Sublime contest
Findings & Analysis Report

2022-07-14

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Sublime smart contract system written in Solidity. The audit contest took place between March 29—March 31 2022.

Wardens

25 Wardens contributed reports to the Sublime contest:

1. hickuphh3
2. WatchPug (jtp and ming)
3. kyliek
4. rayn
5. Dravee

6. MetaOxNull
7. lllllll
8. kenta
9. robee
10. 0x1f8b
11. defsec
12. 0xkatana
13. gzeon
14. sseefried
15. 0v3rf10w
16. hake
17. 0xDjango
18. BouSalman
19. dirk_y
20. rfa
21. Funen
22. Tomio
23. 0xNazgul
24. csanuragjain

This contest was judged by HardlyDifficult.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 6 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 18 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 19 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Sublime contest repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 1,936 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] `LenderPool` : Principal withdrawable is incorrectly calculated if start() is invoked with non-zero start fee

Submitted by hickuphh3

LenderPool.sol#L594-L599
LenderPool.sol#L399-L404

The `_principalWithdrawable` calculated will be more than expected if `_start()` is invoked with a non-zero start fee, because the borrow limit is reduced by the fee, resulting in `totalSupply[id]` not being 1:1 with the borrow limit.

```
function _calculatePrincipalWithdrawable(uint256 _id, address _l
  uint256 _borrowedTokens = pooledCLConstants[_id].borrowLimit;
  uint256 _totalLiquidityWithdrawable = _borrowedTokens.sub(POOL
  uint256 _principalWithdrawable = _totalLiquidityWithdrawable.m
  return _principalWithdrawable;
}
```

Proof of Concept

Assume the following conditions:

- Alice, the sole lender, provided `100_000` tokens: `totalSupply[_id] = 100_000`
- `borrowLimit = 99_000` because of a 1% startFee
- Borrower borrowed zero amount

When Alice attempts to withdraw her tokens, the `_principalWithdrawable` amount is calculated as

```
_borrowedTokens = 99_000
_totalLiquidityWithdrawable = 99_000 - 0 = 99_000
_principalWithdrawable = 99_000 * 100_000 / 99_000 = 100_000
```

This is more than the available principal amount of `99_000`, so the withdrawal will fail.

Recommended Mitigation Steps

One hack-ish way is to save the initial supply in `minBorrowAmount` (perhaps rename the variable to `minInitialSupply`) when the credit line is accepted, and replace `totalSupply[_id]` with it.

The other places where `minBorrowAmount` are used will not be affected by the change because:

- startTime has been zeroed, so `start()` cannot be invoked (revert with error S1)
- credit line status would have been changed to `ACTIVE` and cannot be changed back to `REQUESTED`, meaning the check below will be false regardless of the value of `minBorrowAmount`.

```
_status == PooledCreditLineStatus.REQUESTED &&
block.timestamp > pooledCLConstants[_id].startTime &&
totalSupply[_id] < pooledCLConstants[_id].minBorrowAmount
```

Code amendment example:

```
function _accept(uint256 _id, uint256 _amount) internal {
  ...
  // replace delete pooledCLConstants[_id].minBorrowAmount; with
  pooledCLConstants[_id].minInitialSupply = totalSupply[_id];
}

// update comment in _withdrawLiquidity
// Case 1: Pooled credit line never started because desired amou
// state will never revert back to REQUESTED if credit line is a

function _calculatePrincipalWithdrawable(uint256 _id, address _l
  uint256 _borrowedTokens = pooledCLConstants[_id].borrowLimit;
  uint256 _totalLiquidityWithdrawable = _borrowedTokens.sub(POOL
  // totalSupply[id] replaced with minInitialSupply
  uint256 _principalWithdrawable = _totalLiquidityWithdrawable.m
  return _principalWithdrawable;
}
```

