Sublime contest
Findings & Analysis Report

2022-02-14

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 code 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 code contest outlined in this document, C4 conducted an analysis of Sublime contest smart contract system written in Solidity. The code contest took place between December 9—December 15 2021.

Wardens

21 Wardens contributed reports to the Sublime contest:

1. cmichel
2. WatchPug (jtp and ming)
3. leastwood
4. 0x0x0x
5. certora
6. harleythedog
7. kemmio
8. hyh
9. 0x1f8b
10. gpersoon
11. 0xngndev
12. TomFrenchBlockchain
13. sirhashalot
14. Jujic
15. broccolirob
16. pedroais
17. robee
18. defsec
19. pmerkleplant
20. gzeon
21. p4st13r4 (0xb4bb4, 0x69e8)

This contest was judged by Oxean.

Final report assembled by tealtiger, CloudEllie, and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 43 unique vulnerabilities and 93 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 11 received a risk rating in the category of HIGH severity, 8 received a risk rating in the category of MEDIUM severity, and 24 received a risk rating in the category of LOW severity.

C4 analysis also identified 18 non-critical recommendations and 32 gas optimizations.

Scope

The code under review can be found within the C4 Sublime contest repository, and is composed of 16 smart contracts written in the Solidity programming language and includes 1,689 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.

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 (11)

[H-01] In `CreditLine#_borrowTokensToLiquidate`, oracle is used wrong way

Submitted by 0x0x0x

Current implementation to get the price is as follows:

```
(uint256 _ratioOfPrices, uint256 _decimals) =
IPriceOracle(priceOracle).getLatestPrice(_borrowAsset,
_collateralAsset);
```

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/CreditLine/CreditLine.sol#L1050

But it should not consult `borrowToken / collateralToken`, rather it should consult the inverse of this result. As a consequence, in `liquidate` the liquidator/lender can lose/gain funds as a result of this miscalculation.

Mitigation step

Replace it with

```
(uint256 _ratioOfPrices, uint256 _decimals) =
IPriceOracle(priceOracle).getLatestPrice(_collateralAsset,
_borrowAsset);
```

ritik99 (Sublime) confirmed

[H-02] Wrong returns of `SavingsAccountUtil.depositFromSavingsAccount()` can cause fund loss

Submitted by WatchPug

The function `SavingsAccountUtil.depositFromSavingsAccount()` is expected to return the number of equivalent shares for given `_asset`.

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/Pool/Pool.sol#L225-L267

```
    /**
     * @notice internal function used to get amount of collateral de
     * @param _fromSavingsAccount if true, collateral is transferrec
     * @param _toSavingsAccount if true, collateral is transferred t
     * @param _asset address of the asset to be deposited
     * @param _amount amount of tokens to be deposited in the pool
     * @param _poolSavingsStrategy address of the saving strategy us
     * @param _depositFrom address which makes the deposit
     * @param _depositTo address to which the tokens are deposited
     * @return _sharesReceived number of equivalent shares for giver
     */
    function _deposit(
        bool _fromSavingsAccount,
        bool _toSavingsAccount,
        address _asset,
        uint256 _amount,
        address _poolSavingsStrategy,
        address _depositFrom,
        address _depositTo
    ) internal returns (uint256 _sharesReceived) {
        if (_fromSavingsAccount) {
            _sharesReceived = SavingsAccountUtil.depositFromSavingsA
                ISavingsAccount(IPoolFactory(poolFactory).savingsAcc
                _depositFrom,
                _depositTo,
                _amount,
                _asset,
                _poolSavingsStrategy,
                true,
                _toSavingsAccount
            );
        } else {
            _sharesReceived = SavingsAccountUtil.directDeposit(
                ISavingsAccount(IPoolFactory(poolFactory).savingsAcc
                _depositFrom,
                _depositTo,
                _amount,
```

```
                _asset,
                _toSavingsAccount,
                _poolSavingsStrategy
            );
        }
    }
```

However, since `savingsAccountTransfer()` does not return the result of `_savingsAccount.transfer()`, but returned `_amount` instead, which means that `SavingsAccountUtil.depositFromSavingsAccount()` may not return the actual shares (when pps is not 1).

