Idle Finance Yield Tranches Security Review

Reviewer
Hans

September 2, 2023

Contents

1 Executive Summary

Over the course of 5 business days in total, Idle Finance engaged with Hans to review idle-tranches.

Summary

Type of Project	Defi
Timeline	28th Aug, 2023 - 1st Sep, 2023
Methods	Manual Review
Documentation	High
Testing Coverage	High

A comprehensive security review identified a total of 7 issues and 8 Gas optimization suggestions.

Repository	**Initial Commit**
Idle Finance	119de741c82426602fdfa1a948a8db72c00de802

Total Issues

High Risk	0
Medium Risk	4
Low Risk	2
Informational	1
Gas Optimization	8

The reported vulnerabilities were addressed by the Idle Finance team, and the mitigation underwent a review process and was verified by Hans.

Repository	**Final Commit**
Idle Finance	f2d5d9a6f88a8c4810052768195f23d33b80e09a

2 Scope of the Audit

This incremental audit was conducted for 2 files:

- `IdleCDO.sol` - Main file with all the logic for tranches management and for eventual loss management.
- `IdleCDOInstadappLiteVariant.sol` - A variant of `IdleCDO.sol` with some additional logic to handle Instadapp Lite integration for iETHv2 (which has withdrawal fees).

3 About Hans

Hans is an esteemed security analyst in the realm of smart contracts, boasting a firm grounding in mathematics that has sharpened his logical abilities and critical thinking skills. These attributes have fast-tracked his journey to the peak of the Code4rena leaderboard, marking him as the number one auditor in a record span of time. In addition to his auditor role, he also serves as a judge on the same platform. Hans' innovative insight is evident in his creation of Solodit, a vital resource for navigating consolidated security reports. In addition, he is a co-founder of Cyfrin, where he is dedicated to enhancing the security of the blockchain ecosystem through continuous efforts.

4 Disclaimer

I endeavor to meticulously identify as many vulnerabilities as possible within the designated time frame; however, I must emphasize that I cannot accept liability for any findings that are not explicitly documented herein. It is essential to note that my security audit should not be construed as an endorsement of the underlying business or product. The audit was conducted within a specified timeframe, with a sole focus on evaluating the security aspects of the solidity implementation of the contracts.

While I have exerted utmost effort in this process, I must stress that I cannot guarantee absolute security. It is a well-recognized fact that no system can be deemed completely impervious to vulnerabilities, regardless of the level of scrutiny applied.

5 Protocol Summary

Idle DAO is a decentralized collective that built a set of products that aim to unlock the power of decentralized finance as a one-stop source of yield. It allows users to algorithmically optimize their digital asset allocation across leading DeFi protocols, whether they want to maximize it or keep tabs on their risk-return profile. The product to be audited is called Yield Tranches (YT) and allows users to deposit assets in a Defi protocol with two risk-adjusted investment profiles (Senior and Junior).

6 Additional Comments

The security assessment was carried out with a narrow focus on the contracts. Due to time limitations and the incremental nature of the reviews, the results might not be comprehensive and might not represent the complete security profile of the protocol.

