Goldilocks Audit Report

Prepared by Cyfrin

Version 1.1

Lead Auditor

Hans

April 14, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Goldilocks is a DAO focused on creating specialized DeFi infrastructure for Berachain.

4.1 Key Components

- **Goldiswap:** A custom-designed AMM that users can buy/sell `LOCKS` with.
- **Goldilend:** An NFT lending platform designed exclusively for Bong Bear NFTs and their rebases.
- **Goldivault:** Vaults for yield splitting enabling users to trade and speculate on the future earnings generated by yield-bearing positions across Berachain's native and blue-chip DeFi protocols.

4.2 Design Features

- **Goldiswap** includes two primary liquidity pools: the Floor Supporting Liquidity Pool (FSL) and the Price Supporting Liquidity Pool (PSL). These pools will exclusively consist of HONEY, Berachain's native stablecoin, which is fully collateralized.
- **Goldilend** will enable holders of Bong Bear series NFTs (including Bong, Bond, Boo, Baby, Band, and Bit) to obtain loans against their NFTs without subjecting them to the risk of price-based liquidations.
- **Goldilocks** is partnering with Berachain ecosystem projects to allow their NFTs to have multiple utilities within Goldilend. Users can lock partner NFTs to decrease the interest rate paid on that loan or boost PORRIDGE yield for GiBGT stakers.

4.3 Architecture

Goldilocks presents a distinctive DeFi strategy, incorporating elements such as automated market making (AMM), liquidity pools, and NFT lending.

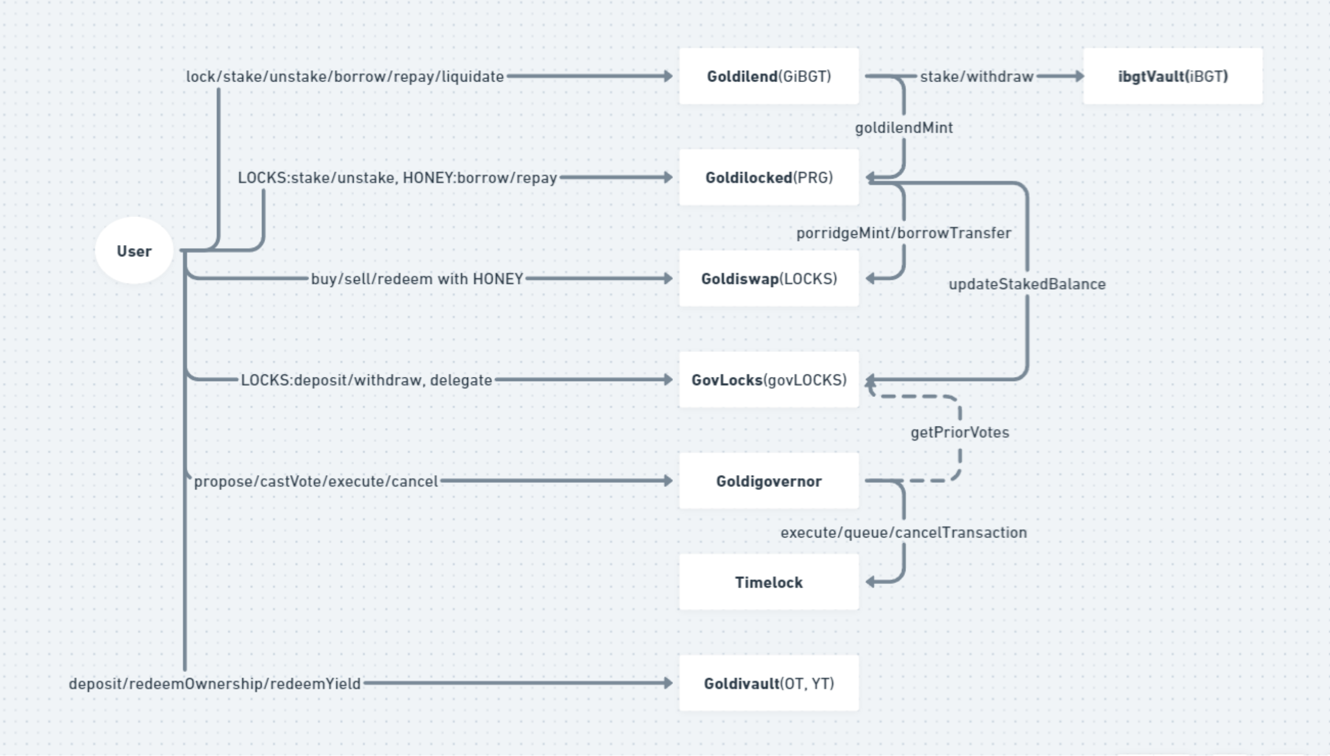
lock/stake/unstake/borrow/repay/liquidate
Goldilend(GiBGT)
stake/withdraw
ibgtVault(iBGT)
goldilendMint
LOCKS:stake/unstake, HONEY:borrow/repay
Goldilocked(PRG)
User
porridgeMint/borrowTransfer
buy/sell/redeem with HONEY
Goldiswap(LOCKS)
updateStakedBalance
LOCKS:deposit/withdraw, delegate
GovLocks(govLOCKS)
getPriorVotes
propose/castVote/execute/cancel
Goldigovernor
execute/queue/cancelTransaction
Timelock
deposit/redeemOwnership/redeemYield
Goldivault(OT, YT)