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/SavingsAccount/SavingsAccountUtil.sol#L11-L26

```
    function depositFromSavingsAccount(
        ISavingsAccount _savingsAccount,
        address _from,
        address _to,
        uint256 _amount,
        address _token,
        address _strategy,
        bool _withdrawShares,
        bool _toSavingsAccount
    ) internal returns (uint256) {
        if (_toSavingsAccount) {
            return savingsAccountTransfer(_savingsAccount, _from, _t
        } else {
            return withdrawFromSavingsAccount(_savingsAccount, _from
        }
    }
```

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/SavingsAccount/SavingsAccountUtil.sol#L66-L80

```
    function savingsAccountTransfer(
        ISavingsAccount _savingsAccount,
        address _from,
```

```
        address _to,
        uint256 _amount,
        address _token,
        address _strategy
    ) internal returns (uint256) {
        if (_from == address(this)) {
            _savingsAccount.transfer(_amount, _token, _strategy, _to
        } else {
            _savingsAccount.transferFrom(_amount, _token, _strategy,
        }
        return _amount;
    }
```

As a result, the recorded `_sharesReceived` can be wrong.

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/Pool/Pool.sol#L207-L223

```
    function _depositCollateral(
        address _depositor,
        uint256 _amount,
        bool _transferFromSavingsAccount
    ) internal nonReentrant {
        uint256 _sharesReceived = _deposit(
            _transferFromSavingsAccount,
            true,
            poolConstants.collateralAsset,
            _amount,
            poolConstants.poolSavingsStrategy,
            _depositor,
            address(this)
        );
        poolVariables.baseLiquidityShares = poolVariables.baseLiquic
        emit CollateralAdded(_depositor, _amount, _sharesReceived);
    }
```

PoC

Given:

- the price per share of yearn USDC vault is `1.2`
- Alice deposited `12,000 USDC` to `yearn` strategy, received `10,000` share tokens;
- Alice created a pool, and added all the `12,000 USDC` from the saving account as collateral; The recorded `CollateralAdded` got the wrong number: `12000` which should be `10000` ;
- Alice failed to borrow money with the pool and tries to `cancelPool()` , it fails as the recorded collateral `shares` are more than the actual collateral.

As a result, Alice has lost all the `12,000 USDC` .

If Alice managed to borrow with the pool, when the loan defaults, the liquidation will also fail, and cause fund loss to the lenders.

Recommendation

Change to:

```
    function savingsAccountTransfer(
        ISavingsAccount _savingsAccount,
        address _from,
        address _to,
        uint256 _amount,
        address _token,
        address _strategy
    ) internal returns (uint256) {
        if (_from == address(this)) {
            return _savingsAccount.transfer(_amount, _token, _strate
        } else {
            return _savingsAccount.transferFrom(_amount, _token, _st
        }
    }
```

ritik99 (Sublime) confirmed

[H-03] denial of service

Submitted by certora

https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/Pool/Pool.sol#L645 if the borrow token is address(0) (ether), and someone calls withdrawLiquidity, it calls SavingsAccountUtil.transferTokens which will transfer to msg.sender, msg.value (of withdrawLiquidity, because it's an internal function). In other words, the liquidity provided will pay to themselves and their liquidity tokens will still be burned. therefore they will never be able to get their funds back.

Recommended Mitigation Steps

the bug is in https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccountUtil.sol It is wrong to use msg.value in transferTokens because it'll be the msg.value of the calling function. therefore every transfer of ether using this function is wrong and dangerous, the solution is to remove all msg.value from this function and just transfer _amount regularly.