In `terminate()`, the shares withdrawable can simply be `_sharesHeld`.

```
function terminate(uint256 _id, address _to) external override o
  address _strategy = pooledCLConstants[_id].borrowAssetStrategy
  address _borrowAsset = pooledCLConstants[_id].borrowAsset;
  uint256 _sharesHeld = pooledCLVariables[_id].sharesHeld;

  SAVINGS_ACCOUNT.withdrawShares(_borrowAsset, _strategy, _to, _
  delete pooledCLConstants[_id];
  delete pooledCLVariables[_id];
```

```
}
```

ritik99 (Sublime) confirmed

[H-02] `PooledCreditLine` : termination likely fails because `_principleWithdrawable` is treated as shares

Submitted by hickuphh3, also found by rayn and WatchPug

LenderPool.sol#L404-L406

`_principalWithdrawable` is denominated in the borrowAsset, but subsequently treats it as the share amount to be withdrawn.

```
// _notBorrowed = borrowAsset amount that isn't borrowed
// totalSupply[_id] = ERC1155 total supply of _id
// _borrowedTokens = borrower's specified borrowLimit
uint256 _principalWithdrawable = _notBorrowed.mul(totalSupply[_i

SAVINGS_ACCOUNT.withdrawShares(_borrowAsset, _strategy, _to, _pr
```

Recommended Mitigation Steps

The amount of shares to withdraw can simply be `_sharesHeld`.

Note that this comes with the assumption that `terminate()` is only called when the credit line is `ACTIVE` or `EXPIRED` (consider ensuring this condition on-chain), because `_sharesHeld` **excludes principal withdrawals**, so the function will fail once a lender withdraws his principal.

```
function terminate(uint256 _id, address _to) external override o
  address _strategy = pooledCLConstants[_id].borrowAssetStrategy
  address _borrowAsset = pooledCLConstants[_id].borrowAsset;
  uint256 _sharesHeld = pooledCLVariables[_id].sharesHeld;

  SAVINGS_ACCOUNT.withdrawShares(_borrowAsset, _strategy, _to, _
  delete pooledCLConstants[_id];
```

```
      delete pooledCLVariables[_id];
    }
```

ritik99 (Sublime) confirmed

Medium Risk Findings (4)

[M-01] Pool Credit Line May Not Able to Start When _borrowAsset is Non ERC20 Compliant Tokens

Submitted by MetaOxNull, also found by Dravee and kenta

```
IERC20(_borrowAsset).transfer(_to, _fee);
```

If the USDT token is supported as _borrowAsset, the unsafe version of .transfer(_to, _fee) may revert as there is no return value in the USDT token contract's transfer() implementation (but the IERC20 interface expects a return value).

Function start() will break when _borrowAsset is USDT or Non ERC20 Compliant Tokens. USDT is one of the most borrowed Asset in DEFI. This may cause losing a lot of potential users.

Proof of Concept

LenderPool.sol#L327

Recommended Mitigation Steps

Use .safeTransfer instead of .transfer

```
IERC20(_borrowAsset).safeTransfer(_to, _fee);
```

ritik99 (Sublime) confirmed

saxenism (Sublime) commented:

> Shifted from transfer to safeTransfer sublime-finance/sublime-v1#372

[M-02] Lack of access control allow anyone to `withdrawInterest()` for any lender

Submitted by WatchPug

[LenderPool.sol#L442](#)

```
    function withdrawInterest(uint256 _id, address _lender) external
        _withdrawInterest(_id, _lender);
    }

    function _withdrawInterest(uint256 _id, address _lender) interna
        address _strategy = pooledCLConstants[_id].borrowAssetStrate
        address _borrowAsset = pooledCLConstants[_id].borrowAsset;

        (uint256 _interestToWithdraw, uint256 _interestSharesToWithd
            _id,
            _lender,
            _strategy,
            _borrowAsset
        );
        pooledCLVariables[_id].sharesHeld = pooledCLVariables[_id].sl

        if (_interestToWithdraw != 0) {
            SAVINGS_ACCOUNT.withdraw(_borrowAsset, _strategy, _lende
        }
        emit InterestWithdrawn(_id, _lender, _interestSharesToWithdr
    }
```

`withdrawInterest()` at a certain time may not be in the best interest of the specific `lender`.