Figure 1: Protocol Summary

4.4 Tokens to interact with

- **LOCKS:** Governance token for Goldilocks DAO
- **govLOCKS:** Governance wrapper for LOCKS
- **PRG:** Porridge token
- **GiBGT:** Ownership token of Goldilend
- **HONEY:** External token, stablecoin on Berachain
- **iBGT:** External token, Infrared governance token

5 Audit Scope

The following Solidity files were included in the scope of the audit:

```
Goldilend.sol
Goldiswap,sol
Goldivault.sol
OwnershipToken.sol
YieldToken.sol
Goldigovernor.sol
Timelock.sol
Govlocks.sol
Goldilocked.sol
```

6 Executive Summary

Over the course of 16 days, the Cyfrin team conducted an audit on the Goldilocks smart contracts provided by Goldilocks. In this period, a total of 26 issues were found.

Summary

Project Name	Goldilocks
Repository	goldilocks-core
Commit	0822d92fe18a...
Audit Timeline	March 18th - April 8th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	6
Medium Risk	6
Low Risk	7
Informational	2
Gas Optimizations	5
Total Issues	26

Summary of Findings

[H-1] `Goldilend.lock()` will always revert	Resolved
[H-2] Wrong `PoolSize` increment in `Goldilend.repay()`	Resolved
[H-3] Users can extend an expired boost using invalidated NFTs	Resolved
[H-4] Team members can't unstake the initial allocation forever.	Resolved
[H-5] In `GovLocks`, it shouldn't use a `deposits` mapping	Resolved
[H-6] Some functions of `Goldilend` will revert forever.	Resolved
[M-1] `Goldigovernor._getProposalState()` shouldn't use `totalSupply`	Resolved
[M-2] In `Goldivault.redeemYield()`, users can redeem more yield tokens using reentrancy	Resolved
[M-3] Wrong validation in `Goldigovernor.cancel()`	Resolved
[M-4] Users wouldn't cancel their proposals due to the increased `proposalThreshold`	Acknowledged
[M-5] `Goldilend.liquidate()` might revert due to underflow	Resolved

[M-6] In `Goldigovernor`, wrong assumption of block time	Resolved
[L-1] `Goldivault.changeProtocolParameters()` shouldn't update `endTime`	Resolved
[L-2] `Goldilocked._lockedLocks()` should round up	Resolved
[L-3] Possible failure to redeem	Acknowledged
[L-4] Inconsistent comparison while checking `eta`	Resolved
[L-5] Wrong comment	Resolved
[L-6] Missing bracket in docs	Acknowledged
[L-7] Rounding loss of 1 wei	Resolved