**ritik99 (Sublime) confirmed **

[H-04] Yearn token <> shares conversion decimal issue

Submitted by cmichel

The yearn strategy `YearnYield` converts shares to tokens by doing `pricePerFullShare * shares / 1e18`:

```
    function getTokensForShares(uint256 shares, address asset) publi
        if (shares == 0) return 0;
        // @audit should divided by vaultDecimals
        amount = IyVault(liquidityToken[asset]).getPricePerFullShare
    }
```

But Yearn's `getPricePerFullShare` seems to be in `vault.decimals()` precision, i.e., it should convert it as `pricePerFullShare * shares / (10 ** vault.decimals())`. The vault decimals are the same as the underlying token decimals

Impact

The token and shares conversions do not work correctly for underlying tokens that do not have 18 decimals. Too much or too little might be paid out leading to a loss for either the protocol or user.

Recommended Mitigation Steps

Divide by `10**vault.decimals()` instead of `1e18` in `getTokensForShares`. Apply a similar fix in `getSharesForTokens`.

ritik99 (Sublime) confirmed

[H-05] Aave's share tokens are rebasing breaking current strategy code

Submitted by cmichel, also found by WatchPug and leastwood

When depositing into Aave through the `AaveYield.lockTokens` contract strategy, one receives the `sharesReceived` amount corresponding to the diff of `aToken` balance, which is just always the deposited amount as aave is a rebasing token and `1.0 aToken = 1.0 underlying` at each deposit / withdrawal.

Note that this `sharesReceived` (the underlying deposit amount) is cached in a `balanceInShares` map in `SavingsAccount.deposit` which makes this share *static* and not dynamically rebasing anymore:

```
    function deposit(
        uint256 _amount,
        address _token,
        address _strategy,
        address _to
    ) external payable override nonReentrant returns (uint256) {
        require(_to != address(0), 'SavingsAccount::deposit receiver
        uint256 _sharesReceived = _deposit(_amount, _token, _strateg
        balanceInShares[_to][_token][_strategy] = balanceInShares[_t
        emit Deposited(_to, _sharesReceived, _token, _strategy);
        return _sharesReceived;
    }

    function getTokensForShares(uint256 shares, address asset) publi
```

```
        if (shares == 0) return 0;
        address aToken = liquidityToken(asset);

        (, , , , , , , uint256 liquidityIndex, , ) = IProtocolDataPr

        // @audit-info tries to do (user shares / total shares) * ur
        amount = IScaledBalanceToken(aToken).scaledBalanceOf(address
            IERC20(aToken).balanceOf(address(this))
        );
    }
```

However, the `getTokensForShares` function uses a rebasing total share supply of `IERC20(aToken).balanceOf(this)`.

POC

- SavingsAccount deposits 1000 DAI for user and user receives 1000 aDAI as shares. These shares are cached in `balanceInShares[user][dai][aave]`.
- Time passes, Aave accrues interest for lenders, and the initial 1000 aTokens balance has rebased to 1200 aTokens
- SavingsAccount `withdraw`s 1000 aDAI shares for user which calls `AaveYield.unlockTokens`. The user receives only 1000 DAI. The interest owed to the user is not paid out.
- Note that `getTokensForShares` also returns the wrong amount as `1200 * 1000 / 1200 = 1000`

Impact

Interest is not paid out to users. Pool collateral is measured without the interest accrued as it uses `getTokensForShares` which will lead to early liquidations and further loss.

Recommended Mitigation Steps

If the user shares are not rebasing, you cannot have the “total shares supply” (the shares in the contract) be rebasing as in `getTokensForShares`. Also withdrawing the share amount directly from Aave as in `_withdrawERC` does not withdraw the yield. A fix could be to create a *non-rebasing* wrapper LP token that is paid out to the user proportional to the current strategy TVL at time of user deposit.

ritik99 (Sublime) acknowledged:

> We've been aware of this issue for some time.. ended up including the AaveYield file in the scope by mistake! We do not plan to include the Aave strategy in our launch (we maintain a strategy registry that allows us to add/drop yield strategies), and as noted in #128, we will be utilizing wrapper contracts that mimics behaviour of non-rebasing LP tokens

0xean (judge) commented:

> going to side with the warden since they believed the contract to be in scope and it's a valid concern.