It's unconventional and can potentially cause leak of value for the `lender`. For example, the lender may still want to accrued more interest from the strategy.

Recommended Mitigation Steps

Change to:

```
    function withdrawInterest(uint256 _id, address _lender) external
        require(msg.sender == _lender);
        _withdrawInterest(_id, _lender);
    }
```

ritik99 (Sublime) confirmed, but disagreed with Medium severity and commented:

> This is a valid suggestion. This allowed more flexibility from a composability/complexity perspective (for eg, an abstraction can be built that regularly withdraws interests for all lenders), hence the check was not put in place. We will add a check as suggested.

> Since assets are not at risk (any withdrawn amount is still transferred to the correct lender), we would suggest lowering the severity to (1) low-risk.

HardlyDifficult (judge) commented:

> Agree with medium risk as this seems to potentially leak some value for the lender.

saxenism (Sublime) commented:

> Added access control for withdrawInterest in LenderPool sublime-finance/sublime-v1#374

[M-03] Potentially depositing at unfavorable rate since anyone can deposit the entire lenderPool to a known strategy at a pre-fixed time

Submitted by kyliek

LenderPool.sol#L312
LenderPool.sol#L336

An attacker could keep track of the `totalSupply` of each LenderPool to see if it is more than the `minBorrowAmount`. If so, at `startTime`, which is pre-announced, the attacker could call `start`, which will trigger `SAVINGS_ACCOUNT.deposit()` of the

entire pool assets to mint LP tokens from external strategy, for example, in CompoundYield.

There is potentially a big sum depositing into a known Compound `cToken` contract at a known fixed time. Thus, the attacker could prepare the pool by depositing a fair sum first to lower the exchange rate before calling `start` in lenderPool. Hence, the deposit of the entire pool could be at a less favourable rate. This also applies to other potential strategies that are yet to be integrated. For example, in Curve pool, the attacker could prime the pool to be very imbalanced first and trigger the deposit and then harvest the arbitrage bonus by bringing the pool back to balance.

This attack can happen once only when the pooledCreditLine becomes active for each new lenderPool.

Proof of Concept

Step 1: When a new LenderPool started, note the borrowAsset token and its strategy target pool, as well as the collection period (i.e. start time)

Step 2: Closer to the start time block number, if `totalSupply` of the lenderPool is bigger than the `minBorrowAmount`, prepare a good sum to manipulate the target strategy pool for unfavorable exchange rate or arbitrage opportunity afterwords.

Step 3: Call `start` function before others, also put in his own address to `_to` to pocket the protocol fee.

Step 4: Depending on the strategy pool, harvest arbitrage. Or perhaps just withdraw one's money from Step 2 for griefing.

Recommended Mitigation Steps

Add access control on start, e.g. only borrower can call through pooledCreditLine.

ritik99 (Sublime) commented:

> We're not sure how this leads to an attack. Fluctuation in yield is known and expected for most yield strategies like Compound. This, however, does not cause a loss of funds. An attacker instantly withdrawing their liquidity doesn't affect others

> because interest rates are not “locked in” on the yield strategies. There are no exchanges taking place. Some more info on possible attacks might help.

HardlyDifficult (judge) commented:

> Followed up with the warden and there appears to be a way to leak value by debalancing the pool before start and then rebalancing to extract some profit. This could be done with a flashbot for example to limit exposure.

> The warden referenced https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md as an example of something similar.

ritik99 (Sublime) commented:

> Had a discussion with the warden and the judge regarding this issue. For Compound in particular we checked that the exchange rate does not change upon deposits or withdrawals. Thus, sandwiching a call to `start()` couldn’t possibly lead to an attack vector. Additionally, because of another issue related to start fees #19, we decided to restrict `start()` to be callable only by the borrower.