7 Findings

7.1 High Risk

7.1.1 `Goldilend.lock()` will always revert

Severity: High

Description: In `lock()`, it calls `_refreshiBGT()` before pulling `iBGT` from the user and will revert while calling `iBGTVault(ibgtVault).stake()`.

```
  function lock(uint256 amount) external {
    uint256 mintAmount = _GiBGTMintAmount(amount);
    poolSize += amount;
    _refreshiBGT(amount); //@audit should call after depositing funds
    SafeTransferLib.safeTransferFrom(ibgt, msg.sender, address(this), amount);
    _mint(msg.sender, mintAmount);
    emit iBGTLock(msg.sender, amount);
  }
...
  function _refreshiBGT(uint256 ibgtAmount) internal {
    ERC20(ibgt).approve(ibgtVault, ibgtAmount);
    iBGTVault(ibgtVault).stake(ibgtAmount); //@audit will revert here
  }
```

Impact: Users can't lock `iBGT` as `lock()` always reverts.

Recommended Mitigation: `_refreshiBGT()` should be called after pulling funds from the user.

Client: Fixed in PR #1

Cyfrin: Verified.

7.1.2 Wrong `PoolSize` increment in `Goldilend.repay()`

Severity: High

Description: When a user repays his loan using `repay()`, it increases `poolSize` with the repaid interest. During the increment, it uses the wrong amount.

```
  function repay(uint256 repayAmount, uint256 _userLoanId) external {
    Loan memory userLoan = loans[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][_userLoanId].borrowedAmount -= repayAmount;
    loans[msg.sender][_userLoanId].interest -= interest;
    poolSize += userLoan.interest * (1000 - (multisigShare + apdaoShare)) / 1000; //@audit should use
    ↪  interest instead of userLoan.interest
...
  }
```

It should use `interest` instead of `userLoan.interest` because the user repaid `interest` only.

Impact: `poolSize` would be tracked wrongly after calling `repay()` and several functions wouldn't work as expected.

Recommended Mitigation: `poolSize` should be updated using `interest`.

Client: Fixed in PR #2

Cyfrin: Verified.

7.1.3 Users can extend an expired boost using invalidated NFTs.

Severity: High

Description: In Goldilend.sol#L251, a user can extend a boost with invalidated NFTs.

- The user has created a boost with a valid NFT.
- After that, the NFT was invalidated using adjustBoosts().
- After the original boost is expired, the user can just call boost() with empty arrays, and the boost will be extended again with the original magnitude.

```
  function _buildBoost(
    address[] calldata partnerNFTs,
    uint256[] calldata partnerNFTIds
  ) internal returns (Boost memory newUserBoost) {
    uint256 magnitude;
    Boost storage userBoost = boosts[msg.sender];
    if(userBoost.expiry == 0) {
...
    }
    else {
      address[] storage nfts = userBoost.partnerNFTs;
      uint256[] storage ids = userBoost.partnerNFTIds;
      magnitude = userBoost.boostMagnitude; //@audit use old magnitude without checking
      for (uint256 i = 0; i < partnerNFTs.length; i++) {
        magnitude += partnerNFTBoosts[partnerNFTs[i]];
        nfts.push(partnerNFTs[i]);
        ids.push(partnerNFTIds[i]);
      }
      newUserBoost = Boost({
        partnerNFTs: nfts,
        partnerNFTIds: ids,
        expiry: block.timestamp + boostLockDuration,
        boostMagnitude: magnitude
      });
    }
  }
```

Impact: Malicious users can use invalidated NFTs to extend their boosts forever.

Recommended Mitigation: Whenever users extend their boosts, their NFTs should be evaluated again.

Client: Fixed in PR #3

Cyfrin: Verified.

7.1.4 Team members can't unstake the initial allocation forever.

Severity: High

Description: When users call unstake(), it calculates the vested amount using _vestingCheck().

```
  function _vestingCheck(address user, uint256 amount) internal view returns (uint256) {
    if(teamAllocations[user] > 0) return 0; //@audit return 0 for team members
    uint256 initialAllocation = seedAllocations[user];
    if(initialAllocation > 0) {
      if(block.timestamp < vestingStart) return 0;
      uint256 vestPortion = FixedPointMathLib.divWad(block.timestamp - vestingStart, vestingEnd -
      ↪  vestingStart);
      return FixedPointMathLib.mulWad(vestPortion, initialAllocation) - (initialAllocation -
      ↪  stakedLocks[user]);
    }
    else {
      return amount;
    }
  }
```

But it returns 0 for team members and they can't unstake forever. Furthermore, in `stake()`, it just prevents seed investors, not team members. So if team members have staked additionally, they can't unstake also.

