yAudit Goldilocks Review

Review Resources:

- GoldiDOCS
- GoldiDOCS v1.2

Auditors:

- fedebianu
- Spalen

Table of Contents

Review Summary

Goldilocks

Goldilocks provides multiple DeFi products for Bearchain users. It is comprised of three main components: Goldiswap, a custom AMM for governance token LOCKS, which features two liquidity pools, one for the floor price at which LOCKS can be redeemed for stablecoin HONEY, and another for the market price at which LOCKS can be traded. Acquired tokens can be locked to enable voting and borrowing of HONEY from Goldiswap. Users are rewarded with PRG tokens, which can be burned for LOCKS at the floor price. The second component is Goldilend, a lending contract that allows users to lend their iBGT tokens and earn interest. Borrowers can use specific NFTs as collateral to borrow iBGT tokens and pay fixed interest for a fixed loan term. Idle iBGT tokens are deposited into Infrared to earn yield until they are borrowed again. The third component is Goldivault, a vault that allows users to split yield-bearing tokens into ownership and yield tokens, enabling users to trade and speculate on the future earnings of yield-bearing positions. The protocol is governed by Goldilocks DAO, which allows LOCKS holders to vote on proposals and change the protocol parameters via the standard timelock controller.

The contracts of the Goldilocks Repo were reviewed over 15 days. 2 auditors performed the code review between June 24 and July 12, 2024. The repository was under active development during the review, but the review was limited to the latest commit at the start. This was commit bde901524bd19d31912d3e67fa1fc9a81d779a9a for the Goldilocks repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src
├── core
│   ├── goldigovernance
│   │   ├── Goldigovernor.sol
│   │   ├── GovLocks.sol
│   │   └── Timelock.sol
│   ├── goldilend
│   │   └── Goldilend.sol
│   ├── goldiswap
│   │   ├── Goldilocked.sol
│   │   └── Goldiswap.sol
│   └── goldivault
│       ├── Goldivault.sol
│       ├── GoldivaultNegative.sol
│       ├── OwnershipToken.sol
│       └── YieldToken.sol
├── interfaces
│   ├── IGoldilend.sol
│   ├── IGoldilocked.sol
│   ├── IGoldiswap.sol
│   └── IGoldivault.sol
```

After the findings were presented to the Goldilocks team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Goldilocks and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Average	Adequate access control is present in user and admin-controlled functionality.
Mathematics	Average	There is no complex math. FixedPointMathLib library is used to scale numbers to avoid rounding errors.
Complexity	Low	The protocol builds multiple DeFi primitives inside one codebase, including custom AMM, NFT lending, locking, and voting. It will be deployed on Bearchain, which adds additional complexity, such as untested protocols, such as Infrared, for wrapping non-transferable native BGT tokens.
Libraries	Good	The protocol uses standard OpenZeppelin and Solady libraries.
Decentralization	Average	Most admin functions are behind the timelock controller, which is a good step toward decentralization. Some admin functions are behind the team's multisig.
Code stability	Good	No code changes were made during the audit.

Category	Mark	Description
Documentation	Average	All interface contracts provide NatSpec comments; internal documentation is complete but sometimes limited in explaining some of the protocol's inner workings.
Monitoring	Low	Events were emitted where applicable but are missing in most permission functions.
Testing and verification	Average	The codebase includes unit, fuzz, and invariant testing, which are well organized. Invariant tests could be improved by defining more contract invariants, adding more function selectors, and limiting the number of reverts.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - First lock advantage if `poolSize` is initialized

When a user locks his `iBGT`, they receive `GiBGT` in exchange. However, the calculation of `mintAmount` is wrong if `poolSize` is initialized in `initializeParameters()`.

Technical Details

A user can lock his `iBGT` with `lock()`. The function calculates the `GiBGT` to mint to the user by calling `_GiBGTMintAmount()`. The first user receives the amount of `GiBGT` equal to the locked amount of `iBGT`, because `totalSupply` is zero.

However, if `poolSize` is set during `initializeParameters()`, the next users, locking the same amount, will receive a smaller amount of `GiBGT`, as shown in the following PoC, where they receive ~1/100 of first user's `GiBGT`:

```
function testPoC() public {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");
    address carol = makeAddr("carol");

    vm.startPrank(alice);
    deal(address(ibgt), alice, txAmount);
    ibgt.approve(address(goldilend), txAmount);
    goldilend.lock(txAmount);

    console2.log("alice GiBGT balance", goldilend.balanceOf(alice));

    vm.startPrank(bob);
    deal(address(ibgt), bob, txAmount);
    ibgt.approve(address(goldilend), txAmount);
    goldilend.lock(txAmount);

    console2.log("bob GiBGT balance  ", goldilend.balanceOf(bob));

    vm.startPrank(carol);
    deal(address(ibgt), carol, txAmount);
    ibgt.approve(address(goldilend), txAmount);
    goldilend.lock(txAmount);

    console2.log("carol GiBGT balance", goldilend.balanceOf(carol));
}
```

Test results:

```
Ran 1 test for test/unit/Goldilend.t.sol:UnitGoldilendTest
[PASS] testPoC() (gas: 815281)
Logs:
  alice GiBGT balance 10000000000000000000
  bob GiBGT balance   99009900990099000
  carol GiBGT balance 99009900990099000
```

Impact

High. The first user that locks `iBGT` in the contract will receive a higher amount of `GiBGT` if `poolSize` is set during initialization.

Recommendation

Remove the possibility to set a starting `poolSize` during initialization. You can always lock any amount of `iBGT` after the contract initialization.

Developer Response

Fixed here

2. High - Collateral NFT can get locked in `Goldilend`

User loans are fetched from the array, which can lead to out-of-gas exceptions.

Technical Details

To repay their debt, the user must specify `userLoanId`. The `repay()` function uses `_lookupLoan()` to loop through all loans created by the user. If the user has made too many loans, this loop could revert with an out-of-gas exception. This will lock the user in a position where they cannot repay their loan.

The protocol won't be able to liquidate this loan because the `liquidate()` function also uses the same lookup function, leading to locked NFTs and bad debt for the protocol.

This scenario is possible for regular users because NFT holders will probably be long-term holders. By randomly checking the Bong Bears collection, Bong Bear #28 has had the same owner for the last three years, Bong Bear #62 also three years, and Bong Bear #01 2 years. Malicious users can create enough small loans to fill the array, take a big loan, and leave the protocol with locked NFTs and bad debt.

Impact

High. If the user creates too many loans, the NFT will get locked because of an out-of-gas exception.

Recommendation

Delete repaid and liquidated loans and use events to get this data. If loan data must stay on-chain, start the lookup loop from the last item in the array because the first ones are most likely already repaid. Additionally, limit the maximum number of active loans a user can have to prevent the loan search loop from ending with an out-of-gas exception.

Developer Response

Fixed here

3. High - Users can borrow above the collateral NFT fair value

Users can borrow above the collateral NFT fair value, leaving the protocol in a high bad debt position.

Technical Details

Using the `Goldilend` contract, users can borrow `iBGT` against NFTs as collateral. NFT fair value is defined by GoldilocksDAO and calculated using the function `_calculateFairValue()`. In the borrowing flow, it is validated that the user-defined `borrowAmount` is below the collateral NFT value. The problem is that this calculation doesn't consider the interest the user will have to pay for the borrowed amount. This leaves the option for the user to take the maximum fair value for the maximum duration, which will end up with the loan position being worth more than the fair value of the collateral NFT.

Bad debt will occur if the user doesn't repay their loan. To liquidate this position, liquidation must pay the borrowed amount plus interest, which can exceed the fair value and possibly the value of the NFT, making it unfavorable to liquidate.

The following code shows the user borrowing more than the fair value of their NFT. The borrowed amount is 165% of the collateral's fair value, putting the protocol in a high bad debt position.