7 Findings

7.1 Medium Risk

7.1.1 Read-only reentrancy in `IdleCDO::_withdraw()` and `IdleCDOInstadappLiteVariant::_withdraw()`

Severity: Medium

Context: IdleCDO.sol#L499, IdleCDOInstadappLiteVariant.sol#L67

Description: The current implementation of `IdleCDO::_withdraw()` and `IdleCDOInstadappLiteVariant::_withdraw()` are against the CEI pattern and vulnerable to read-only reentrancy. `IdleCDO::_withdraw()` is implemented as below and we can see that the underlying tokens are sent to the sender before updating the protocol's core accounting state variables. Because of the `nonReentrant` modifier, it is not directly exploitable via a normal reentrancy attack but it is still vulnerable to a read-only reentrancy attack. It is worthy noting that the `lastNAVAA` and `lastNAVBB` are used in the function `virtualPrice()`, so if the attacker query the `virtualPrice()` of the tranche price in the transfer callback (if any), the protocol will return a wrong price. Assuming the underlying tokens are carefully allowed by the Idle team, the likelihood is low for this function. The similar issue exists in the function `IdleCDOInstadappLiteVariant::_withdraw()` as well.

```
IdleCDO.sol
  function _withdraw(uint256 _amount, address _tranche) virtual internal nonReentrant returns (uint256 toRedeem) {
    // check if a deposit is made in the same block from the same user
    _checkSameTx();
    // check if _strategyPrice decreased
    _checkDefault();
    // accrue interest to tranches and updates tranche prices
    _updateAccounting();
    // redeem all user balance if 0 is passed as _amount
    if (_amount == 0) {
      _amount = IERC20Detailed(_tranche).balanceOf(msg.sender);
    }
    require(_amount > 0, '0');
    address _token = token;
    // get current available unlent balance
    uint256 balanceUnderlying = _contractTokenBalance(_token);
    // Calculate the amount to redeem
    toRedeem = _amount * _tranchePrice(_tranche) / ONE_TRANCHE_TOKEN;
    if (toRedeem > balanceUnderlying) {
      // if the unlent balance is not enough we try to redeem what's missing directly from the strategy
      // and then add it to the current unlent balance
      // NOTE: A difference of up to 100 wei due to rounding is tolerated
      toRedeem = _liquidate(toRedeem - balanceUnderlying, revertIfTooLow) + balanceUnderlying;
    }
    // burn tranche token
    IdleCDOTranche(_tranche).burn(msg.sender, _amount);
    // send underlying to msg.sender
    IERC20Detailed(_token).safeTransfer(msg.sender, toRedeem);//@audit-issue aginst CEI pattern

    // update NAV with the _amount of underlyings removed
    if (_tranche == AATranche) {
      lastNAVAA -= toRedeem;
    } else {
      lastNAVBB -= toRedeem;
    }

    // update trancheAPRSplitRatio
    _updateSplitRatio(_getAARatio(true));
  }
```

Although it is not strictly in scope, we can see that the same vulnerability exists in `IdleCDOPoLidoVariant::_withdraw()` and the likelihood is high because standard NFT interfaces include the `onERC721Received()` function which is called by the `safeTransferFrom()` function.

Impact The attacker can query the wrong tranche price in the transfer callback and this can further be exploited for other attacks.

Recommendation: Follow CEI pattern and update the protocol's core accounting state variables before sending the underlying tokens to the sender.

Client: The finding is valid but the likelihood on tranches without NFTs is basically null as the underlying tokens are carefully allowed by the Idle DAO team. The likelihood on tranches with NFTs (IdleCDOPoLidoVariant) is possible but due to the non easy composability of this tranche we don't expect integrators to use this. Nonetheless we will fix this issue in the next version of IdleCDO.

Hans: Verified the fix at commit c315183 and 209125c

7.1.2 The `trancheAPRSplitRatio` is updated using a slightly stale prices in the function `_withdraw()` in case that excess tokens are received from the strategy

Severity: Medium

Context: IdleCDO.sol#L509

Description: If a user requests withdrawal of more than available underlying tokens, the function `_withdraw()` will call `_liquidate()` to redeem the missing tokens from the strategy. The function `_liquidate()` will return the amount of tokens that were actually redeemed from the strategy capped to the requested amount.

```
IdleCDO.sol
565:   function _liquidate(uint256 _amount, bool _revertIfNeeded) internal virtual returns (uint256
↪  _redeemedTokens) {
566:     _redeemedTokens = IIdleCDOStrategy(strategy).redeemUnderlying(_amount);
567:     if (_revertIfNeeded) {
568:       uint256 _tolerance = liquidationTolerance;
569:       if (_tolerance == 0) {
570:         _tolerance = 100;
571:       }
572:       // keep `_tolerance` wei as margin for rounding errors
573:       require(_redeemedTokens + _tolerance >= _amount, '5');
574:     }
575:
576:     if (_redeemedTokens > _amount) {
577:       _redeemedTokens = _amount;//@follow-up capped amount is returned while it is not clear what
↪  happens with the excess tokens
578:     }
579:   }
```