Impact: Team members can't unstake forever.

Recommended Mitigation: `_vestingCheck` should use the same logic as initial investors for team mates.

Client: Acknowledged, it is intended that the team cannot unstake their tokens. PR #4 fixes issue of `stake` not preventing team members from staking.

Cyfrin: Verified.

7.1.5 In `GovLocks`, it shouldn't use a `deposits` mapping

Severity: High

Description: In `GovLocks`, it tracks every user's deposit amount using a `deposits` mapping. As users can transfer `govLocks` freely, they might have fewer `deposits` than their `govLocks` balance and wouldn't be able to withdraw when they want.

```
  function deposit(uint256 amount) external {
    deposits[msg.sender] += amount; //@audit no need
    _moveDelegates(address(0), delegates[msg.sender], amount);
    SafeTransferLib.safeTransferFrom(locks, msg.sender, address(this), amount);
    _mint(msg.sender, amount);
  }

  /// @notice Withdraws Locks to burn Govlocks
  /// @param amount Amount of Locks to withdraw
  function withdraw(uint256 amount) external {
    deposits[msg.sender] -= amount; //@audit no need
    _moveDelegates(delegates[msg.sender], address(0), amount);
    _burn(msg.sender, amount);
    SafeTransferLib.safeTransfer(locks, msg.sender, amount);
  }
```

Here is a possible scenario.

- Alice has deposited 100 `LOCKS` and got 100 `govLOCKS`. Also `deposits[Alice] = 100`.
- Bob bought 50 `govLOCKS` from Alice to get voting power.
- When Bob tries to call `withdraw()`, it will revert because `deposits[Bob] = 0` although he has 50 `govLOCKS`.

Impact: Users wouldn't be able to withdraw `LOCKS` with `govLOCKS`.

Recommended Mitigation: We don't need to use the `deposits` mapping at all and we can just rely on `govLocks` balances.

Client: Fixed in PR #8

Cyfrin: Verified.

7.1.6 Some functions of `Goldilend` will revert forever.

Severity: High

Description: `Goldilend.multisigInterestClaim()/apdaoInterestClaim()/sunsetProtocol()` will revert forever because they doesn't withdraw `ibgt` from `ibgtVault` before the transfer.

```
  function multisigInterestClaim() external {
    if(msg.sender != multisig) revert NotMultisig();
    uint256 interestClaim = multisigClaims;
    multisigClaims = 0;
    SafeTransferLib.safeTransfer(ibgt, multisig, interestClaim);
  }

  /// @inheritdoc IGoldilend
  function apdaoInterestClaim() external {
    if(msg.sender != apdao) revert NotAPDAO();
    uint256 interestClaim = apdaoClaims;
    apdaoClaims = 0;
    SafeTransferLib.safeTransfer(ibgt, apdao, interestClaim);
  }

...

  function sunsetProtocol() external {
    if(msg.sender != timelock) revert NotTimelock();
    SafeTransferLib.safeTransfer(ibgt, multisig, poolSize - outstandingDebt);
  }
```

As `ibgtVault` has all `ibgt` of `Goldilend`, they should withdraw from `ibgtVault` first.

Impact: `Goldilend.multisigInterestClaim()/apdaoInterestClaim()/sunsetProtocol()` will revert forever.

Recommended Mitigation: 3 functions should be changed like the below.