```
function test_borrowAboveFairValue() public dealUseriBGT dealUserBeras {
  address[] memory nfts = new address[](1);
  nfts[0] = address(bondbear);
  uint256 fairValue = goldilend.getFairValues(nfts);

  goldilend.borrow(fairValue, goldilend.maxDuration(), address(bondbear), 1);
  Goldilend.Loan memory userLoan = goldilend.lookupLoan(address(this), 1);
  console2.log("borrowedAmount: ", userLoan.borrowedAmount);
  console2.log("fairValue: ", fairValue);
  assertGt(userLoan.borrowedAmount, fairValue, "borrowedAmount <= fairValue");
}
```

```
[PASS] test_borrowAboveFairValue() (gas: 811952)
Logs:
  borrowedAmount:  8250000000000000000
  fairValue:  5000000000000000000
```

Impact

High. Users borrowing above NFT fair value will leave the protocol with bad debt.

Recommendation

Borrowing interest should be paid at the start of borrowing to limit the accumulation of bad debt. Alternatively, add the borrowing interest to the borrowing amount before verifying that the user cannot borrow above the NFT fair value.

Developer Response

Fixed here

4. High - `stir()` could be vulnerable to sandwich attacks

`stir()` offers the ability to mint `LOCKS` at the floor price by burning `PORRIDGE`. However, the market price falls when this happens, opening a path for sandwich attacks.

Technical Details

`stir()` transfers the cost of `LOCKS` at the floor price to `goldiswap` and then calls `porridgeMint()`, which increases the `fsl` and mints `LOCKS` tokens to the user.

However, by only incrementing the `fsl`, the market price of `LOCKS` decreases, creating an opportunity for sandwich attacks against the protocol. Unlike user sales, which are protected from such price manipulations by the slippage protection mechanism in `sell()`, `stir()` remains vulnerable.

This attack becomes possible when the market price drop exceeds the 5% sell tax, as shown in the following PoC:

```
function testPoC() public dealStakeLocks {
    address attacker = makeAddr("attacker");
    address user = makeAddr("user");
    uint256 amount = 1_000_000 ether;

    deal(address(honey), user, type(uint256).max);
    deal(address(goldilocked), user, type(uint256).max);

    deal(address(goldiswap), attacker, amount); // ~18000$

    vm.startPrank(attacker);
    honey.approve(address(goldiswap), type(uint256).max);
    goldiswap.approve(address(goldiswap), type(uint256).max);

    vm.startPrank(user);
    honey.approve(address(goldilocked), type(uint256).max);
    goldiswap.approve(address(goldiswap), type(uint256).max);

    console.log("\nBefore");
    console.log("floor price\t", goldiswap.floorPrice());
    console.log("market price\t", goldiswap.marketPrice());
    console.log("honey balance\t", honey.balanceOf(address(goldiswap)) / 1e18);

    console.log("\n  Attacker balances:");
    console.log("honey balance\t", honey.balanceOf(attacker) / 1e18);
    console.log("locks balance\t", goldiswap.balanceOf(attacker) / 1e18);

    vm.startPrank(attacker);
    goldiswap.sell(amount, 0);

    console.log("\nAfter sell");
    console.log("floor price\t", goldiswap.floorPrice());
    console.log("market price\t", goldiswap.marketPrice());
    console.log("honey balance\t", honey.balanceOf(address(goldiswap)) / 1e18);
```

```
    console.log("\n  Attacker balances:");
    console.log("honey balance\t", honey.balanceOf(attacker) / 1e18);
    console.log("locks balance\t", goldiswap.balanceOf(attacker) / 1e18);

    vm.startPrank(user);
    goldilocked.stir(amount * 6); // ~34863$

    console.log("\nAfter stir");
    console.log("floor price\t", goldiswap.floorPrice());
    console.log("market price\t", goldiswap.marketPrice());
    console.log("honey balance\t", honey.balanceOf(address(goldiswap)) / 1e18);

    console.log("\n  Attacker balances:");
    console.log("honey balance\t", honey.balanceOf(attacker) / 1e18);
    console.log("locks balance\t", goldiswap.balanceOf(attacker) / 1e18);

    vm.startPrank(attacker);
    goldiswap.buy(amount, type(uint256).max);

    console.log("\nAfter buy");
    console.log("floor price\t", goldiswap.floorPrice());
    console.log("market price\t", goldiswap.marketPrice());
    console.log("honey balance\t", honey.balanceOf(address(goldiswap)) / 1e18);

    console.log("\n  Attacker balances:");
    console.log("honey balance\t", honey.balanceOf(attacker) / 1e18);
    console.log("locks balance\t", goldiswap.balanceOf(attacker) / 1e18);
}
```

Test results show that 33 `HONEY` can be drained from the protocol in the best case. With a stir amount twice as large, the amount of `HONEY` stolen escalates to 945. When the stir amount is ten times greater, it increases to 5461.

```
Ran 1 test for test/unit/Goldilocked.t.sol:UnitGoldilockedTest
[PASS] testPoC() (gas: 3443889)
Logs:

Before
  floor price    5810526315789473
  market price   19268478085335537
  honey balance  400000

  Attacker balances:
  honey balance  0
  locks balance  1000000

After sell
  floor price    5814202840837007
  market price   18351286481804873
  honey balance  382144

  Attacker balances:
  honey balance  17855
  locks balance  0

After stir
  floor price    5814202840837007
  market price   17392556743556686
  honey balance  417030

  Attacker balances:
  honey balance  17855
  locks balance  0

After buy
  floor price    5814202840837007
  market price   18161283411566451
  honey balance  434798
```

```
Attacker balances:
honey balance  33
locks balance  1000000
```

Impact

High. `HONEY` can be extracted from the protocol.

Recommendation

Prevent the market price to decrease more than 5% when `stir()` is called:

```
     function porridgeMint(address to, uint256 amount, uint256 cost) external {
         if (msg.sender != goldilocked) revert NotGoldilocked();
+        uint256 markekPriceBefore = _marketPrice(fsl, psl, totalSupply());
         fsl += cost;
         _mint(to, amount);
+        uint256 marketPriceAfter = _marketPrice(fsl, psl, totalSupply());
+        if (marketPriceAfter < markekPriceBefore * 95 / 100) revert ExcessiveSlippage();
     }
```

This check protects one transaction at time. However, the likelihood of multiple `stir()` calls occurring within the same block is low.

Developer Response

Fixed here

Medium Findings

1. Medium - `castVoteBySig()` incorrectly uses `msg.sender` to cast a vote instead of the signer

`castVoteBySig()` is used to vote on behalf of a signer. However, the voter is incorrectly set as `msg.sender`.

Technical Details

At the end of the `castVoteBySig()` there is a `_castVoteInternal()` call that finalize the vote. However, the voter parameter is set as `msg.sender` and not as the derived `signatory` meaning that the sender gives his votes for signed support instead of signer votes.

A user can use `msg.sender` votes to vote for a proposal and then vote again with `castVote()` or `castVoteWithReason()`.

Impact

Medium. Anyone who calls `castVoteBySig()` will lose their votes, while the signer can still vote.

Recommendation

Change the voter to `signatory`:

```
   /// @notice Cast a vote for a proposal by signature
   /// @dev Accepts EIP-712 signatures for voting on proposals
   function castVoteBySig(uint256 proposalId, uint8 support, uint8 v, bytes32 r, bytes32
s) external {
     bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH,
keccak256(bytes(name)), _getChainId(), address(this)));
     bytes32 structHash = keccak256(abi.encode(BALLOT_TYPEHASH, proposalId, support));
     bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator,
structHash));
     address signatory = ecrecover(digest, v, r, s);
     if(signatory == address(0)) revert InvalidSignature();
-    emit VoteCast(signatory, proposalId, support, _castVoteInternal(msg.sender,
proposalId, support), "");
+    emit VoteCast(signatory, proposalId, support, _castVoteInternal(signatory,
proposalId, support), "");
   }
```

Developer Response

Fixed here

2. Medium - Not repaying a loan during liquidation could lead to bad debt

Not repaying debt when a liquidation occurs could result in bad debt. If this debt is significant, it could lead to problems within the protocol.

Technical Details

`liquidate()` is used to liquidate ended positions that likely are unhealthy. This means that the value of the borrowed `iBGT` at the end of the loan is higher than the collateral NFTs values.

If the position is unhealthy, nobody will likely liquidate it because there are no incentives to do so. After five days, multisig can liquidate such positions without repaying the associated debt. The NFTs are sent to a multisig address, which could then sell them for a lower price, but there is no function to compensate the lenders by increasing `poolSize`.

This scenario leads to the accumulation of bad debt, and given the high value of NFTs typically involved, the total bad debt could be substantial. Subsequently, a decrease in `iBGT` balances within the contract could cause' GiGBT' to lose value.

This is acknowledged in the docs:

> In the unlikely event of a crisis, the DAO may vote to compensate GiBGT holders through additional PORRIDGE emissions, effectively using PORRIDGE to buy the NFT at the price of the outstanding debt.

However, this solution involves an exceptional emission of `PORRIDGE`, which merely shifts the problem elsewhere. Given this, a more robust approach should be considered before resorting to this emergency measure.

Impact

Medium. Bad debt can occur, and `GiGBT` tokens may lose value.

Recommendation

Consider reevaluating the liquidation process. Here are some suggestions:

- Offer incentives for liquidations when the position is unhealthy. This approach helps minimize any bad debt created.
- Use a reserve fund to repay debt during liquidations. This fund can be replenished by selling off the collateralized NFTs afterward.
- Add a function to donate `iBGT` and increase `poolSize`, with access limited to the multisig. This will enable Goldilocks DAO to compensate lenders directly using funds from sold NFTs that were liquidated by multisig.

Developer Response

Fixed here

3. Medium - Lost value in `Goldiswap` accounting

`Goldiswap` uses internal variables `fsl` and `psl` to track the amount of `HONEY` in the contract. When receiving `HONEY`, the contract rounds down both `fsl` and `psl`, leaving some `HONEY` unaccounted for despite being received by the contract.

Technical Details

The function `injectLiquidity()` is used to explain how Goldiswap changes values of `fsl` and `psl`:

```
function injectLiquidity(uint256 liquidity) external {
  uint256 _fsl = fsl;
  uint256 _psl = psl;
  fsl += FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _fsl), (_fsl +
_psl));
  psl += FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _psl), (_fsl +
_psl));
  SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), liquidity);
}
```

The function takes an input parameter, `liquidity`, which represents the amount of `HONEY` transferred to the contract. `fsl` and `psl` are calculated from this liquidity value. These calculations use division that rounds down, leading to value loss. As a result, some `HONEY` will be left unaccounted for and cannot be recovered or swept. This issue is present in all `fsl` and `psl` calculations, not just in the `injectLiquidity()` function.

The following fuzz test can be added to `FuzzGoldiswapTest` to verify the rounding error and demonstrate that not all added liquidity is accounted for:

```
function testFuzzInjectLiquidityRoundingErrors(uint256 liquidity) public {
  vm.assume(liquidity < 1e40);
  uint256 _fsl = goldiswap.fsl();
  uint256 _psl = goldiswap.psl();

  // track psl and fsl before
  uint256 pslAndFslBefore = _fsl + _psl;

  deal(address(honey), address(timelock), liquidity);
  vm.prank(goldiswap.timelock());
  honey.approve(address(goldiswap), liquidity);
  vm.prank(goldiswap.timelock());
  goldiswap.injectLiquidity(liquidity);

  // verify all added liquidity is accounted
  uint256 pslAndFslAfter = goldiswap.fsl() + goldiswap.psl();
  assertEq(pslAndFslBefore + liquidity, pslAndFslAfter, "!pslAndFsl");

  assertEq(honey.balanceOf(address(goldiswap)), liquidity + initialPSL, "!honey");
  assertEq(honey.balanceOf(address(this)), 0);
}
```

Impact

Medium. `Goldiswap` will lose some of the received `HONEY` because of rounding errors.

Recommendation

When calculating `fsl` and `psl` values, compute only one amount (`psl`) and derive the other value (`fsl`) by subtracting from the total amount. This approach will also reduce gas costs.

Here's an example of how to modify the function `injectLiquidity()`. This implementation reduces the gas cost from `44225` to `39301`.

```
function injectLiquidity(uint256 liquidity) external {
  uint256 _fsl = fsl;
  uint256 _psl = psl;
  uint256 additionalPsl = FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity,
_psl), (_fsl + _psl));
  psl = _psl + additionalPsl;
  fsl = _fsl + liquidity - additionalPsl;
  SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), liquidity);
}
```

Apply the same implementation when calculating both `fsl` and `psl` values in the function `sell()`.

Modify the tests to verify that all received `HONEY` is properly accounted for.

Developer Response

Fixed here

4. Medium - `repay()` and `liquidate()` will revert if `ibgtVault` is paused

Crucial functions `repay()` and `liquidate()` depend on calling the Infrared vault, which can be paused.

Technical Details

From the provided docs file, the Infrared vault has the following function description for staking:

```
function stake(uint256 amount) external nonReentrant whenNotPaused
updateReward(msg.sender);
```

The function has the modifier `whenNotPaused`, meaning that the function call will revert when the contract is paused. In the `Goldilend` contract, `ibgtVault` is used in the function `_refreshiBGT()`. This function is used in the `repay()` and `liquidate()` flows.

If Infrared pauses the `ibgtVault` contract, it will disable calling `repay()` and `liquidate()` on the `Goldilend` contract.

Impact

Medium. When relying on other protocols, be aware of possible negative scenarios like pausing.

Recommendation

Before staking into the Infrared vault, verify that the contract is not paused so the crucial functions won't revert. In the case of a paused contract, track the unstaked amount. Use this unstaked amount in the next stake calls. Add a function to enable the staking of the unstaked amount.

Validate for potential call reverts to the Infrared contract after it is public.

Developer Response

Fixed here

5. Medium - Limit the number of NFTs per loan

The user can create a loan with an undefined number of NFTs as collateral. Too many NFTs as collateral will make it undesirable for the loan to be liquidated.

Technical Details

The function `borrow()` accepts an array of NFTs as collateral for opening a loan. There is no limit on the number of NFTs that can be used as collateral. The only limit is 1/10th the size of the pool. A loan with multiple NFTs will be hard to liquidate, making it undesirable to the protocol.

This problem will have a bigger impact when the loan is created with multiple NFTs from the same collection. Liquidating multiple NFTs from the same collection will lower the floor price with each additional sale, making it hard to liquidate at a fair price.

Impact

Medium. Loans with multiple NFTs as collateral will be undesirable to liquidate, leading to bad debt.

Recommendation

Consider removing the option to create borrowing positions with multiple NFTs or at least limit the maximum number of NFTs per loan. If loans from the same collection are needed, scale down the fair value of each additional NFT collateral.

Developer Response

Fixed here

6. Medium - Yield can get locked in Goldivaults

`Goldivault` and `GoldivaultNegative` transfer an array of yield tokens. Some of the tokens could revert when sending zero values.

Technical Details

Yield tokens are added using the function `addYieldTokens()`, but there is no function to remove a yield token once it has been added.

Users can claim yield tokens by calling `redeemYield()`, which will loop through all yield tokens and send part of the yield tokens earned by the user. If one of the tokens doesn't accept zero-value transfers, it will revert, causing the whole redeem transaction to revert. This will prevent all users from redeeming any future yield because the yield tokens cannot be removed.

Additionally, if too many tokens are added to the list, the function `redeemYield()` could cause an out-of-gas exception inside the sending yield tokens loop. Adding yield tokens is limited to a multisig and not a timelock controller. The same problem with a too-long token list `rewardTokens` could occur in `Goldilend`.

Impact

Medium. Redeeming yield could be locked for all users.

Recommendation

Skip sending zero amounts to eliminate possible reverts and save gas.

```
  uint256 claimable = FixedPointMathLib.mulWad(finalYield, yieldShare);
+ if (claimable != 0) {
    SafeTransferLib.safeTransfer(yieldTokens[i], msg.sender, claimable);
+ }
```

Limit the maximum number of tokens that can be added to the lists `yieldTokens` and `rewardTokens`.

Developer Response

Fixed here and here

Low Findings

1. Low - The proposal can be canceled after it has reached the threshold

`Goldigovernor` is forked from Uniswap's `GovernorBravoDelegate` with some modifications. These modifications enable anyone to cancel proposals even when the proposer delegates are at the threshold value.

Technical Details

Uniswap `GovernorBravoDelegate` enables to cancel the proposal only when the proper delegates are below the threshold. The function description specifies the same: "proposer delegates dropped below proposal threshold".

`Goldigovernor` has the same function description but uses a slightly different logic. It uses `revert` instead of `require` and inverts the logic to cancel proposals. However, it does not handle the edge case where the proposer's delegates are equal to the threshold, as described in the function notice.

Impact

Low.

Recommendation

Disable canceling proposals when the proposer's delegates are equal to the threshold:

```
- if(msg.sender != proposal.proposer &&
GovLocks(govlocks).getPriorVotes(proposal.proposer, block.number - 1) >
proposalThreshold) revert InvalidCancel();
+ if(msg.sender != proposal.proposer &&
GovLocks(govlocks).getPriorVotes(proposal.proposer, block.number - 1) >=
proposalThreshold) revert InvalidCancel();
```

Developer Response

Fixed here

2. Low - `Defeated` proposals are validated as `Succeeded`

`Goldigovernor` is forked from Uniswap's `GovernorBravoDelegate` with some modifications. The modifications use an incorrect order of state checks and return the `Defeated` state as `Succeeded`. This means that a `Defeated` proposal can be added to the queue. Defeated proposals are the ones that are downvoted and those that didn't pass the quorum.

Technical Details

Uniswap's `GovernorBravoDelegate` when checking proposal state, first checks if the proposal has been `Defeted`, meaning the more votes, or an equal number, are against than for. Only if this check passes, then verifies the proposal `Succeeded` using the check `eta == 0`. The check `eta == 0` is `true` as long as the proposal has not been queued. This means that all proposals will have `eta == 0` and until they are added to the queue.

`Goldigovernor` uses a naive approach in checking proposal state. It first checks the value `eta == 0` and only after checks if it has enough votes. This enables queueing proposals that are `Defeated`.

Impact

Low. Proposals that didn't pass quorum and didn't have enough votes will be validated as successful and can be added to the queue.

Recommendation

Use the correct order of proposal state checks:

```
function _getProposalState(uint256 proposalId) internal view returns (ProposalState) {
    Proposal storage proposal = proposals[proposalId];
    if (proposal.cancelled) return ProposalState.Canceled;
    else if (block.number <= proposal.startBlock) return ProposalState.Pending;
    else if (block.number <= proposal.endBlock) return ProposalState.Active;
+   else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes <
quorumVotes) {
+     return ProposalState.Defeated;
+   }
    else if (proposal.eta == 0) return ProposalState.Succeeded;
    else if (proposal.executed) return ProposalState.Executed;
-   else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes <
quorumVotes) {
-      return ProposalState.Defeated;
-   }
    else if (block.timestamp >= proposal.eta + Timelock(timelock).GRACE_PERIOD()) {
      return ProposalState.Expired;
    }
    else {
      return ProposalState.Queued;
    }
  }
```

Add the following test to validate that the `Defeated` proposal has the correct state and that a `Defeated` proposal cannot be added to the queue:

```
function testDefeatedProposalFailQueueAndReturnDefeated() public {
    (
      address[] memory targets,
      string[] memory signatures,
      bytes[] memory calldatas,
      uint256[] memory values
    ) = proposyDiff();
    deal(address(goldiswap), address(this), 399e18);
    goldiswap.approve(address(govlocks), 399e18);
    govlocks.deposit(399e18);
    govlocks.delegate(address(this));
    vm.roll(2);
    goldigov.propose(targets, values, signatures, calldatas, "");
    vm.roll(72);
    goldigov.castVote(1, 1);
    vm.roll(18000);
    Goldigovernor.ProposalState state = goldigov.state(1);
    assertEq(uint256(state), 3);
    vm.expectRevert(abi.encodeWithSelector(Goldigovernor.InvalidProposalState.selector));
    goldigov.queue(1);
 }
```

Developer Response

Fixed here

3. Low - Anyone can burn `govLOCKS` without losing voting power

Solady ERC20 implementation allows transfers to `address(0)`, allowing anyone to burn `govLOCKS` tokens. However, the associated voting power is not transferred when tokens are burned in this way.

Technical Details

Solady ERC20 implementation specifies:

> /// - The ERC20 standard allows minting and transferring to and from the zero address, /// minting and transferring zero tokens, as well as self-approvals. /// For performance, this implementation WILL NOT revert for such actions. /// Please add any checks with overrides if desired.

When someone performs a transfer `_afterTokenTransfer()` is called. However, `_moveDelegates()` is not called when `to` is `address(0)`.

As a result, the voting power is not transferred, as shown in the following PoC:

```
function testPoC() public {
    deal(address(goldiswap), address(this), govLocksAmt);
    goldiswap.approve(address(govlocks), govLocksAmt);
    govlocks.deposit(govLocksAmt);
    govlocks.delegate(address(this));
    (bool success,) =
        address(govlocks).call(abi.encodeWithSignature("transfer(address,uint256)",
address(0), govLocksAmt));
    require(success, "Transfer failed");
    vm.roll(2);

    assertEq(govlocks.getVotes(address(this)), govLocksAmt);
    assertEq(govlocks.getPriorVotes(address(this), 1), govLocksAmt);
}
```

Impact

Low. Anyone can burn `govLOCKS`, but their associated voting power is not transferred subsequently.

Recommendation

Prevent arbitrary burn of `govLOCKS` tokens:

```
function transferFrom(address from, address to, uint256 amt) public override returns
(bool) {
    if (to == address(0)) revert TransferToZeroAddress();
    return super.transferFrom(from, to, amt);
}

function transfer(address to, uint256 amt) public override returns (bool) {
    if (to == address(0)) revert TransferToZeroAddress();
    return super.transfer(to, amt);
}
```

Developer Response

Fixed here

4. Low - Contracts can be initialized multiple times

Contract initialization should be done only once. However, contracts `Goldivault.sol`, `GoldivaultNegative.sol`, `GoldivaultSwap.sol` and `Goldilend.sol` can be initialized multiple times.

Technical Details

Initializers do not check if the contracts have already been initialized and can be called multiple times:

- `Goldivault.initializeProtocol()`
- `GoldivaultNegative.initializeProtocol()`
- `Goldiswap.initializeProtocol()`
- `Goldilend.initializeParameters()`
- `Goldilend.initializeBeras()`
- `Goldilend.initializePartners()`

This could lead to unexpected behavior since the functions modify relevant state variables.

Impact

Low. Allowing multiple initializations could lead to unexpected behavior. However, this risk is mitigated because the function could only be called by `multisig`.

Recommendation

Prevent multiple calls to initializer functions.

Developer Response

Fixed here and here

5. Low - `Goldiswap` invariant can be broken

`Goldiswap` has two variables to track the amount of `HONEY` tokens in the contract. The contract can be initialized without enough `HONEY`, breaking the contract's invariant.

Technical Details

`Goldiswap` variables `fsl` and `psl` to the track amount of `HONEY` and define LOCK floor and market prices. Setting the initial values of these variables is done in the constructor without guaranteeing that there will be enough `HONEY` to cover these values.

The function `initializeProtocol()` is used to initialize the contract. One parameter is the amount of `HONEY` to transfer to the contract. If the amount of sent `HONEY` is below the sum of variables `fsl` and `psl` the contract will be in an invalid state. `Goldilocked` can borrow `HONEY` from `Goldiswap` leading to less `HONEY` in the contract.

This invariant must always be true: `fsl + psl - borrowed <= balance of HONEY`. If there is less `HONEY`, the last users won't be able to redeem `LOCKS` for `HONEY`.

Impact

Low. Initializing without enough funds will lead to bad accounting, leaving the last users without funds.

Recommendation

Option A

In the case of borrowing the maximum amount of `HONEY` for the team and seed investors in `Goldilocker` constructor, validate that the initial `LOCKS` supply won't exceed borrowed `HONEY` at the floor price.

```
function initializeProtocol(uint256 amount) external {
  if(msg.sender != multisig) revert NotMultisig();
  tradingActive = true;
  SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), amount);
  uint256 initialSupply = totalSupply();
  uint256 borrowedHoney = FixedPointMathLib.mulWad(initialSupply, _floorPrice(fsl,
initialSupply));
  if (ERC20(honey).balanceOf(address(this)) < (fsl + psl - borrowedHoney)) revert
MissingHoney();
}
```

Extend `InvariantGoldiswapTest` with invariant `fsl + psl - initiallyBororwedAmount <= balance of HONEY`:

```
function invariant_enoughHoney() public {
  uint256 honeyBalnace = honey.balanceOf(address(goldiswap));
  uint256 honeyAccounted = goldiswap.fsl() + goldiswap.psl() - initialBorrowing;
  assertGe(honeyBalnace, honeyAccounted, "Not enough honey");
}

function invariant_enougHoneyToCoverFloorPrice() public {
  uint256 totalSupply = goldiswap.totalSupply();
  uint256 floorPrice = goldiswap.floorPrice();
  assertGe(honey.balanceOf(address(goldiswap)) + initialBorrowing, totalSupply *
floorPrice / 1e18, "Not enough honey");
}
```

Option B

Inside function `initializeProtocol()` verify that there is enough `HONEY` in the contract. Additionally, disable borrowing before initializing for easier accounting.

```
function initializeProtocol(uint256 amount) external {
  if(msg.sender != multisig) revert NotMultisig();
  tradingActive = true;
  SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), amount);
  if (ERC20(honey).balanceOf(address(this)) < (fsl + psl)) revert MissingHoney();
}
```

Extend `InvariantGoldiswapTest` with invariant `fsl + psl <= balance of HONEY`:

```
function invariant_enoughHoney() public {
  uint256 honeyBalnace = honey.balanceOf(address(goldiswap));
  uint256 honeyAccounted = goldiswap.fsl() + goldiswap.psl();
  assertGe(honeyBalnace, honeyAccounted, "Not enough honey");
}
```

Developer Response

Fixed here

6. Low - `Goldilocked` invariants can be broken

`Goldilocked` initially distributes an initial amount of stacked `LOCKS` tokens. The distributed amount can differ from `Goldilocked` `LOCKS` balance, breaking contract invariants.

Technical Details

`Goldiswap` constructor mints an initial supply amount of `LOCKS` to `Goldilocked`, which then distributes these tokens among two team members and seed investors as stacked `LOCKS`.

However, no checks are in place to ensure that the distributed amounts are the same as the originally minted `LOCKS` amount.

If errors occur during the distribution to team members and seed investors, the initial `Goldilocked` balance of the `LOCKS` token and the sum of `stackedLocks` balances can differ.

Impact

Low. This situation could result in `LOCKS` tokens becoming stuck in the contract if the distributed amount is less than the `Goldilocked` `LOCKS` balance. If the distributed amount exceeds the `Goldilocked` `LOCKS` balance, it could break another key invariant: “The sum of `stackedLocks` must always be less than or equal to the `LOCKS` total supply.” This would prevent the last user from unstaking their `LOCKS` tokens.

Recommendation

Verify that the distributed amount in the `Goldilocked` constructor matches the amount minted in the `Goldiswap` constructor.

Developer Response

Fixed here and here

7. Low - `Goldilend` invariant can be broken

`Goldilend` has two variables to track the amount of `iBGT` token in the contract. The contract can be initialized without enough HONEY, breaking the contract invariant at the start.

Technical Details

`Goldilend` variables `poolSize` and `outstandingDebt` to track the amount of `iBGT` and define `GiBGT` mint ratio. Idle `iBGT` is deposited into `ibgtVault`, and its balance must be considered.

The function `initializeParameters()` is used to initialize start values. The `_startingPoolSize` param is used to set `poolSize` but without verifying that there is enough `iBGT` in the contract.

This invariant must always be true: `poolSize + outstandingDebt == iBGT balance + ibgtVault deposits`.

Impact

Low. Initializing without enough funds will lead to bad accounting.

Recommendation

Remove the option to set start `poolSize` or verify there is enough `iBGT` token in the contract and `ibgtValut` to cover the start value.

Extend `InvariantGoldilendTest` with the invariant `poolSize + outstandingDebt == iBGT balance + ibgtVault deposits`:

```
function invariant_poolSize() public {
  uint256 poolSize = goldilend.poolSize();
  uint256 outstandingDebt = goldilend.outstandingDebt();
  uint256 vaultDeposit = IiBGTVault(goldilend.ibgtVault()).balanceOf(address(goldilend));
  uint256 ibgtBalance = ibgt.balanceOf(address(goldilend));
  assertEq(poolSize, outstandingDebt + vaultDeposit + ibgtBalance);
}
```

Developer Response

Fixed here

8. Low - Borrowers may lose some opportunity value on loans

Borrowers pay interest on their loaned assets for the entire term. However, if they wait until the loan's end date to repay, they risk having their collateral liquidated.

Technical Details

Borrowers can repay their loans by calling `repay()`, but they are unable to do so once the `endDate` has passed.

If they wait until the `endDate` to repay, they could be at risk of liquidation, forcing them to settle their debts before this deadline. A small congestion in the network would leave them without the option of repaying the loan.

Impact

Low. Borrowers may lose some opportunity value on their loaned amount and are still required to pay interest for the entire term.

Recommendation

Introduce a buffer period during which borrowers can repay their debts and are protected from liquidation.

Developer Response

Fixed here

9. Low - `repay()` is susceptible to reentrancy

`safeTransferFrom()` is susceptible to reentrancy. A user can reenter the contract in `repay()` before the `iBGT` transfer is made.

Technical Details

When `safeTransferFrom()` is called, if the callee is a contract, reentrancy is possible because a call to `onERC721Received()` is triggered. The `repay()` function does not follow the Checks-Effects-Interactions (CEI) pattern because the `iBGT` transfer is done after the external `safeTransferFrom()` call.

A user can reenter the `borrow()` function, borrow again against their NFTs, and use the borrowed amount to repay the initial debt.

Impact

Low. Borrowers can reenter `borrow()` to repay their debt.

Recommendation

Follow the CEI pattern in `repay()`:

```
     function repay(uint256 repayAmount, uint256 userLoanId) external {
         (Loan memory userLoan, uint256 index) = _lookupLoan(msg.sender, userLoanId);
         if (userLoan.borrowedAmount < repayAmount) revert ExcessiveRepay();
         if (block.timestamp > userLoan.endDate) revert LoanExpired();
         uint256 interestLoanRatio = FixedPointMathLib.divWad(userLoan.interest,
userLoan.borrowedAmount);
         uint256 interest = FixedPointMathLib.mulWadUp(repayAmount, interestLoanRatio);
         outstandingDebt -= repayAmount - interest > outstandingDebt ? outstandingDebt :
repayAmount - interest;
         loans[msg.sender][index].borrowedAmount -= repayAmount;
         loans[msg.sender][index].interest -= interest;
         poolSize += interest * (1000 - (multisigShare + apdaoShare)) / 1000;
         _updateInterestClaims(interest);
+        SafeTransferLib.safeTransferFrom(ibgt, msg.sender, address(this), repayAmount);
+        _refreshiBGT(repayAmount);
         if (userLoan.borrowedAmount - repayAmount == 0) {
             for (uint256 i; i < userLoan.collateralNFTs.length;) {
                 IERC721(userLoan.collateralNFTs[i]).safeTransferFrom(
                     address(this), msg.sender, userLoan.collateralNFTIds[i]
                 );
                 unchecked {
                     ++i;
                 }
             }
         }
-        SafeTransferLib.safeTransferFrom(ibgt, msg.sender, address(this), repayAmount);
-        _refreshiBGT(repayAmount);
         emit Repay(msg.sender, repayAmount);
     }
```

As suggested in the gas finding, `transferFrom()` can be used instead of `safeTransferFrom()`.

Developer Response

Fixed here

Gas Saving Findings

1. Gas - Remove unused code

Some code is unused and can be deleted to save gas on deployment.

Technical Details

The following events can be deleted:

- Event `NewAdmin` in Goldigovernor
- Event `NewAdmin` in Timelock

Impact

Gas savings.

Recommendation

Remove unused code to save gas.

Developer Response

Fixed here

2. Gas - Optimize for loops

Technical Details

Many for loops can be optimized to save gas.

Impact

Gas savings.

Recommendation

Optimize for loops by applying these changes:

- increment `i` in an `unchecked` block with `++i` statement
- cache the array length outside a loop as it saves reading it on each iteration

- don’t initialize the loop counter variable as there is no need to initialize it to zero because it is the default value

Developer Response

Fixed here

3. Gas - Declare variables immutable when possible

Using immutable variables can provide gas savings compared to non-immutable variables if the variables only need to be set once.

Technical Details

Several variables are set in the constructors of `Goldivault.sol` and `GoldivaultNegative.sol` and are not modified later. To save gas, these variables can be declared immutable.

Impact

Gas savings.

Recommendation

Declare those variables as immutable.

Developer Response

Fixed here

4. Gas - Use local variables instead of storage variables

Using local variables instead of storage variables can save gas.

Technical Details

The function `injectLiquidity()` is accessing storage variable `fsl` and `psl` while it could use local variables `_fsl` and `_psl`.

Impact

Gas savings.

Recommendation

Change the function `injectLiquidity()`. Gas cost on average from provided test are down from `43936` to `43668`.

```
function injectLiquidity(uint256 liquidity) external {
  uint256 _fsl = fsl;
  uint256 _psl = psl;
- fsl += FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _fsl), (_fsl + _psl));
- psl += FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _psl), (_fsl + _psl));
+ fsl = _fsl + FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _fsl), (_fsl + _psl));
+ psl = _psl + FixedPointMathLib.divWad(FixedPointMathLib.mulWad(liquidity, _psl), (_fsl + _psl));
  SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), liquidity);
}
```

Developer Response

Fixed here

5. Gas - Remove `outstandingRewardsPerReward` mapping and use a local array

Storage variable `outstandingRewardsPerReward` can be removed because it is used only locally to save gas.

Technical Details

In `_updateClaimableRewards()`, there are many storage writes and reads that can be avoided by using a local array instead of the `outstandingRewardsPerReward` storage mapping.

Impact

Gas savings.

Recommendation

Remove the `outstandingRewardsPerReward` mapping and use a local array in `_updateClaimableRewards()`:

```
   function _updateClaimableRewards(address user) internal {
     uint256 rewardTokensLength = rewardTokens.length;
+    uint256[] memory outstandingRewardsPerReward = new uint256[](rewardTokensLength);
     for(uint8 i; i < rewardTokensLength; ++i) {
-      outstandingRewardsPerReward[rewardTokens[i]] =
ERC20(rewardTokens[i]).balanceOf(address(this));
+      outstandingRewardsPerReward[i] = ERC20(rewardTokens[i]).balanceOf(address(this));
     }
     iBGTVault(ibgtVault).getReward();
     for(uint8 i; i < rewardTokensLength; ++i) {
       address rewardToken = rewardTokens[i];
-      uint256 outstandingRewards = ERC20(rewardToken).balanceOf(address(this)) -
outstandingRewardsPerReward[rewardToken];
+      uint256 outstandingRewards = ERC20(rewardToken).balanceOf(address(this)) -
outstandingRewardsPerReward[i];
       claimableRewardsPerGiBGTStored[rewardToken] =
_claimableRewardPerGiBGT(rewardToken, outstandingRewards);
       lastRewardUpdateTime[rewardToken] = block.timestamp;
       if(user != address(0)) {
         claimableRewards[user][rewardToken] = _calculateClaimableRewards(user,
rewardToken, outstandingRewards);
         rewardPerTokenDebt[user][rewardToken] =
claimableRewardsPerGiBGTStored[rewardToken];
       }
```

```
    }
  }
```

This will lower the average gas cast of function `claim()` from 207566 to 203731. Removing the `_calculateClaimableRewards()` function can also provide further gas savings.

Developer Response

Fixed here

6. Gas - Delete mapping items instead of setting to default values

Delete mapping items instead of setting them to default values to save gas.

Technical Details

The function `withdrawBoost()` sets mapping item to new item with defaults value. Using delete will save gas.

Impact

Gas savings.

Recommendation

The average gas cost in `UnitGoldilendTest` is lowered from `86828` to `86729`.

```
- Boost memory newUserBoost = Boost({
-   partnerNFTs: nfts,
-   partnerNFTIds: ids,
-   expiry: 0,
-   boostMagnitude: 0
- });
- boosts[msg.sender] = newUserBoost;
+ delete boosts[msg.sender];
```

Developer Response

Fixed here

7. Gas - Calculate the same values only once

Calculate the same values only once and store them as a local variable to save gas.

Technical Details

In the function `_calculateInterest()`, the value `FixedPointMathLib.divWad(duration, 365 days)` is calculated twice: first at L582 and second at L583.

Impact

Gas savings.

Recommendation

Calculate the same value only once and store it as the local variable.

Developer Response

Fixed here

8. Gas - Use `uint256` instead of `uint8` as for loop counters

Type `uint256` is cheaper than type `uint8` for loop counters.

Technical Details

Here are some for loops that can be changed in the `Goldilend` contract: L636, L643, L673, and others.

For more information on why it is cheaper to use `uint256`, check this link.

Impact

Gas savings.

Recommendation

Change the loop counter to the type `uint256` in all loops to save gas.

Developer Response

Fixed here

9. Gas - Remove the `concluded` variable from vault contracts

The bool variable `concluded` can be removed to save gas.

Technical Details

Contracts `Goldivault` and `GoldivaultNegative` use the bool variable `concluded` to track if the vault has been concluded. These contracts also store timestamps `concludeTime` to track when the contracts were concluded. Because the time and bool flag are updated at the same time, the bool flag can be safely removed.

Checking `concludeTime != 0` is equivalent to `concluded = true`.

Impact

Gas savings.

Recommendation

Remove the storage variable `concluded`.

Developer Response

Fixed here

10. Gas - Use constant variables

Variables set to a known value before deployment and not changed after deployment can be constant variables to save gas.

Technical Details

The variables `MAX_FLOOR_REDUCE` and `MAX_RATIO`

Impact

Gas savings.

Recommendation

Change variables to constants when possible.

Developer Response

Fixed here

11. Gas - Limit the number of incremental loops

`LOCKS` tokens buy and sell operations are performed in incremental loops. If there are too many loops, the function will revert with an out-of-gas exception.

Technical Details

Buy and sell loop increments are 0.001% of the total supply. If a user tries to buy a lot of tokens during a small supply, the buy transaction will revert with an out-of-gas exception and consume the entire gas limit.

Impact

Gas savings.

Recommendation

Define the maximum number of loops and revert immediately if the required number of loops is higher. This will save gas for the end user.

Developer Response

Acknowledged, the supply will never be small due to team tokens being permanently staked.

12. Gas - Use `transferFrom()` instead of `safeTransferFrom()`

For ERC721 tokens, where it is safe to save gas, use `transferFrom()` instead of `safeTransferFrom()`.

Technical Details

In the `Goldilend` contract, it is safe to use `IERC721.transferFrom()` instead of `IERC721.safeTransferFrom()`. When the transfer is done from `msg.sender` to the contract, it is safe to use `transferFrom()` because the contract has implemented `onERC721Received()`. In the other direction, it is also safe to use `transferFrom()` because the token returns to the `msg.sender`, who can receive ERC721 tokens, as the sender initially sent it to the contract.

The only place where `safeTransferFrom()` is needed is in `liquidate()` because the ERC721 token is sent to the liquidator, which can be a contract without a receiver interface.

Using `safeTransferFrom()` enables reentrancy because `onERC721Received()` is triggered on the user side when he receives the token. This opens a scenario where the user can repay the current loan with new borrowing. Using `transferFrom()` will eliminate this option.

Impact

Gas savings.

Recommendation

In the' Goldilend' contract, use `transferFrom()` instead of `safeTransferFrom()` to transfer ERC721 tokens, except in the `liquidate()` function when transferring collateral tokens to the liquidator.

Developer Response

Fixed here

Informational Findings

1. Informational - Avoid possible arithmetic overflows or underflows

Technical Details

In Goldivault.sol and GoldivaultNegative.sol `deposit()` there is a calculation that can underflow and throw a `Panic` error.

Impact

Informational.

Recommendation

Avoid the possible underflow:

```
-    uint256 remainingTime = endTime - block.timestamp;
+    uint256 remainingTime = block.timestamp > endTime ? 0 : endTime - block.timestamp;
```

Developer Response

Fixed here

2. Informational - Avoid duplicated code

Technical Details

`Goldivault.sol` and `GoldivaultNegative.sol` share the same code except for `deposit()` and `redeemOwnership()`. It is best practice to avoid duplicated code to improve maintainability.

Using modifiers instead of `if` checks can remove duplicated code. Some examples are:

- checks for timelock
- checks for multisig

Impact

Informational.

Recommendation

Use a base contract for the shared code and implement `deposit()` and `redeemOwnership()` in `Goldivault.sol` and `GoldivaultNegative.sol`.

Use modifiers to reduce code and possible errors.

Developer Response

Acknowledged, there is additional changes (differences in redeemYield() and a parameter being removed) between Goldivault and GoldivaultNegative that would be easier if they were separated. Modifiers were removed in Goldilocked to reduce amount of deployed bytecode.

3. Informational - Follow the order of functions best practice

Technical Details

Functions within all the contracts are not ordered according to the Solidity style guide.

e.g., `view` functions are placed before non-view functions in all the contracts within the protocol, while the Solidity style guide recommends:

> Within a grouping, place the view and pure functions last.

Impact

Informational.

Recommendation

Order functions as per Solidity style guide to improve readability.

Developer Response

Fixed here

4. Informational - Missing event emits

Some functions that transfer values or modify state variables do not have events. Events can assist with analyzing contracts' on-chain history and are beneficial for adding to important functions.

Technical Details

All permissioned functions are missing events in the following contracts:

- `Goldilend`
- `Goldivault`
- `GoldivaultNegative`
- `Goldilocked`

The event should be emitted after updating the storage variable `targetRatio` in `Goldiswap`.

Impact

Informational.

Recommendation

Add events to the functions listed above.

Developer Response

Fixed here

5. Informational - Replace magic numbers with constants

Constant variables should be used instead of magic numbers to prevent typos and better explain such values.

Technical Details

Some of the numbers could be defined as private constants to explain their value:

- 3 could be replace with `BUY_TAX` and `1000` with `PRECISION`.
- 100_000e18 is loop step size.
- 100 is a full ratio and 50 is a half ratio.
- 1000 is maximum discount.
- 5e17 can be `INTEREST_PAYMENT_PERCENTAGE`, 5% value.

Impact

Informational.

Recommendation

Use constant variables instead of magic numbers. This will not change gas consumption.

Developer Response

Fixed here

6. Informational - Align documentation with implementation

Technical Details

Docs states:

> Borrowers will be able to borrow up to the full valuation of their NFT's

However, there is a check inside the `borrow()` function that prevents users to borrow more than 10% of the entire pool size.

Impact

Informational.

Recommendation

Add this information to the docs.

Developer Response

Fixed here

7. Informational - Incorrect NatSpec

Some comments in NatSpec are incorrect.

Technical Details

NatSpec specifies that the function returns four values, but the implementation returns only three, here and here.

For `borrow()`, NatSpec specifies the `amount` parameter is the amount of `HONEY` to borrow, but this is not correct because the fee is subtracted from the specified amount. This will leave users with a lower amount of `HONEY` than requested.

Impact

Informational.

Recommendation

Update comments in NatSpec to match the implementation code. Alternatively, change the `borrow()` function to borrow the specified amount of `HONEY`.

Developer Response

Fixed here

8. Informational - Cap repay value to the maximum possible rather than reverting

Capping the repay value to the maximum possible will improve user experience. A similar flow is used in other lending protocols where you can repay using `type(uint256).max`.

Technical Details

The function `repay()` is used to repay a user loan. The function will revert if the user tries to repay the full debt and sends a higher amount. Capping the user input value to their debt achieves a better user experience.

The contract `Goldilend` also has a `repay()` function. Capping `repayAmount` to `borrowedAmount` would improve user experience.

Impact

Informational.

Recommendation

```
function repay(uint256 amount) external {
-  if(borrowedHoney[msg.sender] < amount) revert ExcessiveRepay();
+  uint256 borrowed = borrowedHoney[msg.sender];
+  if (amount > borrowed) amount = borrowed;
```

```
function repay(uint256 repayAmount, uint256 userLoanId) external {
    (Loan memory userLoan, uint256 index) = _lookupLoan(msg.sender, userLoanId);
-   if(userLoan.borrowedAmount < repayAmount) revert ExcessiveRepay();
+   if (repayAmount > userLoan.borrowedAmount) repayAmount = userLoan.borrowedAmount;
```

Developer Response

Fixed here

9. Informational - Improve naming choices

Follow the standard naming choices.

Technical Details

Custom tokens `OwnershipToken` and `YieldToken` use unconventional naming (`to`) for the address parameter in the burn function. Burning tokens is done `from` some address, not `to` some address.

Impact

Informational.

Recommendation

Rename the address parameter `to` to `from` to follow standard implementation naming as in the Solady library.

Developer Response

Fixed here

10. Informational - Consider refactoring the redemption process for `YT` tokens

Technical Details

`YT` tokens enable their holders to claim yields generated by the initial `depositToken`. Yields can be claimed by calling `redeemYield()` only after expiration plus a delay. This means that only the owner of `YT` at that time can claim all yields accrued from the initial `depositToken`. Consequently, if a user sells their `YT`, the yields generated up to that point will go to the buyer.

Although this is a design choice, there may be scenarios (e.g., points for airdrops) where this approach may not be the best choice.

Impact

Informational.

Recommendation

Consider refactoring `redeemYield()` to allow child contracts flexibility to choose between two options: claiming yields only at the end or allowing users to claim their accrued yields at any time.

Developer Response

Acknowledged, this redemption process is a design choice.

11. Informational - Provide the ability to calculate slippage amounts directly from the contract

Technical Details

Mechanisms for slippage protection are implemented in `buy()` and `sell()`, where you can set a `maxAmount` for buys and a `minAmount` for sells. However, the effective price is not determined until `_buyLoop()` and `_sellLoop()` are executed.

This means that third parties wanting to interact with these functions must calculate the correct amounts themselves or delegate the decision to the user.

Impact

Informational.

Recommendation

Make `_buyLoop()` and `_sellLoop()` `public`. Since they are `pure` functions, interacting with them will not cost gas and will provide external parties with a secure and accurate outcome.

Developer Response

Fixed here

12. Informational - Incorrect dependency setup

Some dependencies are missing in the project setup.

Technical Details

The project dependencies are defined in the file `.gitmodules`, but it contains only the forge dependency. Solady, openzeppelin-contracts, and huff-foundry are missing.

Impact

Informational. This is not in the review scope.

Recommendation

Add all needed dependencies to the project setup.

Developer Response

Fixed here

13. Informational - Goldivaults don't support fee-on-transfer tokens

`Goldivault` and `GoldivaultNegative` don't support fee-on-transfer tokens as deposit tokens.

Technical Details

In `Goldivault`, after a deposit, the deposit token should be deposited into the `depositVault`. If the deposit token has a fee-on-transfer, this deposit will revert. Additionally, if the deposit to the deposit vault is skipped in the internal implementation, the accounting will be incorrect.

Impact

Informational.

Recommendation

Make additional changes if you plan to support fee-on-transfer tokens.

Developer Response

Acknowledged, we do not plan to support fee-on-transfer tokens.

Final remarks

Goldilocks carefully selected the DeFi services they want to provide. The governance part is forked from a well-known Uniswap Governance contract, enabling token holders to vote on future protocol improvements via the timelock controller. The custom AMM allows users to trade LOCKS tokens while maintaining a floor price that can only increase. This feature allows Berachain's native stablecoin, HONEY, to be borrowed without liquidation risk. The Berachain community is strong on NFTs; an additional component is built around that. NFT holders can use their tokens as collateral to borrow iBGT. Using NFTs as collateral without accruing bad

debt is difficult to tackle. Multiple high-risk problems were found in the current design, which needs additional changes to lower the risk of bad debt. As NFT values can change, defining a process for these actions is recommended in case of significant value changes.

The codebase is very well organized and has multiple types of testing. The invariant tests could be improved by defining more contract invariants, adding more function selectors, and limiting the number of reverts. The contracts Goldiswap, Goldilocked, and Goldilend would benefit from better invariant tests. It has a good setup with the handlers but needs more work to be effective. Other protocols used by Goldilocks, such as Infrared, are not deployed on the testnet. It is important to verify and test integrations before deploying. Additional integration tests using the latest fork are recommended.

The protocol is built only for Berachain, a high-performance EVM-identical Layer 1 blockchain. Considering that the Berachain mainnet is not out yet, an additional review should be done before deploying on the mainnet.