> However, the yield strategies that we whitelist might still be internally susceptible to this attack (for eg, https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md). We’ll be incorporating checks for this while onboarding strategies. Picking riskier strategies is optional and not enforced at the contract-level.

[M-04] Interest accrued could be zero for small decimal tokens

Submitted by hickuphh3

PooledCreditLine.sol#L1215-L1221

Interest is calculated as

```
(_principal.mul(_borrowRate).mul(_timeElapsed).div(YEAR_IN_SECONI
```

It is possible for the calculated interest to be zero for principal tokens with small decimals, such as EURS (2 decimals). Accumulated interest can therefore be zero by borrowing / repaying tiny amounts frequently.

Proof of Concept

Assuming a borrow interest rate of 5% (`5e17`) and principal borrow amount of `10_000` EURS (`10_000 * 1e2 = 1_000_000`), the interest rate calculated would be 0 if principal updates are made every minute (around 63s).

```
// in this example, maximum duration for interest to be 0 is 63s
1_000_000 * 5e17 * 63 / (86400 * 365) / 1e18 = 0.99885 // = 0
```

While plausible, this method of interest evasion isn't as economical for tokens of larger decimals like USDC and USDT (6 decimals).

Recommended Mitigation Steps

Take caution when allowing an asset to be borrowed. Alternatively, scale the principal amount to precision (1e18) amounts.

ritik99 (Sublime) disagreed with High severity and commented:

> Tokens are whitelisted to ensure precision issues would not occur. Hence the issue is improbable and doesn't occur for widely used tokens as the decimals are generally higher.

> Since there is no direct attack path (the steps required for this to occur would be: the token would first have to be whitelisted -> a loan request created using it -> lenders supply sufficient liquidity for this request to go active) and is, in essence, a value leak, we would suggest reducing the severity of the issue to (1) Low / (2) Medium.

HardlyDifficult (judge) decreased severity to Medium

KushGoyal (Sublime) commented:

> [LIME-286 c4march-10 interest accrued could be zero for small decimal tokens sublime-finance/sublime-v1#391]

Low Risk and Non-Critical Issues

For this contest, 18 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: lllllll, kyliek, WatchPug, 0x1f8b, robee, sseefried, BouSalman, MetaOxNull, rayn, 0xkatana, Dravee, hake, 0v3rf10w, 0xDjango, dirk_y, defsec, *and* gzeon.

Codebase Impressions & Summary

Overall, code quality for the PooledCreditLine contracts is great. Majority of the logic lies in the 2 contracts `PooledCreditLine` and `LenderPool`, with a small part on the `twitterVerifier` that handles verification via Twitter.

Complexity

The project is a little high in complexity because there are quite a number of possible states that a pooled credit line can have over its lifecycle, which means state handling has to be thoroughly scrutinised between transitions. The handling of interest rate calculations, borrower and lender shares accounting, and shares <> amounts conversions for integration with the saving account and strategies are other factors that raise the complexity. A lot of logic and functionality is thus packed into the 2 contracts that makes this 3 day contest feel underscoped.

Documentation

The documentation provided was adequate in understanding the pool credit line functionality. Documentation about the termination functionality and start and protocol fees were unfortunately omitted. It would be great to add them in.

It would also have been great if inline comments were added to the `_calculateInterestToWithdraw()` and `_rebalanceInterestWithdrawn()` functions as I spent quite a bit of time deciphering what these functions were doing.

Nevertheless, there were sufficient inline comments for the other parts of the contracts.

Tests

Tests were unfortunately omitted because last minute changes were made to the contract and *"the tests couldn't be modified to meet those changes in time for the contest. In order to not confuse people we decided it was best to remove the tests from the final release".* It would have been a nice to have so that coverage can be run, and for us to quickly spin up POCs. I'm not sure how feasible it would have been to postpone the contest by a few days so that tests could be modified, but it would've been beneficial.

