-class security services through a single command center to more than 300 projects — including Sky (formerly MakerDAO), Optimism, Polygon, GMX, Reserve, Chainlink, TheGraph, Gnosis Chain, Lido, LayerZero, Arbitrum, StarkNet, EigenLayer, AAVE, ZKsync, Morpho, Ethena, USDT0, Stacks, Babylon, Fuel, Sei, Scroll, XION, Wormhole, Firedancer, Jito, Pyth, Eclipse, PancakeSwap and many more.

Magnus unifies SecOps across the entire onchain lifecycle, combining Immunefi's market leading products and community of elite security researchers with a curated set of the very best security products and technologies provided by top security firms — including Runtime Verification, Dedaub, Fuzzland, Nexus Mutual, Failsafe, OtterSec and others.

Magnus is powered by Immunefi's proprietary vulnerabilities dataset — the largest and most comprehensive in web3, ensuring that security leaders and teams have the best possible tools for identifying and mitigating life threats before they cause catastrophic harm, all while reducing operational overhead and complexity.

Learn how you can benefit too at immunefi.com.

# TERMINOLOGY

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| LIKELIHOOD | IMPACT | | |
|---|---|---|---|
| | HIGH | MEDIUM | LOW |
| CRITICAL | Critical | Critical | High |
| HIGH | High | High | Medium |
| MEDIUM | Medium | Medium | Low |
| LOW | Low | | |
| NONE | None | | |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# EXECUTIVE SUMMARY

Over the course of 5 days in total, Hoenn engaged with Immunefi to review the smart contracts located in *hoenn-protocol/src/contracts*. In this period of time a total of 13 issues were identified.

SUMMARY

| | |
|---|---|
| Name | Hoenn |
| Audit Repository | https://github.com/hoenn-fi/hoenn-protocol/src/contracts |
| Audit Commit | e4edff3f8837217edc7a73bab6802e0082e121ef |
| Type of Project | Liquidity Provider, Staking |
| Audit Timeline | May 26th - May 30th |
| Fix Period | June 10th |

ISSUES FOUND

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical | 0 | 0 | 0 |
| High | 1 | 1 | 0 |
| Medium | 3 | 3 | 0 |
| Low | 2 | 2 | 0 |
| Insight | 7 | 5 | 2 |

CATEGORY BREAKDOWN

| | |
|---|---|
| Bug | 6 |
| Gas Optimization | 0 |
| Informational | 7 |

# FINDINGS

## IMM-HIGH-01

Interest index is not updated correctly

| Id | IMM-HIGH-01 |
|---|---|
| Severity | High |
| Category | Bug |
| Status | Fixed in be488db486e4f142d769387a09d18289178590e7 |

**Description**

Function `_accrueInterest` is called to accrue interest for a user. If the user has not yet borrowed `uTokens`, the function sets the user's interest index to the old value and returns early. It should instead set it to the updated interest index. Otherwise, the user will accumulate interest that should not exist.

This issue is especially problematic in the early days of the protocol, when the interest index is not updated frequently and larger amounts of interest may be incorrectly accrued.

Consider the following example:

- Vault creator creates Vault A. Initial values: `lastUpdateTimestamp = block.timestamp`, `interestIndex = RATE_PRECISION`.
- One week later, a user deposits collateral in Vault A. The user has zero debt, so the interest index is not updated. The user's `interestIndex` is set to `RATE_PRECISION`.
- In the same block, the user borrows `uTokens`. Debt is still zero, so the interest index is not updated. The user's `interestIndex` remains `RATE_PRECISION`.
- Also in the same block, the user borrows more `uTokens`, increasing their debt. The interest index is now updated. However, interest is accrued as if the user borrowed one week ago, resulting in incorrect interest charges.

**Recommendation**

Apply the following fix:

```
TypeScript
  function _accrueInterest(address _user) internal {
+         _updateInterestIndex();
+
        if (debtBalances[_user] == 0) {
            userInterestIndex[_user] = interestIndex;
            return;
        }

-       _updateInterestIndex();
...
```