Client: Fixed in PR #9 and PR #12

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 `Goldigovernor._getProposalState()` shouldn't use `totalSupply`

Severity: Medium

Description: In `_getProposalState()`, it uses `Goldiswap(goldiswap).totalSupply()` during the comparison.

```
  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.eta == 0) return ProposalState.Succeeded;
    else if (proposal.executed) return ProposalState.Executed;
    else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes <
    ↪  Goldiswap(goldiswap).totalSupply() / 20) { //@audit shouldn't use totalSupply
      return ProposalState.Defeated;
    }
    else if (block.timestamp >= proposal.eta + Timelock(timelock).GRACE_PERIOD()) {
      return ProposalState.Expired;
    }
    else {
      return ProposalState.Queued;
    }
  }
```

As totalSupply is increasing in real time, a Queued proposal might be changed to Defeated one unexpectedly due to the increased supply.

Impact: A proposal state might be changed unexpectedly.

Recommended Mitigation: We should introduce another mechanism for the quorum check rather than using totalSupply.

Client: Fixed in PR #5

Cyfrin: Verified.

7.2.2 In Goldivault.redeemYield(), users can redeem more yield tokens using reentrancy

Severity: Medium

Description: Possible reentrancy in Goldivault.redeemYield() if yieldToken has a beforeTokenTransfer hook.

- Let's assume yt.totalSupply = 100, yieldToken.balance = 100 and the user has 20 yt.
- The user calls redeemYield() with 10 yt.
- Then yt.totalSupply will be changed to 90 and it will transfer 100 * 10 / 100 = 10 yieldToken to the user.
- Inside the beforeTokenTransfer hook, the user calls redeemYield() again with 10 yt.
- As yieldToken.balance is still 100, he will receive 100 * 10 / 90 = 11 yieldToken.

```
  function redeemYield(uint256 amount) external {
    if(amount == 0) revert InvalidRedemption();
    if(block.timestamp < concludeTime + delay || !concluded) revert NotConcluded();
    uint256 yieldShare = FixedPointMathLib.divWad(amount, ERC20(yt).totalSupply());
    YieldToken(yt).burnYT(msg.sender, amount);
    uint256 yieldTokensLength = yieldTokens.length;
    for(uint8 i; i < yieldTokensLength; ++i) {
      uint256 finalYield;
      if(yieldTokens[i] == depositToken) {
        finalYield = ERC20(yieldTokens[i]).balanceOf(address(this)) - depositTokenAmount;
      }
      else {
        finalYield = ERC20(yieldTokens[i]).balanceOf(address(this));
      }
      uint256 claimable = FixedPointMathLib.mulWad(finalYield, yieldShare);
      SafeTransferLib.safeTransfer(yieldTokens[i], msg.sender, claimable);
    }
    emit YieldTokenRedemption(msg.sender, amount);
  }
```

Impact: Malicious users can steal yieldToken using redeemYield().

Recommended Mitigation: We should add a nonReentrant modifier to redeemYield().

Client: Fixed in PR #13

Cyfrin: Verified.

7.2.3 Wrong validation in `Goldigovernor.cancel()`

Severity: Medium

Description: In `Goldigovernor.cancel()`, the proposer should have fewer votes than `proposalThreshold` to cancel his proposal.

```
function cancel(uint256 proposalId) external {
  if(_getProposalState(proposalId) == ProposalState.Executed) revert InvalidProposalState();
  Proposal storage proposal = proposals[proposalId];
  if(msg.sender != proposal.proposer) revert NotProposer();
  if(GovLocks(govlocks).getPriorVotes(proposal.proposer, block.number - 1) > proposalThreshold)
  ↪  revert AboveThreshold(); //@audit incorrect
  proposal.cancelled = true;
  uint256 targetsLength = proposal.targets.length;
  for (uint256 i = 0; i < targetsLength; i++) {
    Timelock(timelock).cancelTransaction(proposal.targets[i], proposal.eta, proposal.values[i],
    ↪  proposal.calldatas[i], proposal.signatures[i]);
  }
  emit ProposalCanceled(proposalId);
}
```

Impact: A proposer can't cancel his proposal unless he decreases his voting power.

Recommended Mitigation: It should be modified like this.

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

Client: Fixed in PR #7

Cyfrin: Verified.