Responsiveness

I would like to commend Ritik for his quick responses to my DMs and question on the Discord channel! =)

[01] Discrepancy between recorded borrow amount in event and state update

PooledCreditLine.sol#L910
PooledCreditLine.sol#L913
PooledCreditLine.sol#L917

A protocol fee is taken whenever the borrower decides to borrow more tokens. The state update includes this protocol fee, but the amount emitted in the `BorrowedFromPooledCreditLine` event does not.

In my opinion, since the protocol fee should be included in the emitted event.

Recommended Mitigation Steps

```
    emit BorrowedFromPooledCreditLine(_id, _borrowedAmount.add(proto
```

[02] Use upgradeable version of OZ contracts

LenderPool.sol#L7-L9
PooledCreditLine.sol#L5-L7

It is recommended to use the upgradeable version of OpenZeppelin contracts, as some contracts like ReentrancyGuard has a constructor method to set the initial status as `_NOT_ENTERED = 1`. The status would currently defaults to `0`, which fortunately doesn't break the `nonReentrant()` functionality.

Nevertheless, it would be recommended to use the upgradeable counterparts instead.

[03] `calculatePrincipalWithdrawable()` should return user balance for `CANCELLED` status

LenderPool.sol#L579-L592

In the event the borrower cancels his borrow request, the principal withdrawable by the lender should be the liquidity he provided, but the function returns 0 instead.

Recommended Mitigation Steps

Add the `CANCELLED` case in the second if branch.

```
    else if (
      (
        _status == PooledCreditLineStatus.REQUESTED &&
        block.timestamp > pooledCLConstants[_id].startTime &&
        totalSupply[_id] < pooledCLConstants[_id].minBorrowAmount
      ) || (_status == PooledCreditLineStatus.CANCELLED)
    ) {
      return balanceOf(_lender, _id);
    }
```

[04] Use `continue` instead of `return` in `_beforeTokenTransfer()`

LenderPool.sol#L686-L688

Should the contract be upgraded to use `_mintBatch()` in the future, the function will terminate prematurely after minting the first id.

Recommended Mitigation Steps

Replace `return` with `continue`.

```
    if (from == address(0)) {
      continue;
    }
```

[05] Token approval issues

- `safeApprove()` has been deprecated in favour of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`
- using `approve()` might fail because some tokens (eg. USDT) don't work when changing the allowance from an existing non-zero allowance value

Recommended Mitigation Steps

Update instances of `approve()` and `safeApprove()` to `safeIncreaseAllowance()`.

[06] Typos

Do a CTRL / CMD + F for the following errors:

`terminatd` → `terminated`

`pooleed` → `pooled`

`reqeuested` → `requested`

[07] Definition mix-up in documentation

Reference: https://docs.sublime.finance/sublime-docs/the-protocol/pooled-credit-lines#creating-a-pooled-credit-line

The definitions for the `Collateral Savings Strategy` and `Borrowed Asset Savings Strategy` have been mixed up.

Recommended Mitigation Steps

```
9. Collateral Savings Strategy: Savings strategy where any colla
10. Borrowed Asset Savings Strategy: Savings strategy where any
```

[08] Inconsistent Naming

It would be great to have variable naming kept consistent for better readibility.

- `_lendingShare`, `_liquidityProvided` to represent `balanceOf(msg.sender, _id);`
- `withdrawnShares` vs `sharesWithdrawn`

ritik99 (Sublime) commented:

> Thanks for the comments! We'll definitely be updating our documentation to make it more detailed, both the external docs as well as inline comments.

> All the issues mentioned by the warden are relevant. Usually, where approve() is used the allowance is used entirely in the subsequent transfer step, so it shouldn't be an issue, although we'll recheck all such instances. The report is of high quality.

HardlyDifficult (judge) commented:

> For scoring, also including Issue #20.