### Proof of Concept

You can add the following test to the Vault.t.sol file to test this issue:

```
TypeScript
    function test_Interest_Accrued_Bug() public {
       uint256 depositAmount = 10 ether;
       vm.startPrank(alice);
       collateral.approve(address(vaultProxy), depositAmount);
       vm.expectEmit(true, true, true, true, address(vaultProxy));
       emit IVault.Deposited(alice, depositAmount);
       vaultProxy.deposit(depositAmount, alice);

       vm.warp(block.timestamp + 7 days);

       vaultProxy.mint(1 ether);

       // After minting 1 ether, the debt + interest should be 1 ether
       assert(vaultProxy.getTotalDebtWithInterest(alice) == 1 ether);

       vaultProxy.mint(1 ether);

       // After minting 1 more ether, the debt + interest should be 2 ether, but it's more
       assert(vaultProxy.getTotalDebtWithInterest(alice) > 2 ether);

       vm.stopPrank();
    }
```

## IMM-MED-01

No mechanism to handle bad debt

| | |
|---|---|
| Id | IMM-MED-01 |
| Severity | Medium |
| Category | Bug |
| Status | Fixed in<br>• c7c4de90069b5f7c0949e1f0bac6f1c9d4091f13<br>• 90b440ba6292dad65d3cc58bb76b96df981a6bbc<br>• e5fd287ea9f5af43a0b0cbe89efae8fd2ef4a7f9 |

**Description**

If a position is insolvent (i.e., the value of the debt exceeds the value of the collateral) or has a health factor slightly over 100%, liquidations are meant to reduce the risk it poses to the protocol. However, it's possible that after multiple liquidations, a position could be left with no collateral but still have remaining debt.

This is a dangerous state because the position would continue to accrue interest with no collateral to cover it. The current protocol has no mechanism to resolve or absorb this bad debt.

**Recommendation**

Implement an absorb mechanism to allow the protocol to recognize and absorb unbacked debt. This would enable it to close out positions that are beyond recovery and prevent ongoing accumulation of bad debt.

## IMM-MED-02

Liquidation should leave positions with better health, whenever possible

| Id | IMM-MED-02 |
|---|---|
| Severity | Medium |
| Category | Bug |
| Status | Fixed in 53117a1156668cd811a95783657969ec61b8b12f |

**Description**

If a position is not insolvent, liquidation should not leave it in a worse state of health than before. Otherwise, it increases the risk of repeated or cascading liquidations. In such cases, the protocol should prioritize the liquidator's share, even if it means reducing the protocol's own share.

Currently, the protocol always distributes both the liquidator and protocol shares as defined, even if doing so worsens the borrower's health or renders the position insolvent.

Example:

- Liquidation penalty: 10%
- Liquidator bonus: 5%
- Liquidation threshold: 80%
- User deposits 10 ETH and borrows 7 ETH
- Collateral price drops to 0.75 ETH → collateral value is now 7.5 ETH → LTV is above 80%, eligible for liquidation

If a liquidator repays 3.5 ETH (50% of the debt), they receive:
`3.5 * (1 + 10%) / 0.75 = 5.13 ETH` in collateral.
This leaves the borrower with less than 50% of their original collateral, worsening the position's health.

But if only the liquidator's share is paid:
`3.5 * (1 + 5%) / 0.75 = 4.9 ETH` in collateral is transferred.
This preserves more collateral for the borrower and improves post-liquidation health.

**Recommendation**

Consider reducing or skipping the protocol's share in cases where full liquidation payout would worsen the

borrower's position. This would help prevent unnecessary risk of cascading liquidations and better protect overall protocol solvency.

## IMM-MED-03

Risk of liquidations after unpausing

| | |
|---|---|
| Id | IMM-MED-03 |
| Severity | Medium |
| Category | Bug |
| Status | Fixed in 277f1f1858877e6612a8253f6036f9f041bb7a66 |

**Description**

When the protocol is paused, borrowers are at risk of being liquidated once operations resume. This happens because all functions that could improve a user's position—such as repayments or collateral top-ups—are also paused. During this time, interest continues to accrue, increasing user debt. Additionally, slashing events could occur while the protocol is paused, further worsening users' health factors.

As a result, borrowers may face immediate liquidation when the protocol is unpaused.

**Recommendation**

To mitigate this risk, consider the following possible solutions:

- **Allow repayments during pause**: This gives users the opportunity to reduce their debt and improve their health before the protocol is unpaused.
- **Implement a cooldown period after unpausing**: Introduce a delay before liquidations are enabled again, allowing borrowers time to act.

These are just a few possible solutions. The key goal is to prevent borrowers from being unfairly liquidated immediately after the protocol resumes.

