yAudit

November 25, 2025

Prepared for
Goldilocks

Audited by
Panda
fedebianu

Goldilocks Goldilend update

Smart Contract Security Assessment

Contents

1 Review Summary **2**
- 1.1 Protocol Overview . . . 2
- 1.2 Audit Scope . . . 2
- 1.3 Risk Assessment Framework . . . 2
  - 1.3.1 Severity Classification . . . 2
- 1.4 Key Findings . . . 2
- 1.5 Overall Assessment . . . 2

2 Audit Overview **3**
- 2.1 Project Information . . . 3
- 2.2 Audit Team . . . 3
- 2.3 Audit Timeline . . . 3
- 2.4 Audit Resources . . . 3
- 2.5 Critical Findings . . . 5
- 2.6 High Findings . . . 5
  - 2.6.1 Wrong ternary precedence in `renew()` drops roll-over interest . . . 5
- 2.7 Medium Findings . . . 5
  - 2.7.1 Missing staleness checks on Pyth oracle . . . 5
  - 2.7.2 Incorrect calculation: unvested vs unclaimed value . . . 6
  - 2.7.3 Timelock bypass via `initializeGovParams` callable by `multisig` . . . 6
  - 2.7.4 `deposit()` enforces `liquidityThreshold` on pre-deposit supply . . . 7
  - 2.7.5 Hardcoded vesting duration mismatch . . . 7
- 2.8 Low Findings . . . 8
  - 2.8.1 `utilizationRatioMultiplier` scales the denominator and is not WAD-scaled . . . 8
  - 2.8.2 `renew()` cannot be called during the grace period . . . 9
- 2.9 Gas Savings Findings . . . 9
- 2.10 Informational Findings . . . 9
  - 2.10.1 `winner` variable is used outside its scope . . . 9
  - 2.10.2 Missing upper bound check on `unvestedWeights` . . . 10
- 2.11 Final Remarks . . . 10

1 Review Summary

1.1 Protocol Overview

Goldilocks Goldilend is a fixed-term NFT lending protocol that enables users to borrow assets using NFTs as collateral. The protocol features two main lending contracts: RebaseGoldilend (supporting Rebase Bera NFTs with HONEY tokens) and BeraBondGoldilend (supporting BeraBond NFTs with native BERA tokens). The system implements dynamic interest rates based on utilization ratios and loan duration, with upfront interest payment and comprehensive liquidation mechanisms.

1.2 Audit Scope

This audit covers two smart contracts totaling 408 lines of code across one and a half days of review.

```
src/core/goldilend/
├── GoldilendDebtAsset.sol
└── RebaseGoldilend.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	5
Low	2
Informational	2

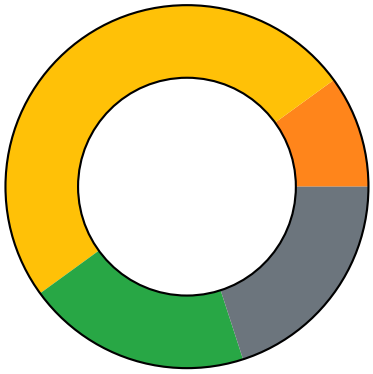

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The protocol exhibits a robust and improved architecture with well-structured NFT-backed lending mechanics. No critical bugs were discovered during the audit.

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

2 Audit Overview

2.1 Project Information

Protocol Name: Goldilocks
Repository: https://github.com/0xgeeb/goldilocks-core/tree/og/src/core/goldilend
Commit Hash: 2a658adcd27ef630b5b5c2f93b7626b2dda4c849
Commit URL:
https://github.com/0xgeeb/goldilocks-core/blob/2a658adcd27ef630b5b5c2f93b7626b2dda4c849/

2.2 Audit Team

Panda, fedebianu

2.3 Audit Timeline

The audit was conducted from October 30 to 31, 2025.

2.4 Audit Resources

Code repositories and documentation Additional resources: whitepaper, previous audits, etc.

Category	Mark	Description
Access Control	Good	Roles and permissions are defined with restricted privileged actions and clear ownership patterns.
Mathematics	Good	Rate and valuation calculations follow a coherent model using fixed-point arithmetic and established utilities.
Complexity	Good	The system is modular with responsibilities separated across components, maintaining manageable complexity.
Libraries	Good	Leverages established libraries like FixedPointMathLib for mathematical operations and OpenZeppelin contracts for standard functionality. Pyth libraries could be added.
Decentralization	Average	The introduction of a timelock improves decentralization by adding delay and transparency to governance changes, though control remains primarily centralized.
Code Stability	Good	The codebase remained stable throughout the review.
Documentation	Good	High-level architecture and interfaces are described. Additional detail would benefit operators and integrators.
Monitoring	Good	Core operations emit events that facilitate observability, accounting, and off-chain indexing.
Testing and verification	Average	Tests cover primary flows and behaviors, with room to broaden edge cases and scenario depth.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Wrong ternary precedence in `renew()` drops roll-over interest

Technical Details

In `renew()`, the expression that computes the additional interest for a renewal relies on a ternary without parentheses. In Solidity, arithmetic (`+`) has higher precedence than the ternary `? :`, so the whole expression is parsed incorrectly: the sum happens first, then the boolean comparison, and finally the ternary chooses only the top-up interest, discarding the interest for extending the existing principal.
This almost always evaluates the condition to true and sets `newInterest` to only the top-up interest, omitting the roll-over interest for the period from the old end to the new end on the already borrowed amount.

Impact

High. Borrowers underpay interest on renewals. The protocol loses the interest component for extending the existing debt.

Recommendation

Add parentheses so that only the second addend is conditional:

```
uint256 newInterest = _calculateInterest(userLoan.borrowedAmount, _outstandingDebt,
newEndDate - userLoan.endDate)
    + (newBorrowAmount > 0 ? _calculateInterest(newBorrowAmount, _outstandingDebt,
newDuration) : 0);
```

Developer Response

Fixed here.

2.7 Medium Findings

2.7.1 Missing staleness checks on Pyth oracle

Technical Details

`RebaseGoldilend` reads the `BERA` price via deprecated `IPythUpgradable.getPrice()` and directly uses it inside `_calculateFairValue()` without any freshness validation, so stale or low-confidence prices can drive collateral valuation. The interface exposes `publishTime` and `conf`, but they are unused.

Impact

Medium. Stale prices can overvalue or undervalue collateral, allowing excessive borrowing or inducing improper liquidations. Attackers can exploit delayed updates or poor-quality prices to manipulate borrow limits and auction outcomes.

Recommendation

Switch to `getPriceNoOlderThan()`. Optionally:

- Enforce a max relative confidence, e.g., `conf / |price| ≤ 5%`.
- Convert the price using `PythUtils.convertToUint()` from Pyth sdk.

Developer Response

Fixed here.

2.7.2 Incorrect calculation: unvested vs unclaimed value

Technical Details

The `_calculateFairValue` function in `RebaseGoldilend.sol` calculates the **unvested** (locked/not yet vested) value of an NFT, but should calculate the **unclaimed** (still in the NFT) value.
The function calculates how much value is still **locked in vesting schedule**, but ignores that vested-but-unclaimed rewards are equally valuable as collateral. This distinction is critical because:

1. When an NFT is transferred to the contract as collateral, the borrower **cannot claim rewards**
2. As time passes and rewards vest, they remain **locked in the NFT**
3. Vested rewards are just as much collateral as unvested rewards
4. The function treats them as if they have no value

The calculation progressively reduces collateral value over time, even though nothing has been claimed.

Impact

Medium. The most significant impact is on the `renew()` function. A large amount of tokens is vested at the cliff. If borrowing happens before the cliff, it's likely `renew()` will not have enough unvested tokens when the user wants to renew.

Recommendation

Calculate unclaimed value instead of unvested value.

Developer Response

Acknowledged. We decided that we only want to consider locked tokens as an extra precaution.

2.7.3 Timelock bypass via `initializeGovParams` callable by `multisig`

Technical Details

`changeGovParams()` correctly gates governance parameter changes behind the timelock, but `initializeGovParams()` sets the same parameters and is callable by `multisig` without any one-time guard. This lets `multisig` update governance parameters at any time, bypassing the timelock.

Impact

Medium. Undermines governance separation and delay guarantees. `multisig` can arbitrarily change critical constraints without a timelock.

Recommendation

Add a one-time guard to `initializeGovParams()`.

Developer Response

Fixed here.

2.7.4 `deposit()` enforces `liquidityThreshold` on pre-deposit supply

Technical Details

`deposit()` gates the minimum utilization check only if the current supply exceeds `liquidityThreshold`. The threshold gate is evaluated against the pre-deposit supply, while the utilization ratio uses the post-deposit supply. This allows a deposit that crosses the threshold to bypass the `minUtilization` constraint.

Impact

Medium. The minimum utilization invariant can be bypassed once the deposit pushes supply over the threshold.

Recommendation

Evaluate the threshold against the post-deposit supply:

```
    uint256 _ghoneySupply = GoldilendDebtAsset(glDebtAsset).totalSupply();
    uint256 newghoneySupply = _ghoneySupply + amount;
+   if (newghoneySupply > liquidityThreshold) {
-   if (_ghoneySupply > liquidityThreshold) {
        if (FixedPointMathLib.divWad(outstandingDebt, newghoneySupply) <
minUtilization) {
            revert MinUtilizationExceeded();
        }
    }
```

Developer Response

Fixed here.

2.7.5 Hardcoded vesting duration mismatch

Technical Details

The `_calculateFairValue` function hardcodes the vesting duration to 730 days on line 387:

```
uint256 vestingDuration = 730 days;
```

However, the `StreamingNFT` contract accepts `vestingDuration` as a constructor parameter, meaning different NFT collections can have different vesting schedules. The function queries the `StreamingNFT` contract for `cliffEndTimestamp`, `cliffUnlockAmount`, and `vestedRewards`, but incorrectly assumes all collections use a 730-day vesting period. When calculating the remaining unvested portion (lines 401-403):

```
uint256 elapsedPortion = FixedPointMathLib.divWad(timeSinceCliff, vestingDuration);
uint256 remainingPortion = 1e18 - elapsedPortion;
vest = FixedPointMathLib.mulWad(remainingPortion, vestedRewards);
```

The calculation uses a hardcoded value rather than the actual vesting duration from the streaming contract.

Impact

Medium.

Recommendation

Fetch the vesting duration dynamically from the `StreamingNFT` contract.

Developer response

Fixed here.

2.8 Low Findings

2.8.1 `utilizationRatioMultiplier` scales the denominator and is not WAD-scaled

Technical Details

The interest calculation in `_calculateInterest()` divides by `totalSupply * utilizationRatioMultiplier`. Placing the "multiplier" in the denominator means it arbitrarily rescales utilization in the opposite direction (higher multiplier → lower utilization), and it is not specified as a WAD-scaled parameter. This makes the parameter unintuitive and easy to misconfigure.

Impact

Low. Parameter semantics are ambiguous and can lead to unintended behavior (e.g., raising the multiplier decreases rates).

Recommendation

Applied multiplicatively to the utilization and make it WAD-scaled:

```
uint256 utilizationRatio = FixedPointMathLib.divWad(debt + borrowAmount,
GoldilendDebtAsset(glDebtAsset).totalSupply());
uint256 ratio = FixedPointMathLib.mulWad(utilizationRatio, utilizationRatioMultiplier) +
 interestPaymentPercentage;
```

Developer Response

The variable is WAD-scaled in this commit. Although we want to keep `utilizationRatioMultiplier` in the denominator as the variable is a constant. The idea is that it will initially be set to 2 and `interestPaymentPercentage` will be set to 0.75, which then means that the final ratio that gets applied to interest will be between 0.75 and 1.25.

2.8.2 `renew()` cannot be called during the grace period

Technical Details

`renew()` intends to allow renewals until `endDate + LOAN_GRACE_PERIOD`. However, when the caller is within the grace period where `block.timestamp >= userLoan.endDate`, the subtraction `userLoan.endDate - block.timestamp` underflows, reverting and preventing renewals even though the previous line permits them up to `endDate + LOAN_GRACE_PERIOD`.

Impact

Low. Borrowers cannot renew during the intended grace period.

Recommendation

```
1 -    if(userLoan.endDate - block.timestamp > renewMinDuration) revert
  InvalidRenew();
2 +    if(userLoan.endDate + LOAN_GRACE_PERIOD - block.timestamp >
  renewMinDuration) revert InvalidRenew();
```

Developer Response

Fixed here.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 `winner` variable is used outside its scope

Technical Details

In `closeAuction()` `winner` is declared inside the `else` branch and then referenced after the `if/else`, which is out of scope and causes a compilation error.

Impact

Informational.

Recommendation

Declare `winner` outside the conditional block.

Developer Response

Fixed here.

2.10.2 Missing upper bound check on `unvestedWeights`

Technical Details

The `changeUnvestedWeights` function (lines 514-527) allows setting collateral weights without bounds checking:

```
function changeUnvestedWeights(
    address[] calldata _beras,
    uint256[] calldata _weights,
    address[] calldata _streams
) public {
    if(msg.sender != multisig) revert NotMultisig();
    if(_beras.length != _weights.length) revert ArrayMismatch();
    if(_beras.length != _streams.length) revert ArrayMismatch();
    uint256 weightsLength = _weights.length;
    for(uint256 i; i < weightsLength; ++i) {
        unvestedWeights[_beras[i]] = _weights[i];  // No validation here
        streamingAddresses[_beras[i]] = _streams[i];
    }
}
```

Weights are used as percentage multipliers in the fair value calculation (line 408):

```
return FixedPointMathLib.mulWad(vest, formattedBeraPrice) * unvestedWeights[rebaseBera]
/ 100;
```

No validation prevents weights exceeding 100 (over-valuing collateral) or weights equal to 0 (disabling borrowing).

Impact

Informational.

Recommendation

Add validation to ensure weights are within acceptable bounds. You could use a base percentage value of 10_000 to have more flexibility.

Developer Response

Fixed here.

2.11 Final Remarks

The system has improved. The remaining items are of medium-to-low severity, apart from a high issue impacting protocol gains. Extend tests to cover edge timing (grace periods/auctions), threshold crossings, and oracle failure modes.