HardlyDifficult (judge) commented:

> This report is clear / easy to read. The intro is a great addition to provide some high level feedback.

- 01: Discrepancy between recorded borrow amount in event and state update
  Non-critical: This is somewhat arbitrary, but useful feedback to consider.

Depending on the use case for consumers of this event, it may be useful to emit both _borrowedAmount and protocolFee as separate params as well.

- 02: Use upgradeable version of OZ contracts
  Non-critical: This is a best practice but as the warden points out it will not break anything in the current state. Switching to ReentrancyGuardUpgradeable would save gas on first usage.
- 03: calculatePrincipalWithdrawable() should return user balance for CANCELLED status
  Low-risk: This impacts an external getter that in the original form may return misleading results after a request is canceled.
- 04: Use continue instead of return in _beforeTokenTransfer()
  Low-risk: If the `return` in this loop is executed than other tokenIds being transferred would skip the `require` checks and possibly some expected state updates. However given that the code currently does not batch mint this effectively has no impact but could crop up unexpectedly after an upgrade as the warden pointed out.
- 05: Token approval issues
  Non-issue: Several wardens pointed to this concern. The way the contract is implemented, approval always resets back to 0 after the transfer so the failure scenario would not arise. It's a good consideration though and something to be careful about to ensure that assumption holds true.
- 06: Typos
  Non-critical: Always nice to fix up the spelling errors.
- 07: Definition mix-up in documentation
  Non-critical: This is a nice catch to improve the documentation. Ramping up on a new protocol takes time and changes like this can help the reader create the right mental models.
- 08: Inconsistent Naming
  Non-critical: Naming is always hard to do well. Improving internal consistency does help the reader.

Gas Optimizations

For this contest, 19 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: Dravee, robee, hickuphh3, defsec, 0xkatana, rfa, MetaOxNull, gzeon, Funen, Tomio, 0xNazgul, kenta, 0v3rf10w, rayn, 0x1f8b, csanuragjain, 0xDjango, *and* hake.

[G-01] Multiple mappings can be combined into a single mapping of a value to a struct

1. File: contracts/PooledCreditLine/LenderPool.sol (lines 109-121)

```
/**
 * @notice Mapping that stores constants for pooledCredit cr
 */
mapping(uint256 => LenderPoolConstants) public pooledCLConst
/**
 * @notice Mapping that stores variables for pooledCredit cr
 */
mapping(uint256 => LenderPoolVariables) public pooledCLVaria
/**
 * @notice Mapping that stores total pooledCreditLine token
 * @dev Since ERC1155 tokens don't support the totalSupply f
 */
mapping(uint256 => uint256) public totalSupply;
```

2. File: contracts/PooledCreditLine/PooledCreditLine.sol (lines 184-198)

```
/**
 * @notice stores the collateral shares in a pooled credit l
 * @dev creditLineId => collateralShares
 **/
mapping(uint256 => uint256) public depositedCollateralInShar

/**
 * @notice stores the variables to maintain a pooled credit
 **/
mapping(uint256 => PooledCreditLineVariables) public pooledC

/**
 * @notice stores the constants related to a pooled credit l
 **/
mapping(uint256 => PooledCreditLineConstants) public pooledC
```

[G-02] `++i` / `i++` should be `unchecked{++i}` / `unchecked{++i}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

1. File: contracts/PooledCreditLine/LenderPool.sol (line 670)

```
            for (uint256 i; i < ids.length; ++i) {
```

[G-03] `<array>.length` should not be looked up in every loop of a `for` -loop

Even memory arrays incur the overhead of bit tests and bit shifts to calculate the array length

1. File: contracts/PooledCreditLine/LenderPool.sol (line 670)

```
            for (uint256 i; i < ids.length; ++i) {
```