7.2.4 Users wouldn't cancel their proposals due to the increased `proposalThreshold`.

Severity: Medium

Description: When users call `cancel()`, it validates the caller's voting power with `proposalThreshold` which can be changed using `setProposalThreshold()`.

```
function setProposalThreshold(uint256 newProposalThreshold) external {
  if(msg.sender != multisig) revert NotMultisig();
  if(newProposalThreshold < MIN_PROPOSAL_THRESHOLD || newProposalThreshold > MAX_PROPOSAL_THRESHOLD)
  ↪  revert InvalidVotingParameter();
  uint256 oldProposalThreshold = proposalThreshold;
  proposalThreshold = newProposalThreshold;
  emit ProposalThresholdSet(oldProposalThreshold, proposalThreshold);
}
```

Here is a possible scenario.

- Let's assume `proposalThreshold = 100` and a user has 100 voting power.
- The user has proposed a proposal using `propose()`.
- After that, `proposalThreshold` was increased to 150 by `multisig`.
- When the user calls `cancel()`, it will revert as he doesn't have enough voting power.

Impact: Users wouldn't cancel their proposals due to the increased `proposalThreshold`.

Recommended Mitigation: It would be good to cache `proposalThreshold` as a proposal state.

Client: Acknowledged, we will ensure to only change parameters while there are no pending proposals.

Cyfrin: Acknowledged.

7.2.5 `Goldilend.liquidate()` might revert due to underflow

Severity: Medium

Description: In `repay()`, there would be a rounding during the `interest` calculation.

```
  function repay(uint256 repayAmount, uint256 _userLoanId) external {
      Loan memory userLoan = loans[msg.sender][_userLoanId];
      if(userLoan.borrowedAmount < repayAmount) revert ExcessiveRepay();
      if(block.timestamp > userLoan.endDate) revert LoanExpired();
      uint256 interestLoanRatio = FixedPointMathLib.divWad(userLoan.interest, userLoan.borrowedAmount);
L425  uint256 interest = FixedPointMathLib.mulWadUp(repayAmount, interestLoanRatio); //@audit rounding
↪  issue
      outstandingDebt -= repayAmount - interest > outstandingDebt ? outstandingDebt : repayAmount -
      ↪  interest;
      ...
  }
...
  function liquidate(address user, uint256 _userLoanId) external {
      Loan memory userLoan = loans[msg.sender][_userLoanId];
      if(block.timestamp < userLoan.endDate || userLoan.liquidated || userLoan.borrowedAmount == 0)
      ↪  revert Unliquidatable();
      loans[user][_userLoanId].liquidated = true;
      loans[user][_userLoanId].borrowedAmount = 0;
L448  outstandingDebt -= userLoan.borrowedAmount - userLoan.interest;
      ...
  }
```

Here is a possible scenario.

- There are 2 borrowers of `borrowedAmount = 100, interest = 10`. And `outstandingDebt = 2 * (100 - 10) = 180`.
- The first borrower calls `repay()` with `repayAmount = 100`.
- Due to the rounding issue at L425, `interest` is 9 instead of 10. And `outstandingDebt = 180 - (100 - 9) = 89`.
- In `liquidate()` for the second borrower, it will revert at L448 because `outstandingDebt = 89 < borrowedAmount - interest = 90`.

Impact: `liquidate()` might revert due to underflow.

Recommended Mitigation: In `liquidate()`, `outstandingDebt` should be updated like the below.

```
  /// @inheritdoc IGoldilend
  function liquidate(address user, uint256 _userLoanId) external {
    Loan memory userLoan = loans[msg.sender][_userLoanId];
    if(block.timestamp < userLoan.endDate || userLoan.liquidated || userLoan.borrowedAmount == 0) revert
    ↪  Unliquidatable();
    loans[user][_userLoanId].liquidated = true;
    loans[user][_userLoanId].borrowedAmount = 0;
+  uint256 debtToRepay = userLoan.borrowedAmount - userLoan.interest;
+  outstandingDebt -= debtToRepay > outstandingDebt ? outstandingDebt : debtToRepay;
   ...
  }
```

Client: Fixed in PR #10

Cyfrin: Verified.