## IMM-LOW-01

Lack of upper bound check in `setLiquidationPenalty`

| | |
|---|---|
| Id | IMM-LOW-01 |
| Severity | Low |
| Category | Bug |
| Status | Fixed in 434b489b4223e50717467c4c58ce6534df851d2c |

### Description

The function `setLiquidationPenalty` does not correctly validate the `_newPenaltyBps` input parameter. It checks that the value is greater than `liquidatorBonusBps`, but it does not ensure that it is `<= BASIS_POINTS_DIVISOR`. This could allow a penalty greater than 100% if the function is called with an incorrect value.

### Recommendation

Apply the following fix:

TypeScript

```
    function setLiquidationPenalty(
        uint256 _newPenaltyBps
    ) external onlyRole(TIMELOCK_ROLE) {
        require(
-           _newPenaltyBps > liquidatorBonusBps,
+           _newPenaltyBps > liquidatorBonusBps &&
+           _newPenaltyBps <= BASIS_POINTS_DIVISOR,
            "PENALTY_MUST_EXCEED_BONUS"
        );
        liquidationPenaltyBps = _newPenaltyBps;
        emit ConfigChanged("LiquidationPenalty", _newPenaltyBps);
    }
```

## IMM-LOW-02

`initialLiquidationThresholdBps` must be checked to be strictly greater than `initialLtvBps`

| | |
|---|---|
| Id | IMM-LOW-02 |
| Severity | Low |
| Category | Bug |
| Status | Fixed in e093f39e882e2217254a043efe13668a62253ccf |

### Description

When initializing a `Vault`, the `initialLiquidationThresholdBps` must not be equal to `initialLtvBps`. Otherwise, it would be possible to create positions that are immediately liquidatable. Since the functions `setLoanToValueRatio` and `setLiquidationThreshold` enforce that the threshold must be strictly greater than the LTV ratio, the same check should apply during initialization.

### Recommendation

Apply the following fix:

```
TypeScript
        require(
            params.initialLtvBps <= BASIS_POINTS_DIVISOR &&
-                params.initialLiquidationThresholdBps >= params.initialLtvBps &&
+                params.initialLiquidationThresholdBps > params.initialLtvBps &&
                params.initialLiquidationThresholdBps <= BASIS_POINTS_DIVISOR,
            "INVALID_LIQUIDATION_VARIABLES"
        );
```

## IMM-INSIGHT-01

Rebasing tokens cannot be used as collateral

| | |
|---|---|
| Id | IMM-INSIGHT-01 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Acknowledged |

**Description**

Some LRTs, such as `eETH`, are rebasing tokens. These tokens automatically adjust their balances to reflect yield, which breaks standard accounting assumptions. Using rebasing tokens as collateral can lead to serious issues in the protocol's accounting, including misreporting of collateral balances and accumulation of inaccessible funds.

If a rebasing token is used in a vault, the accrued collateral may become locked and unrecoverable without a contract upgrade.

**Recommendation**

Do not use rebasing tokens as collateral in any of the protocol's vaults. Always verify a token's behavior before creating a vault to ensure compatibility with the protocol's accounting model.

## IMM-INSIGHT-02

Risky exchange rate method

| | |
|---|---|
| Id | IMM-INSIGHT-02 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Acknowledged |

**Description**

The current implementation of `SimpleERC4626Adapter` is risky because it relies on `convertToAssets` to determine the value of the collateral. Depending on the underlying ERC4626 vault's implementation, this function can potentially be manipulated by users to return artificially low or high values, misrepresenting the true value of the collateral.

Using this adapter to price another collateral is equally unsafe, as it could compromise the integrity of the protocol's pricing mechanism.

**Recommendation**

Avoid using this adapter unless the underlying ERC4626 implementation is fully audited and known to be safe from manipulation. Consider alternative approaches that use more robust pricing mechanisms.

## IMM-INSIGHT-03

Interest is supposed to compound annually

| | |
|---|---|
| Id | IMM-INSIGHT-03 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Fixed in a7183a662384e7b892665320d660744f388992bd |

**Description**

According to the documentation, interest is intended to compound annually. However, the current implementation calculates interest based solely on the user's debt balance, meaning it does not compound annually as described.

**Recommendation**

Implementing true annual compounding interest would require major changes to the protocol. Currently, repayments are applied to interest first, then to the principal. To align with annual compounding, the system would need to reverse this: pay down the principal first, then the interest. Additionally, after one year from the initial borrow, the accumulated interest should be added to the debt balance so it begins accruing its own interest. The timer would also need to reset to begin tracking the next compounding period.

*Note*: I've assessed this as an Insight, assuming the inconsistency lies in the documentation rather than the code. If the protocol is truly intended to compound interest annually, this would be at least a Medium severity issue.