[G-04] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

1. File: contracts/Verification/twitterVerifier.sol (line 88)

```
            string memory _name,
```

2. File: contracts/Verification/twitterVerifier.sol (line 89)

```
            string memory _version
```

3. File: contracts/Verification/twitterVerifier.sol (line 120)

```
        string memory _twitterId,
```

4. File: contracts/Verification/twitterVerifier.sol (line 121)

```
        string memory _tweetId,
```

[G-05] `internal` functions only called once can be inlined to save gas

1. File: contracts/PooledCreditLine/LenderPool.sol (lines 694-698)

```
    function _rebalanceInterestWithdrawn(
        uint256 id,
        uint256 amount,
        address from,
        address to
```

2. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 388)

```
    function _limitBorrowedInUSD(address _borrowToken, uint256 _l
```

3. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 671)

```
    function _createRequest(Request calldata _request) internal
```

4. File: contracts/PooledCreditLine/PooledCreditLine.sol (lines 693-701)

```
    function _notifyRequest(
        uint256 _id,
        address _lenderVerifier,
        address _borrowToken,
        address _borrowAssetStrategy,
        uint256 _borrowLimit,
        uint256 _minBorrowedAmount,
```

```
        uint256 _collectionPeriod,
        bool _areTokensTransferable
```

5. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 897)

```
    function _borrow(uint256 _id, uint256 _amount) internal {
```

6. File: contracts/PooledCreditLine/PooledCreditLine.sol (lines 955-959)

```
    function _withdrawBorrowAmount(
        address _asset,
        address _strategy,
        uint256 _amountInTokens
    ) internal returns (uint256) {
```

7. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 1019)

```
    function _repay(uint256 _id, uint256 _amount) internal retur
```

8. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 1223)

```
    function updateStateOnPrincipalChange(uint256 _id, uint256 _
```

[G-06] Multiple `if` -statements with mutually-exclusive conditions should be changed to `if` - `else` statements

If two conditions are the same, their blocks should be combined

1. File: contracts/PooledCreditLine/LenderPool.sol (lines 676-688)

```
            if (from == address(0)) {
                totalSupply[id] = totalSupply[id].add(amount);
            } else if (to == address(0)) {
                uint256 supply = totalSupply[id];
```

```
            require(supply >= amount, 'T3');
            totalSupply[id] = supply - amount;
        } else {
            require(pooledCLConstants[id].areTokensTransferal
        }

        if (from == address(0)) {
            return;
        }
```

[G-07] Use a more recent version of solidity

Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath` Use a solidity version of at least 0.8.2 to get compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

1. File: contracts/Verification/twitterVerifier.sol (line 2)

```
pragma solidity 0.7.6;
```

2. File: contracts/PooledCreditLine/LenderPool.sol (line 2)

```
pragma solidity 0.7.6;
```

3. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 2)

```
pragma solidity 0.7.6;
```

[G-08] Splitting `require()` statements that use `&&` saves gas

See this issue for an example

1. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 642)

```
require(_request.borrowAsset != address(0) && _request.c
```

[G-09] `require()` or `revert()` statements that check input arguments should be at the top of the function

1. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 394)

```
require(_minBorrowAmount <= _borrowLimit, 'ILB2');
```

2. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 778)

```
require(_amount <= _withdrawableCollateral, 'WC1');
```

3. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 900)

```
require(_amount <= calculateBorrowableAmount(_id), 'B3')
```

[G-10] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second access of a state variable within a function. Less obvious optimizations include having local storage variables of mappings within state variable mappings or mappings within state variable structs, having local storage variables of structs within mappings, or having local caches of state variable contracts/addresses.

See **original submission** for instances.

[G-11] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

1. File: contracts/Verification/twitterVerifier.sol (line 117)

```
uint8 _v,
```

2. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 164)

```
uint128 borrowLimit;
```

3. File: contracts/PooledCreditLine/PooledCreditLine.sol (line 165)

```
uint128 borrowRate;
```

[G-12] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

See original submission for instances.

ritik99 (Sublime) commented:

> All suggestions are valid and the report is highly detailed.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top