7.2.6 In `Goldigovernor`, wrong assumption of block time

Severity: Medium

Description: In `Goldigovernor.sol`, voting period/delay limits are set with 15s block time.

```
/// @notice Minimum voting period
uint32 public constant MIN_VOTING_PERIOD = 5760; // About 24 hours

/// @notice Maximum voting period
uint32 public constant MAX_VOTING_PERIOD = 80640; // About 2 weeks

/// @notice Minimum voting delay
uint32 public constant MIN_VOTING_DELAY = 1;

/// @notice Maximum voting delay
uint32 public constant MAX_VOTING_DELAY = 40320; // About 1 week
```

But Berachain has 5s block time according to its documentation.

```
Berachain has the following properties:

- Block time: 5s
```

So these limits will be set shorter than expected.

Impact: Voting period/delay limits will be set shorter than expected.

Recommended Mitigation: We should calculate these limits with 5s block time.

Client: Fixed in PR #14

Cyfrin: Verified.

7.3 Low Risk

7.3.1 `Goldivault.changeProtocolParameters()` shouldn't update `endTime`.

Description: `Goldivault.changeProtocolParameters()` updates `endTime` only without changing `startTime`.

```
function changeProtocolParameters(
  uint256 _earlyWithdrawalFee,
  uint256 _yieldFee,
  uint256 _delay,
  uint256 _duration
) external {
  if(msg.sender != timelock) revert NotTimelock();
  earlyWithdrawalFee = _earlyWithdrawalFee;
  yieldFee = _yieldFee;
  delay = _delay;
  duration = _duration;
  endTime = block.timestamp + _duration;
}
```

It's more appropriate not to update `endTime` as this function is just to change parameters.

Client: Fixed in PR #11

Cyfrin: Verified.

7.3.2 `Goldilocked._lockedLocks()` should round up.

Description: Users might borrow 1 more wei as it rounds down.

```
function _lockedLocks(address user) internal view returns (uint256) {
  return FixedPointMathLib.divWad(borrowedHoney[user], IGoldiswap(goldiswap).floorPrice());//@audit
  ↪  round up
}
```

Client: Fixed in PR #15

Cyfrin: Verified.

7.3.3 Possible failure to redeem

Description: In `Goldivault`, `redeemOwnership()` wouldn't work as expected after calling `changeProtocolParameters/initializeProtocol()` because `endTime/duration` is changed.

Client: Acknowledged, we plan to only call changeProtocolParameters after vault has concluded and ownership token holders have had chance to redeem. We will announce that we are going to update parameters and renew.

Cyfrin: Acknowledged.

7.3.4 Inconsistent comparison while checking `eta`

Description: In `Goldigovernor` and `Timelock`, inconsistent comparisons are used.

```
File: Goldigovernor.sol
386:     else if (block.timestamp >= proposal.eta + Timelock(timelock).GRACE_PERIOD()) {
387:       return ProposalState.Expired;
388:     }

File: Timelock.sol
138:     if(block.timestamp > eta + GRACE_PERIOD) revert TxStale();
```

Client: Fixed in PR #6

Cyfrin: Verified.

7.3.5 Wrong comment

Description:

```
File: Timelock.sol
49:   /// @notice Delay for queueing a transaction in blocks //@audit seconds, not blocks
60:   /// @param _delay Delay the timelock will use, in blocks //@audit seconds, not blocks
103:   /// @param eta Duration of time until transaction can be executed, in blocks //@audit seconds,
↪  not blocks
123:   /// @param eta Duration of time until transaction can be executed, in blocks //@audit seconds,
↪  not blocks
154:   /// @param eta Duration of time until transaction can be executed, in blocks //@audit seconds,
↪  not blocks

File: Goldilend.sol
496:   /// @notice Calculates claimable Porridge per GiBGT //@audit claimable reward token
608:   /// @dev Claims existing vault rewards and updates poolSize //@audit this function doesn't update
↪  poolSize
```

Client: Fixed in PR #16

Cyfrin: Verified.

7.3.6 Missing bracket in docs

Description: The document is missing a bracket.

```
- Market price = Floor Price + ((PSL/Supply)*(FSL+PSL/FSL)^6)
+ Market price = Floor Price + (PSL/Supply)*((FSL+PSL)/FSL)^6
```

Client: Acknowledged, this has been corrected.

