yAudit

November 25, 2025

Prepared for
Goldilocks

Audited by
fedebianu
HHK

Goldilocks Goldilend

Smart Contract Security Assessment

Contents

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 three smart contracts totaling approximately 622 lines of code across 3 days of review.

```
src/core/goldilend/
├── BeraBondGoldilend.sol
├── 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	2
🟧 High	0
🟨 Medium	6
🟩 Low	6
⬛ Informational	9

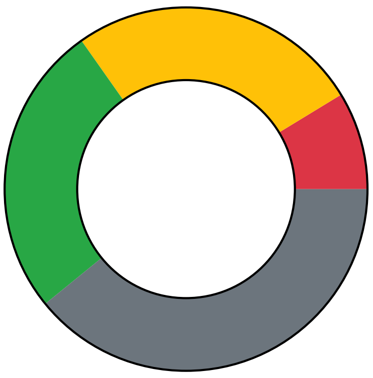

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The protocol demonstrates a sophisticated design for NFT-backed lending with innovative features like Token Bound Account integration. However, the audit revealed several critical vulnerabilities.

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
Commit Hash: 5875627dd0ad99d6a9bc620b474d33b7b5f61ff6
Commit URL: https://github.com/0xgeeb/goldilocks-core/commit/5875627dd0ad99d6a9bc620b474d33b7b5f61ff6

2.2 Audit Team

fedebianu, HHK

2.3 Audit Timeline

The audit was conducted from September 2 to 4, 2025.

2.4 Audit Resources

Code repositories Previous audit

Category	Mark	Description
Access Control	Good	Access control is well implemented.
Mathematics	Average	Some inconsistency in how interests are calculated upwards has been found.
Complexity	Average	The codebase has moderate complexity with clear separation between different lending contracts. However, significant code duplication between BeraBondGoldilend and RebaseGoldilend contracts increases maintenance burden and security risks.
Libraries	Good	Good use of established libraries like FixedPointMathLib for mathematical operations and OpenZeppelin contracts for standard functionality.
Decentralization	Low	The protocol relies entirely on a single multisig for protocol management, creating significant trust assumptions.
Code Stability	Good	Code wasn't updated during the audit.
Documentation	Average	Basic documentation exists with clear function names and comments, but lacks comprehensive documentation of the lending mechanics, interest calculation formulas, and risk parameters.
Monitoring	Good	Event emissions for key operations are in place.
Testing and verification	Average	Good test coverage with comprehensive unit tests, fuzz tests, and invariant tests. However, some critical bugs were not caught by existing tests, suggesting gaps in test scenarios.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

2.5.1 Attacker can steal all funds by repaying already repaid loans

The `repay()` function allows repayment of loans that have already been repaid, enabling attackers to repeatedly withdraw their NFT collateral and drain protocol funds.

Technical Details

The `repay()` function in both RebaseGoldilend and BeraBondGoldilend lacks validation to prevent repayment of already repaid loans.
The function caps the repayment amount to the borrowed amount:

```
if(repayAmount > userLoan.borrowedAmount) repayAmount = userLoan.borrowedAmount;
```

For already repaid loans where `borrowedAmount` is 0, this sets `repayAmount` to 0. The function then marks the loan as repaid and returns the NFT collateral, regardless of whether the loan was previously repaid.
An attacker can exploit this by: 1. Taking a loan and repaying it normally 2. Taking a new loan 3. Calling `repay()` with the old loan ID (and 1 wei payment to bypass the `if(msg.value == 0) revert InvalidAmount()` check in BeraBondGoldilend version) 4. Receiving the NFT collateral back for essentially free 5. Repeating steps 2-4 until all protocol funds are drained

Impact

Critical. Attackers can steal all protocol funds by repeatedly exploiting already repaid loans.

Recommendation

Add a check to prevent repayment of already repaid and liquidated loans.

Developer Response

Fixed in commit ceac8609a462989fee645eb030f542fc1b01ea90.

2.5.2 Protocol can be drained by renewing already repaid or liquidated loans

When a user repays a loan completely, the collateral NFT is returned and the loan is marked as repaid. However, `renew()` does not check if the loan has been repaid, allowing users to receive new funds without providing any collateral.

Technical Details

A user can repay a loan completely with `repay()` in both `BeraBondGoldilend` and `RebaseGoldilend` contracts. When the loan is fully repaid, the collateral NFT is transferred back to the user, and `repaid` is set to `true`.

However, `renew()` does not verify if the loan has been repaid or if the collateral still exists, allowing users to exploit this vulnerability.
The exploit works as follows:

1. User borrows funds with NFT collateral
2. User repays the loan completely, receiving back the NFT
3. User calls `renew()` on the same `loanId`
4. User receives new funds without providing any collateral
5. Repeat

This also applies to liquidation flow.

Impact

Critical. An attacker can steal all funds by renewing already repaid or liquidated loans without providing any collateral.

Recommendation

Verify in `renew()` that `userLoan.repaid == false` and `userLoan.liquidated == false`.

Developer Response

Fixed in commit 32980ec965b9a3e23d3631adf34c227f1e766066.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 Missing slippage protection in borrowing functions

The `borrow()` and `renew()` functions lack slippage protection, potentially causing users to receive less funds than expected due to changing interest rates.

Technical Details

The functions `borrow()` and `renew()` in both RebaseGoldilend and BeraBondGoldilend do not include slippage parameters.
Interest is paid upfront and can fluctuate based on: - Current utilization percentage - Protocol parameters that the team can update
Between transaction submission and execution, interest calculations may change significantly, causing users to receive less borrowed funds than anticipated. Users may only want to proceed if their net borrowing amount meets a minimum threshold.

Impact

Medium. Users may receive fewer borrowed funds than expected due to interest rate changes between transaction preparation, submission, and execution.

Recommendation

Add a `minOut` parameter to both functions and validate that the amount sent to users meets their minimum requirements.

Developer Response

Fixed in commit ad3b6f15f8dc9a64e60c8c99c4e10d7ebd1771f5.

2.7.2 Users can exploit loan renewal to pay lower interest rates

Users can repeatedly renew loans with short durations to pay significantly less interest than they would by borrowing for the full intended duration upfront.

Technical Details

The function `_calculateInterest()` calculates interest with an exponential duration and amount weight; longer durations, mainly, result in higher interest rates.
Users can exploit this by taking short-term loans and repeatedly calling `renew()` with short durations instead of borrowing for the whole intended period.
Example calculation:

```
Parameters: rate = 20e18, debt = 100e18, borrowAmount = 50e18, poolSize = 200e18, slope
= 2e18

//Single 100-day loan:
_calculateInterest(borrowAmount, debt, 100 days) //→ 4.6e18 interest

//Ten 10-day renewals:
_calculateInterest(borrowAmount, debt, 10 days) //→ 0.29e18 per renewal
//Total for 100 days: 0.29e18 × 10 = 2.9e18 interest