The function `_withdraw()` updates the `trancheAPRSplitRatio` at the end of the execution using the `getAARatio(true)`.

```
File: d:\hans\solo\idle\idle-tranches\contracts\IdleCDO.sol
473:   function _withdraw(uint256 _amount, address _tranche) virtual internal nonReentrant returns
↪  (uint256 toRedeem) {
    ...
501:     // update NAV with the _amount of underlyings removed//@audit-issue at this point there can be
↪  excess amount redeemed from the strategy and priceAA and priceBB are not up to date
502:     if (_tranche == AATranche) {
503:       lastNAVAA -= toRedeem;
504:     } else {
505:       lastNAVBB -= toRedeem;
506:     }
507:
508:     // update trancheAPRSplitRatio
509:     _updateSplitRatio(_getAARatio(true));//@audit-issue ratio is updated with old slightly stale
↪  prices
510:   }
```

This means that the `trancheAPRSplitRatio` will be updated using stale prices. It is understood that normally the strategy returns at most the requested amount of tokens, but it is not clear what happens if the strategy returns more tokens than requested. Assuming the amount of excess tokens will be small, the impact on the price is expected to be small as well.

Impact `trancheAPRSplitRatio` is updated using slightly stale prices in case that the strategy returns more tokens than requested.

Recommendation: Consider call `_updateAccounting()` before `_updateSplitRatio()` to update the `trancheAPRSplitRatio` using the latest prices.

Client: Acknowledged. Even if we get 1 underlying more when redeeming from strategy (which would be really a lot) and supposing this will slightly change the TVL ratio we would still avoid calling `updateAccounting` at the end of withdraw as the accounting would only slightly change the interest split until the next `updateAccounting` call, but at the expense of having all withdrawals to be way more expensive in terms of gas which is something we would like to avoid as such call have high gas costs.

Hans: Acknowledged.

7.1.3 `latestHarvestBlock` should be updated first before calling `setReleaseBlocksPeriod()`

Severity: Medium

Context: IdleCDO.sol#L896

Description: If the admin updates `releaseBlocksPeriod` when `latestHarvestBlock != block.number`, `_lockedRewards()` will return the incorrect amount if there are rewards being unlocked.

Consider the following scenario.

1. At block 0, `releaseBlocksPeriod = 6400(1 day)` and `harvestedRewards = 100` after calling `harvest()`. `latestHarvestBlock= 0`.
2. At block 3200, during the normal deposit, `_lockedRewards()` will return `100 * 3200 / 6400 = 50` and it will be used through `_deposit() => updateAccounting() => getContractValue() => lockedRewards()` to update the tranche prices.
3. After that at the same block, the admin called `setReleaseBlocksPeriod()` with `12800(2 days)` and `_lockedRewards()` will return `100 * (12800 - 3200) / 12800 = 75`. It means already unlocked 25 rewards are locked again.

Impact The reward distribution logic using `_lockedRewards()` wouldn't work as expected after calling `setReleaseBlocksPeriod()`.

Recommendation: `setReleaseBlocksPeriod()` should call harvest() first or update `latestHarvestBlock`(should be equal `block.number`) and `harvestedRewards` accordingly to return the same `_lockedRewards` after the update.

Client: Acknowledged. In general I think this is valid, but I don't think we ever called `setReleaseBlocksPeriod` and even if called during a release of rewards we would only posticipate a bit the rewards release (again valid finding but not really concerning). Regarding the recommendation, I would exclude calling `harvest` as it requires a lot of 'logic' (ie params that we need to calculate) and this may not always be possible to do easily (eg if multisig is doing the tx as it's the owner of the contract). If we set `latestHarvestBlock` equal to `block.number` and there is an active distribution we would completely restart the unlock period I think which is probably worse than the currect solution. The easiest thing would be to add a comment for the method but not enforcing this with code, not ideal but better than nothing.

Hans: Acknowledged.

7.1.4 Inconsistent update of `lastNAVAA/lastNAVBB` in `_withdraw()`

Severity: Medium

Context: IdleCDO.sol#L503

IdleCDOInstadappLiteVariant.sol#L117

Description: `IdleCDO._withdraw()` uses inconsistent logic to update `lastNAVAA/lastNAVBB`.