## IMM-INSIGHT-04

Unnecessary check in `_withdraw`

| Id | IMM-INSIGHT-04 |
| --- | --- |
| Severity | INSIGHT |
| Category | Informational |
| Status | Fixed in 6cafdd83430095c373ef84a63bb1d2952aa759dc |

**Description**

The `_withdraw` function in the `Vault` contract includes an unnecessary check that can be removed to improve readability.

TypeScript

```
        ...
        if (totalDebtWithInterest > 0) {
            uint256 remainingCollateralValue = remainingShares > 0
                ? IAdapterRegistry(adapterRegistry).getValueInETH(
                    address(collateralToken),
                    remainingShares
                )
                : 0;

            require(
                remainingCollateralValue > 0 || remainingShares == 0,
                "VALUATION_FAILED"
            );

            uint256 maxBorrowableAfter = (remainingCollateralValue *
                loanToValueRatioBps) / BASIS_POINTS_DIVISOR;
            require(
                totalDebtWithInterest <= maxBorrowableAfter,
                "INSUFFICIENT_COLLATERAL_RATIO"
            );
        }
        ...
```

We see in the final require that for it to pass, maxBorrowableAfter must be greater than zero, since totalDebtWithInterest is known to be greater than zero. This means remainingCollateralValue must also be greater than zero, as it directly affects maxBorrowableAfter.

In turn, for remainingCollateralValue to be greater than zero, remainingShares must also be greater than

zero.

Therefore, the earlier require that checks:

```
Unset
require(
    remainingCollateralValue > 0 || remainingShares == 0,
    "VALUATION_FAILED"
);
```

is unnecessary. If `remainingShares` is zero, the call would revert regardless due to the final `require`, and if it's non-zero, `remainingCollateralValue` must be positive. The second `require` already guarantees correctness.

### Recommendation

Apply the following fix:

```
TypeScript
-
-             require(
-                 remainingCollateralValue > 0 || remainingShares == 0,
-                 "VALUATION_FAILED"
-             );
```

## IMM-INSIGHT-05

Ensure collateral tokens have 18 decimals

| | |
|---|---|
| Id | IMM-INSIGHT-05 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Fixed in 90f02802399f5037ac9192439dd8ef299b5ab053 |

### Description

The protocol implicitly assumes that the collateral token always has 18 decimals. To enforce this assumption and improve security, the `initialize` function in the `Vault` contract should explicitly check the token's decimals. Relying on the fact that current mainstream LRTs use 18 decimals is not future-proof.

### Recommendation

Apply the following fix:

```
TypeScript
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
+import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
...
+       require(IERC20Metadata(params.collateralToken).decimals() == 18, "INVALID_DECIMALS");

        collateralToken = params.collateralToken;
        uToken = params.uToken;
```

## IMM-INSIGHT-06

Redundant check in Vault's `initialize` function

| | |
|---|---|
| Id | IMM-INSIGHT-06 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Fixed in 539aa0ec3bb4a337c879d911f0fc758a8bacc74b |

### Description

The bounds check in the `initialize` function of the `Vault` contract includes an unnecessary condition that can be removed for improved readability. The function ensures that `initialLiquidationPenaltyBps` is `>=` `initialLiquidatorBonusBps` and `<=` `BASIS_POINTS_DIVISOR`. Therefore, the additional check `initialLiquidatorBonusBps <= BASIS_POINTS_DIVISOR` is redundant.

### Recommendation

Apply the following fix:

```
TypeScript
require(
            params.initialLiquidationPenaltyBps >=
                params.initialLiquidatorBonusBps &&
+                params.initialLiquidationPenaltyBps <= BASIS_POINTS_DIVISOR
-                params.initialLiquidationPenaltyBps <= BASIS_POINTS_DIVISOR &&
-                params.initialLiquidatorBonusBps <= BASIS_POINTS_DIVISOR,
            "INVALID_PENALTY_BONUS"
        );
```

## IMM-INSIGHT-07

Usefulness of the constant `MAX_SINGLE_MINT`

| | |
|---|---|
| Id | IMM-INSIGHT-07 |
| Severity | INSIGHT |
| Category | Informational |
| Status | Fixed in 57ec8df1a706ad9de268793a9fed39142b4d4629 |

**Description**

Limiting the maximum single mint to 1M ETH using a constant ( `MAX_SINGLE_MINT`) is not effective.

**Recommendation**

Consider using a governance-configurable variable instead of a constant. If that's not feasible, reduce the constant to a more reasonable value.