//Savings: 4.6e18 - 2.9e18 = 1.7e18 (37% reduction)
```

Users can automate this process with bots to continuously renew loans and minimize interest payments.

Impact

Medium. Users can exploit the interest calculation mechanism to pay significantly lower rates than intended, reducing protocol revenue and creating unfair advantages for sophisticated users.

Recommendation

Implement a fixed renewal fee or modify the interest calculation to account for cumulative loan duration, preventing users from gaming the system through frequent renewals.

Developer Response

Acknowledged, we believe this is not an issue. First, we have a minimum duration check, so there is a minimum length that the loan will always be on renewals, and users will always have to pay the interest on that duration up front. And it's okay if a user gets a lower rate by repeatedly renewing compared to doing one large loan, because the protocol is then taking on less risk, because it is able to update the valuations of the collateral in between renewals.

2.7.3 Share value increase can be sandwiched, diluting historical lenders

The `increaseglDebtAssetBacking()` function instantly increases share value, allowing attackers to sandwich the transaction and extract value intended for historical lenders.

Technical Details

The function `increaseglDebtAssetBacking()` can be called by the multisig to increase the share value of the debtAsset token. This function is typically called after liquidations when NFT collateral is sent to the multisig during `liquidate()` calls, and the team recovers proceeds from liquidating the collateral or using insurance funds.
The share value increase is applied instantly, but at a later time than the liquidation, creating two issues:

1. **Sandwich attacks**: An attacker can deposit large amounts just before the multisig calls `increaseglDebtAssetBacking()`, then immediately withdraw after the transaction to capture most of the value increase at the expense of historical lenders.
2. **Historical lender dilution**: Even without sandwich attacks, new depositors that deposited just after the liquidation but before the call to `increaseglDebtAssetBacking()` benefit from the share increase intended to reimburse historical lenders for liquidation losses, diluting the compensation meant for those who suffered the original losses.

Impact

Medium. The share value increase mechanism can be exploited through sandwich attacks and inherently dilutes compensation for historical lenders who suffered liquidation losses.

Recommendation

Consider implementing a streaming mechanism to distribute the backing increase over time, or restrict deposits/withdrawals during the backing increase process. Alternatively, implement a separate compensation mechanism that directly benefits affected lenders without diluting their recovery through new deposits.

Developer Response

Fixed in commits fa8dbaf5e95732b62fde5cc23f83ce644198c381, 1c3480ea42f6f47987310b48d51977a570e28ba0, 5fa6ec612908c143c628212c2a65547bc7c7c092, a73a57400c63248cbac6684c9f7eb45ed1d739cb and bdfaba489b74cb0fda9b0344beeea45b4a27c20a.

2.7.4 Users can claim yield from NFTs they no longer own

The `claimYield()` function allows users to claim rewards from NFTs they previously used as collateral but no longer own, as it fails to verify the loan status.

Technical Details

When borrowing in the BeraBondGoldilend and contract, the NFT ID used is added to `userTokenIds`.
When loans are repaid or liquidated in the BeraBondGoldilend contract, the NFT ID is not deleted from the `userTokenIds`. Instead, it remains in storage, and the loan struct is updated with `liquidated` or `repaid` set to `true` and `borrowedAmount` set to 0.
The `claimYield()` function allows users to claim rewards on Token Bound Accounts (TBAs) linked to BeraBonds used as collateral. However, it does not verify whether the associated loans are still active:
This creates a scenario where: 1. User A borrows against a BeraBond NFT 2. User A repays the loan and retrieves the NFT 3. User A transfers the NFT to User B 4. User B uses the same NFT as collateral for a new loan 5. User A can still call `claimYield()` to claim rewards from the NFT now owned by User B
Additionally, `borrow()` will always push the NFT ID to the `userTokenIds`, no matter if it has already been added in the past, leading to potential duplication of IDs inside the array.

Impact

Medium. Previous NFT holders can steal yield from current NFT owners who use the same ID.

Recommendation

Delete the NFT ID from the `userTokenIds` array when repaying or liquidating a loan.

Developer Response

Fixed in commit 00b5051aa34a8e82e9136af1a990d2423b6e3747 and 3a0e566db27f6ef69fe3b48f54473772e5f36a92.

2.7.5 Users can bypass the maximum loan duration through repeated `renew()` calls

The `renew()` function allows users to extend loans beyond the intended `maxDuration` limit by calling the function multiple times consecutively.

Technical Details

In both BeraBondGoldilend and RebaseGoldilend, the `renew()` function validates that `newDuration` falls within the `minDuration` and `maxDuration` bounds. However, it then adds this duration to the existing loan:

```
newUserLoan.endDate += newDuration
```

This allows users to batch multiple `renew()` calls, each adding up to `maxDuration` to their loan, effectively creating loans that last much longer than intended. A user could potentially extend their loan indefinitely by repeatedly calling `renew()` with valid duration parameters.

Impact

Medium. Users can bypass the maximum duration restriction, creating liquidity issues for depositors and circumventing protocol-intended loan limits.

Recommendation

Modify the `renew()` function to set the loan duration from the current timestamp rather than extending it:

```
newUserLoan.endDate = block.timestamp + newDuration
```

This ensures loans never exceed the maximum allowed duration.

Developer Response

Fixed in commit b74bb183a2eb902d9ce9a3e8adc77d7015752b98 and a2705e9e4e0df7944abbe8107854470a84f799e5.

2.7.6 Interest calculation on gross amount creates higher rates than expected

When a user borrows assets, the contract calculates and charges interest upfront, but it is incorrectly deducted upfront and sent to the multisig. This results in higher borrowing rates than expected.

Technical Details

Users can borrow assets with `borrow()` in both `BeraBondGoldilend` and `RebaseGoldilend` contracts.
The issue occurs because:

1. Interest is calculated on the full `borrowAmount` requested
2. Interest is deducted upfront and sent to multisig
3. User receives only `borrowAmount - interest`

This creates a discrepancy where the effective interest rate paid by users is higher than the nominal rate, as shown in the following example:

- User requests: 1000 BERA
- Interest calculated: 100 BERA (10% of 1000)
- User receives: 900 BERA (1000 - 100)
- Effective rate: 100/900 = 11.11% instead of 10%

Impact

Medium. Users pay higher effective interest rates than expected, especially problematic for high rates or long durations. The contract also incorrectly tracks pool utilization, including interests.

Recommendation

Transfer all the `borrowAmount` to the user and add the interest to it in storage. Check that `borrowAmount + interest` is not above the NFT fair value, which is already in place. Logic and checks involving `borrowAmount` should also be changed accordingly if needed.

Developer Response

Acknowledged.

2.8 Low Findings

2.8.1 Inconsistent behavior between RebaseGoldilend and BeraBondGoldilend contracts

Technical Details

There is inconsistent behavior between `RebaseGoldilend` and `BeraBondGoldilend`; some checks are done differently, which can lead to unexpected behavior.
For example inside `repay()`, inside the RebaseGoldilend the `repayAmount` is bounded to `outstandingDebt` while inside the BeraBondGoldilend it is bounded to `userLoan.borrowedAmount`.
This inconsistency may lead to different repayment calculations and behaviors between the two contracts, potentially confusing users and creating integration issues.

Impact

Low. While it doesn't seem to be creating major issues, this could lead to unexpected behavior and assumptions between the two contracts.

Recommendation

Standardize the logic between both contracts to ensure consistent behavior.
Consider using a base contract and inheriting from it as suggested in https://github.com/electisec/goldilocks-goldilend-report/issues/4.

Developer Response

Partially fixed in commit 0949d4b8ceeac999307e075577874bd8453d7867. Lack of Base contract can still lead to inconsistencies.

2.8.2 Zero `poolSize` allows unfair dilution of new depositors

Technical Details

Liquidations can lower the `poolSize` down all the way to zero. When depositing, the function `_glDebtAssetMintAmount()` returns the amount of shares depending on the current ratio

between `supply` and `poolSize`, unless one of the two is 0, in which case it will mint 1:1 shares per asset.
When withdrawing, it behaves differently - if the `poolSize` is 0, it will revert with a division by zero error. This shouldn't be a problem as the shares are worth zero, so there is no real interest for a user to withdraw 0 assets.
However, there is an issue for depositors. If the `poolSize` is 0, then they will mint 1:1 shares, increasing the backing for already existing shares, which will dilute the new depositors.
Example: - 100 shares exist, `poolSize` is 0 due to liquidations, the ratio is 0 - New user deposits 100 assets, receives 100 shares - Now there are 200 shares for 100 assets, so the ratio moved from 0 to 0.5 - Older users who experienced liquidations can withdraw up to 50 assets at the cost of the new lender

Impact

Low. It's unlikely that `poolSize` will drop all the way down to 0, and ideally, the team will be able to reimburse liquidations promptly. However, an unaware user may end up depositing and getting diluted.

Recommendation

Consider preventing deposits when `poolSize == 0` but `supply > 0`

Developer Response

Fixed in commit 801a80be3a1fc6305d0596a593a742209a70a5c6.

2.8.3 Inefficient yield claiming may cause gas issues or reverts

The `claimYield()` function loops through all user NFT IDs, including those from repaid or liquidated loans, causing unnecessary gas consumption and potential transaction failures.

Technical Details

The function `claimYield()` always iterates through the entire `userTokenIds` array.
This can be an issue as the array currently doesn't see its NFT IDs removed from it when a user repays his loan or gets liquidated. Additionally, a user may have yield available for only one NFT ID, and not all of them. Attempting to claim on all of them will incur extra gas costs and may even revert.

Impact

Low. Inefficient gas usage and potential denial of service for users with many historical loans.

Recommendation

Add parameters to allow partial claiming:

```
function claimYield(address[] memory rewardContracts, uint256 startIndex, uint256
endIndex) external {
    // Only process userTokenIds[startIndex:endIndex]
}
```

This enables users to claim yield for a subset of NFT IDs instead of all at once.

Developer Response

Fixed in commit 1786177e49e0fab0ef602fa02d22e78c4a4ba77d and 00b5051aa34a8e82e9136af1a990d2423b6e3747.

2.8.4 Pool size limit can be bypassed through repeated `renew()` calls

The `renew()` function only checks the new borrow amount against the pool size limit, allowing users to exceed the 10% borrowing restriction through multiple renewals.

Technical Details

The `renew()` function in both BeraBondGoldilend and RebaseGoldilend includes a check to prevent borrowing more than 10% of the pool size:

```
if(newBorrowAmount > _poolSize / 10) revert InvalidLoanAmount();
```

However, this check only considers the `newBorrowAmount` parameter and ignores the user's existing `borrowedAmount`. This allows users to bypass the borrowing limit by calling `renew()` multiple times with amounts just under the 10% threshold, accumulating a total borrowed amount that exceeds the intended limit.

Impact

Low. The pool size borrowing limit can be easily circumvented, undermining the protocol's risk management controls.

Recommendation

Modify the check to include the existing borrowed amount:

```
if(userLoan.borrowedAmount + newBorrowAmount > _poolSize / 10) revert InvalidLoanAmount
();
```

Developer Response

Fixed in commit 302a158c767e1afe68de86aeb816652adff9318e.

2.8.5 Inconsistent deadline enforcement between `repay()` and `renew()` functions

The `repay()` function blocks repayment after the grace period expires, whereas `renew()` allows loan extensions at any time, resulting in inconsistent behavior.

Technical Details

In both BeraBondGoldilend and RebaseGoldilend contracts, the `repay()` function includes a check that prevents repayment once `block.timestamp > userLoan.endDate + LOAN_GRACE_PERIOD`:

```
if(block.timestamp > userLoan.endDate + LOAN_GRACE_PERIOD) revert LoanExpired();
```

However, the `renew()` function lacks this same validation, allowing users to extend loans indefinitely regardless of how far past the deadline they are.
This inconsistency enables users to bypass the repayment deadline restriction by: 1. Waiting until after the grace period expires (when `repay()` would revert) 2. Calling `renew()` with minimal parameters (short duration, small amount) 3. Immediately calling `repay()` since the loan deadline has been reset

Impact

Low. Users can circumvent deadline restrictions through loan renewal, thereby undermining the intended enforcement of the grace period.

Recommendation

Add the same deadline check to `renew()` for consistency:

```
if(block.timestamp > userLoan.endDate + LOAN_GRACE_PERIOD) revert LoanExpired();
```

Alternatively, remove the check from `repay()` to allow both functions to operate without deadline restrictions.

Developer Response

Fixed in commit 59408f41680c11fa244296a60552f102e427b9fe.

2.8.6 `calculateInterest()` doesn't enforce the same checks as `borrow()`

Technical Details

- `_calculateInterest()` calculates interest based on the total `borrowedAmount`, but the last check against `fairValue` doesn't account for existing interest that has already been paid upfront, as it does in `borrow()`.
- `_calculateInterest()` doesn't check for `maxUtilization` as it does in `borrow()`.

Impact

Low. This can lead to incorrect information being passed to the function caller.

Recommendation

Modify `_calculateInterest()` in `RebaseGoldilend` as follows:

```diff
-       if(borrowAmount > fairValue || borrowAmount > poolSize - debt) revert BorrowLimitExceeded();
-       return _calculateInterest(borrowAmount, debt, duration);
+       uint256 _interest = _calculateInterest(borrowAmount, debt, duration);
+       if(borrowAmount + interest > fairValue || borrowAmount > poolSize - debt) revert BorrowLimitExceeded();
+       if(debt + borrowAmount > poolSize * maxUtilization / 100) revert MaxUtilizationExceeded();
+       return _interest;
```

Developer Response

Fixed in commit f1b650fe146b22e53740f7d105298ae74eacbd59.

2.9 Gas Savings Findings

2.9.1 Redundant liquidation status check in `liquidate()` function

Technical Details

The `liquidate()` function in BeraBondGoldilend and RebaseGoldilend checks both `liquidated` and `borrowedAmount == 0` to determine if a loan is valid for liquidation:

```
if((block.timestamp < userLoan.endDate + LOAN_GRACE_PERIOD || userLoan.liquidated ||
userLoan.borrowedAmount == 0) revert InvalidLoan();
```

When a loan is repaid or liquidated, the `borrowedAmount` is set to 0. Since `borrowedAmount == 0` already identifies loans that cannot be liquidated, checking `userLoan.liquidated` is redundant and wastes gas.

Impact

Gas savings.

Recommendation

Simplify the check by removing the redundant `liquidated` flag verification:

```
if((block.timestamp < userLoan.endDate + LOAN_GRACE_PERIOD || userLoan.borrowedAmount ==
 0) revert InvalidLoan();
```

Developer Response

Fixed in commit 71fe0f7dfc8cb153ff013818389af52cc472258e.

2.9.2 Inefficient use of `safeTransferFrom()`

Technical Details

The contract uses `safeTransferFrom()` to transfer NFTs during liquidation, but since the contract is already the owner of the NFT, received as collateral, using `transfer()` would be more gas efficient. `safeTransfer()` performs additional checks, verifying if the recipient is a contract by calling `onERC721Received()`, which are unnecessary as the multisig is a protocol known contract.

Impact

Gas savings.

Recommendation

Replace `safeTransferFrom()` with `transfer()` in the liquidation function since the contract is already the owner of the NFT.

Developer Response

Fixed in commit b32554a839dee30012476659371288fd58159b3b.

2.10 Informational Findings

2.10.1 Over-repayment results in asset loss

Technical Details

The `repay()` function in both BeraBondGoldilend and RebaseGoldilend allows users to send more assets than their outstanding loan balance.
While the function caps the repayment amount:

```
if(repayAmount > userLoan.borrowedAmount) repayAmount = userLoan.borrowedAmount;
```

The excess payment is neither refunded to the user nor used to increase the `debtAsset` backing, resulting in permanent loss of the overpaid assets.

Impact

Informational. Users who accidentally overpay lose their excess funds with no recovery mechanism.

Recommendation

Consider reverting when the payment exceeds the loan's `borrowedAmount` to prevent accidental asset loss.

Developer Response

Fixed in commit fff7ad21293a8c8ea542dedcde5b828e25003e46.

2.10.2 Remove division by `100` in `_calculateInterest()`

Technical Details

`_calculateInterest()` uses an unnecessary division by `100` at the end, which can be eliminated by appropriately setting the `protocolInterestRate` in 1e18 precision format instead of the current 1e20 format.

Impact

Informational.

Recommendation

Change the initialization of `protocolInterestRate` with a 1e18 precision format value, and remove the `/100` division at the end of `_calculateInterest`. This will make the interest calculation more transparent.

Developer Response

Fixed in commit a976a5f3c7515762ea14dc73a9689043b39d77ad.

2.10.3 `glDebtAsset` tokens do not accumulate yields in the contracts

Technical Details

The `glDebtAsset` tokens are minted when users deposit assets. However, when interests are accrued, these funds are sent directly to the multisig instead of being added to the pool. This means that while `glDebtAsset` tokens represent pool shares, they do not accumulate value over time as users would expect from a vault-like system.

Impact

Informational. Devs state that all revenue/interest goes to a multisig that then bribes a Berachain reward vault for the lending receipt token.

Recommendation

Clearly document the whole process.

Developer Response

Acknowledged, this process will be outlined in the documentation, UI, and marketing materials.

2.10.4 Refactor `initializeBeras()` and `initializeParameters()`

Technical Details

`initializeBeras()` and `initializeParameters()` contain duplicated code, as the same logic already exists in `changeValue()` and `changeLendingParams()`.

Impact

Informational.

Recommendation

Refactor `initializeBeras()` and `initializeParameters()`, eliminating the duplicated logic:

```
function initializeParameters(
    uint256 _protocolInterestRate,
    uint256 _minDuration,
    uint256 _maxDuration,
    uint256 _slope,
    uint256 _maxUtilization
) external {
    if (msg.sender != multisig) revert NotMultisig();
    if (parametersInitialized) revert AlreadyInitialized();

    setParameters(_protocolInterestRate, _minDuration, _maxDuration, _slope,
_maxUtilization);

    parametersInitialized = true;
}

function initializeBeras(address[] calldata _nfts, uint256[] calldata _nftFairValues)
external {
    if (msg.sender != multisig) revert NotMultisig();
    if (berasInitialized) revert AlreadyInitialized();

    changeValue(_nfts, _nftFairValues);

    berasInitialized = true;
    borrowingActive = true;
}
```

`setParameters()` and `changeValue()` must be made `public`.

Developer Response

Fixed in commit 6e764b7ae95d33a55956c2f2dcc73db6e3e4e2f0.

2.10.5 Remove unnecessary `onERC721Received()`

Technical Details

Both `BeraBondGoldilend` and `RebaseGoldilend` contracts implement `onERC721Received()`, which is unnecessary as the contracts only need to receive NFTs as collateral during `borrow()`. This theoretically allows users to accidentally send NFTs to the contract even though the use of `safeTranferFrom()`, potentially resulting in locked assets.

Impact

Informational.

Recommendation

Remove the `onERC721Received()` function entirely.

Developer Response

Fixed in commit 5efe6fe80f72f7edfe36f7ad460cec076980454f.

2.10.6 Remove unnecessary `unchecked` blocks

Technical Details

The contract uses `unchecked` blocks in loops, which are no longer necessary from Solidity version `0.8.22` as the compiler automatically handles loop variable increments optimization.

Impact

Informational.

Recommendation

Remove `unchecked` blocks in loops. This will improve code readability.

Developer Response

Fixed in commit 981ca4f4ac519b9c789063f7dc463f2ed12e4904.

2.10.7 Remove or use unused state variables

Technical Details

- Both `BeraBondGoldilend` and `RebaseGoldilend` contracts declare a `multisigClaims` state variable that is never used.
- In `RebaseGoldilend`, the state variable `timelock` is never used.

Impact

Informational.

Recommendation

Remove the variable from both contracts or use them.

Developer Response

Fixed in commit 63a94a3c4d05115ba52978c5f3a950cdcee4e7fc.

2.10.8 Avoid code duplication between `BeraBondGoldilend` and `RebaseGoldilend` contracts

Technical Details

Both `BeraBondGoldilend` and `RebaseGoldilend` contracts contain identical implementations for internal functions, variables, and some checks, leading to code duplication. While the main functions (borrow, repay, renew, liquidate) have similar logic but handle different collateral types, the internal mathematical and utility functions are completely duplicated.
This duplication means: - Security fixes for internal functions must be applied to both contracts - Bug fixes risk being missed in one of the contracts - Future development should be done twice with error risk

Impact

Informational.

Recommendation

Refactor the common code into a base contract that both `BeraBondGoldilend` and `RebaseGoldilend` can inherit from.

Developer Response

Acknowledged, thank you for this issue. These contracts previously did inherit from the same base contract, but I thought that, due to the native token debt asset and collateral-specific functions in BeraBondGoldilend, separate contracts were necessary.

2.10.9 Avoid mismatch between loan ID and NFT ID position in `userTokenIds`

Technical Details

When a user borrows assets, the contract creates a loan with ID `userLoansLength + 1` but stores the NFT ID in `userTokenIds[msg.sender].push(collateralNFTId)`. This creates a mismatch between loan IDs and their corresponding NFT positions in the array.

Impact

Informational.

Recommendation

Fix the mismatch by either using the same ID for both the loan and the NFT position.

Developer Response

Acknowledged.

2.11 Final Remarks

Major issues were identified during the audit and subsequently addressed. However, we suggest another audit to ensure production readiness. The test suite, while comprehensive, needs enhancement to catch edge cases and complex interaction scenarios that were missed in the initial testing. Additionally, the documentation requires significant expansion to provide clear explanations of the lending mechanics, interest rate formulas, risk parameters, and protocol governance structure.