In `IdleCDO._withdraw()`, it uses `toRedeem` after applying the tolerance.

```
toRedeem = _amount * _tranchePrice(_tranche) / ONE_TRANCHE_TOKEN;
   if (toRedeem > balanceUnderlying) {
     // if the unlent balance is not enough we try to redeem what's missing directly from the strategy
     // and then add it to the current unlent balance
     // NOTE: A difference of up to 100 wei due to rounding is tolerated
     toRedeem = _liquidate(toRedeem - balanceUnderlying, revertIfTooLow) + balanceUnderlying;
   }
   // burn tranche token
   IdleCDOTranche(_tranche).burn(msg.sender, _amount);
   // send underlying to msg.sender
   IERC20Detailed(_token).safeTransfer(msg.sender, toRedeem);

   // update NAV with the _amount of underlyings removed
   if (_tranche == AATranche) {
     lastNAVAA -= toRedeem;
   } else {
     lastNAVBB -= toRedeem;
   }
```

If `toRedeem` was decreased due to the tolerance during `_liquidate()`, it uses the reduced amount for `lastNAVAA`/`lastNAVBB` updates.

But in `IdleCDOInstadappLiteVariant._withdraw()`, it applies the whole amount when `_paidFee > _expectedFee`.

```
    uint256 _want = toRedeem;
    // calculate expected fee
    uint256 _expectedFee = _calcUnderlyingProtocolFee(toRedeem);
    // actual fee paid, considering the unlent balance present in this contract
    // this value should be lte than _expectedFee as only a portion of the toRedeem
    // will be redeemed from the lending provider if there is some unlent balance
    uint256 _paidFee;
    if (toRedeem > balanceUnderlying) {
        // if the unlent balance is not enough we try to redeem what's missing directly from the
↪   strategy
        // and then add it to the current unlent balance
        (toRedeem, _paidFee) = _liquidateWithFee(toRedeem - balanceUnderlying, revertIfTooLow);
        // add the unlent balance to the redeemed amount
        toRedeem += balanceUnderlying;
        // be sure to remove the missing fee, even when using
        // the unlent balance the user should pay the full fee
        if (_paidFee < _expectedFee) {
            toRedeem -= (_expectedFee - _paidFee);
        }
    } else {
        // user is redeeming all from the unlent balance but the fee is still applied,
        // the pool is 'gaining' the _expectedFee which will increase the lastNAV
        toRedeem -= _expectedFee;
    }

    // burn tranche token
    IdleCDOTranche(_tranche).burn(msg.sender, _amount);
    // send underlying to msg.sender
    IERC20Detailed(_token).safeTransfer(msg.sender, toRedeem);

    // update NAV with the _amount of underlyings removed (eventual fee gained is not
    // considered here so virtualPrice will be updated accordingly)
    if (_tranche == AATranche) {
        lastNAVAA -= _want;
    } else {
        lastNAVBB -= _want;
    }
```

Let's assume `_want = 1000, toRedeem = 900, _paidFee = 100, _expectedFee = 50` after calling `_liquidateWithFee()`.

If we apply the same logic as `IdleCDO._withdraw()` including the `_expectedFee`, it should reduce `toRedeem + _expectedFee = 950` because `_paidFee - _expectedFee` contains the tolerance that is ignored in `IdleCDO`.

I think `IdleCDOInstadappLiteVariant`'s logic is correct and `IdleCDO._withdraw()` should use the original `toRedeem` before calling `_liquidate()`.

Impact `lastNAVAA/lastNAVBB` might be tracked incorrectly after the liquidation.

Recommendation: `IdleCDO` and `IdleCDOInstadappLiteVariant` should use the same logic to update `lastNAVAA/lastNAVBB`.

Client: The finding is correct. We will fix this in the next version of `IdleCDO.sol`

Hans: Verified the fix at commit c315183.

7.2 Low Issues

7.2.1 `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead

```
File: IdleCDO.sol

1010:     _lastCallerBlock = keccak256(abi.encodePacked(tx.origin, block.number));

1015:     require(keccak256(abi.encodePacked(tx.origin, block.number)) != _lastCallerBlock, "8");
```

Client: Acknowledged. There are no direct issues of collision here as we use it with `address` + `uint256`

Hans: Acknowledged.

7.2.2 Do not use deprecated library functions

`safeApprove` is deprecated in favor of `safeIncreaseAllowance` and `safeDecreaseAllowance`, refer to here.

```
File: IdleCDO.sol

998:     IERC20Detailed(_token).safeApprove(_spender, 0);
```

Client: We will remove the function `_removeAllowance()` because it is not used anywhere.