Cyfrin: Acknowledged.

7.3.7 Rounding loss of 1 wei

Description: In `Goldivault`, sum of 2 amounts might be less than `amount` due to the rounding loss.

```
File: Goldivault.sol
    if(remainingTime > 0) {
      SafeTransferLib.safeTransfer(depositToken, msg.sender, amount * (1000 - _fee) / 1000);
      SafeTransferLib.safeTransfer(depositToken, multisig, amount * _fee / 1000); //@audit rounding loss
      emit OwnershipTokenRedemption(msg.sender, amount * (1000 - _fee) / 1000);
    }

File: Goldilend.sol
    poolSize += userLoan.interest * (1000 - (multisigShare + apdaoShare)) / 1000;
    _updateInterestClaims(interest); //@audit rounding loss
...
  function _updateInterestClaims(uint256 interest) internal {
    multisigClaims += interest * multisigShare / 1000;
    apdaoClaims += interest * apdaoShare / 1000;
  }
```

Client: Fixed in PR #17

Cyfrin: Verified.

7.4 Informational

7.4.1 Unused constant

```
File: Goldigovernor.sol
  /// @notice Amount of votes to reach quorum
  uint256 public constant quorumVotes = 9_500_000e18; // 5% of LOCKS
```

7.4.2 Bad design to pull `HONEY` twice from the user

Users should approve the transfer twice during the interaction. It would be better to pull the whole amount from the user at a time and transfer to `multisig` from the contract.

```
File: Goldiswap.sol
    SafeTransferLib.safeTransferFrom(honey, msg.sender, address(this), price);
    SafeTransferLib.safeTransferFrom(honey, msg.sender, multisig, tax);
```

7.5 Gas Optimization

7.5.1 Constructors can be marked as `payable`.

Payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided. A constructor can safely be marked as `payable`, since only the deployer would be able to pass funds, and the project itself would not pass any funds. (9 Instances)

7.5.2 Nesting `if`-statements is cheaper than using `andand`.

Nesting `if`-statements avoids the stack operations of setting up and using an extra jumpdest, and saves 6 gas.

```
File: GovLocks.sol
214:     if (srcRep != dstRep && amt > 0) {
237:     if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == block.number) {
266:     if(from != address(0) && to != address(0)) {

File: Goldilend.sol
536:     return _poolSize > 0 && supply > 0 ? FixedPointMathLib.mulWad(lockAmount, _GiBGTRatio(supply,
↪   _poolSize)) : lockAmount;

File: Goldiswap.sol
329:     if (elapsedIncrease >= 1 days && elapsedDecrease >= 1 days) {
```

7.5.3 Use `uint256(1)/uint256(2)` instead of `true/false` to save gas for changes.

Avoids 20000 gas when changing from `false` to `true`, after having been true in the past.

```
File: Goldilend.sol
107:    bool public borrowingActive;

File: Goldivault.sol
91:   bool public concluded;
```

7.5.4 Augmented assignment operator costs more gas than normal addition for state variables.

Normal addition operation (`x=x+y`) costs less gas than augmented assignment operator (`x+=y`) for state variables (113 gas). (34 instances)

7.5.5 Cache array length outside of loops and consider unchecked loop incrementing.

```
File: Goldilend.sol
251:    function boost(
252:      address[] calldata partnerNFTs,
253:      uint256[] calldata partnerNFTIds
254:    ) external {
255:      for(uint256 i; i < partnerNFTs.length; i++) {
256:        if(partnerNFTBoosts[partnerNFTs[i]] == 0) revert InvalidBoostNFT();
257:      }
258:      if(partnerNFTs.length != partnerNFTIds.length) revert ArrayMismatch();
259:      boosts[msg.sender] = _buildBoost(partnerNFTs, partnerNFTIds);
260:      for(uint8 i; i < partnerNFTs.length; i++) {
261:        IERC721(partnerNFTs[i]).safeTransferFrom(msg.sender, address(this), partnerNFTIds[i]);
262:      }
263:    }
```