Hans: Verified it's fixed at commit f2d5d9a.

7.3 Informational Findings

7.3.1 `require()` / `revert()` statements should have descriptive reason strings

```
File: IdleCDO.sol

811:     require(_maxDecreaseDefault < FULL_ALLOC);
```

Client: Fixed at commit f2d5d9a.

Hans: Verified.

7.4 Gas Optimizations

7.4.1 Use assembly to check for `address(0)`

Refer to here for more details.

```
File: IdleCDO.sol

63:     require(token == address(0), '1');

64:     require(_rebalancer != address(0) && _strategy != address(0) && _guardedToken != address(0),
↪   "0");

64:     require(_rebalancer != address(0) && _strategy != address(0) && _guardedToken != address(0),
↪   "0");

64:     require(_rebalancer != address(0) && _strategy != address(0) && _guardedToken != address(0),
↪   "0");

290:     if (_referral != address(0)) {

848:     require((rebalancer = _rebalancer) != address(0), '0');//@audit-ok

854:     require((feeReceiver = _feeReceiver) != address(0), '0');//@audit-ok

860:     require((guardian = _guardian) != address(0), '0');//@audit-ok
```

Client: We will keep it as is as the gain should be minimal but more readable.

Hans: Acknowledged.

7.4.2 Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
File: IdleCDO.sol

658:     for (uint256 i = 0; i < _rewards.length; i++) {
```

Client: Fixed at commit f2d5d9a.

Hans: Verified.

7.4.3 Use Custom Errors

Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost. Refer to here for more details.

```
File: IdleCDO.sol

64:     require(_rebalancer != address(0) && _strategy != address(0) && _guardedToken != address(0),
↪   "0");

320:      require(skipDefaultCheck, "4");

551:      require(lastStrategyPrice * (FULL_ALLOC - maxDecreaseDefault) / FULL_ALLOC <= currPrice,
↪   "4");

979:     require(msg.sender == guardian || msg.sender == owner(), "6");

984:     require(msg.sender == rebalancer || msg.sender == owner(), "6");

1015:     require(keccak256(abi.encodePacked(tx.origin, block.number)) != _lastCallerBlock, "8");
```

```
File: IdleCDOPoLidoVariant.sol

51:        require(_amount > 0, "0");

65:        require(tokenIds.length != 0, "no NFTs");
```

Client: Acknowledged.

Hans: Acknowledged.

7.4.4 Don't initialize variables with default value

```
File: IdleCDO.sol

658:     for (uint256 i = 0; i < _rewards.length; i++) {
```

Client: Fixed at commit f2d5d9a.

Hans: Verified.

7.4.5 `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

```
File: IdleCDO.sol

658:     for (uint256 i = 0; i < _rewards.length; i++) {
```

Client: Fixed at commit f2d5d9a.

Hans: Verified.

7.4.6 Use `private` rather than `public` for constants

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

```
File: IdleCDOPoLidoVariant.sol

20:     IStMATIC public constant stMatic = IStMATIC(0x9ee91F9f426fA633d227f7a9b000E28b9dfd8599);
```

Client: Acknowledged. Won't fix because those pulbic constants might be being used by other contracts.

Hans: Acknowledged.

7.4.7 Splitting require() statements that use && saves gas

```
File: IdleCDO.sol

64:     require(_rebalancer != address(0) && _strategy != address(0) && _guardedToken != address(0),
↪   "0");
```

Client: Fixed at commit f2d5d9a.

Hans: Verified.

7.4.8 Use != 0 instead of > 0 for unsigned integer comparison

```
File: IdleCDO.sol

245:     if (_stkIDLEPerUnderlying > 0) {

393:     if (totalGain > 0) {

406:       if (totalGain > 0) {

422:         if (_newJuniorTVL > 0) {

460:     if (_amount > 0) {

484:     require(_amount > 0, '0');

599:     if (_path.length > 0) {

651:     if (_extraData.length > 0) {
```

```
File: IdleCDOPoLidoVariant.sol

51:         require(_amount > 0, "0");
```

Client: The finding does not apply for `totalGain` and `_newJuniorTVL` as they are signed integers. Fixed others at commit f2d5d9a.

Hans: Verified.