[H-06] Anyone can liquidate credit line when autoLiquidation is false without supplying borrow tokens

Submitted by harleythedog

Impact

It is intended that if a credit line has autoLiquidation as false, then only the lender can be the liquidator (see docs here: https://docs.sublime.finance/sublime-docs/smart-contracts/creditlines). However, this is not correctly implemented, and anyone can liquidate a position that has autoLiquidation set to false.

Even worse, when autoLiquidation is set to false, the liquidator does not have to supply the initial amount of borrow tokens (determined by `_borrowTokensToLiquidate`) that normally have to be transferred when autoLiquidation is true. This means that the liquidator will be sent all of the collateral that is supposed to be sent to the lender, so this represents a huge loss to the lender. Since the lender will lose all of the collateral that they are owed, this is a high severity issue.

Proof of Concept

The current implementation of liquidate is here: https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/CreditLine/CreditLine.sol#L996.

Notice that the autoLiquidation value is only used in one place within this function, which is in this segment of the code:

```
...
    if (creditLineConstants[_id].autoLiquidation && _lender != m
        uint256 _borrowTokens = _borrowTokensToLiquidate(_borrow
        if (_borrowAsset == address(0)) {
            uint256 _returnETH = msg.value.sub(_borrowTokens, 'I
            if (_returnETH != 0) {
                (bool success, ) = msg.sender.call{value: _retur
                require(success, 'Transfer fail');
            }
        } else {
        IERC20(_borrowAsset).safeTransferFrom(msg.sender, _lende
        }
    }

    _transferCollateral(_id, _collateralAsset, _totalCollateralT
    emit  CreditLineLiquidated(_id, msg.sender);
}
```

So, if `autoLiquidation` is false, the code inside of the if statement will simply not be executed, and there are no further checks that the sender HAS to be the lender if `autoLiquidation` is false. This means that anyone can liquidate a non-autoLiquidation credit line, and receive all of the collateral without first transferring the necessary borrow tokens.

For a further proof of concept, consider the test file here: https://github.com/code-423n4/2021-12-sublime/blob/main/test/CreditLines/2.spec.ts. If the code on line 238 is changed from `let _autoLiquidation: boolean = true;` to `let _autoLiquidation: boolean = false;` , all the test cases will still pass. This confirms the issue, as the final test case “Liquidate credit line” has the `admin` as the liquidator, which should not work in non-autoLiquidations since they are not the lender.

Tools Used

Inspection and confirmed with Hardhat.

Recommended Mitigation Steps

Add the following require statement somewhere in the `liquidate` function:

```
    require(
        creditLineConstants[_id].autoLiquidation ||
        msg.sender == creditLineConstants[_id].lender,
        "not autoLiquidation and not lender");
```

ritik99 (Sublime) labeled sponsor confirmed

[H-07] SavingsAccount withdrawAll and switchStrategy can freeze user funds by ignoring possible strategy liquidity issues

Submitted by hyh, also found by cmichel

Impact

Full withdrawal and moving funds between strategies can lead to wrong accounting if the corresponding market has tight liquidity, which can be the case at least for `AaveYield`. That is, as the whole amount is required to be moved at once from Aave, both `withdrawAll` and `switchStrategy` will incorrectly account for partial withdrawal as if it was full whenever the corresponding underlying yield pool had liquidity issues.

`withdrawAll` will delete user entry, locking the user funds in the strategy: user will get partial withdrawal and have the corresponding accounting entry removed, while the remaining actual funds will be frozen within the system.

`switchStrategy` will subtract full number of shares for the `_amount` requested from the old strategy, while adding lesser partial number of shares for `_tokensReceived` to the new one with the same effect of freezing user's funds within the system.

Proof of Concept

SavingsAccount.withdrawAll https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L286

SavingsAccount.switchStrategy: https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L152

When full withdrawal or strategy switch is performed it is one withdraw via `unlockTokens` without checking the amount received.

In the same time the withdraw can fail for example for the strategy switch if old strategy is having liquidity issues at the moment, i.e. Aave market is currently have utilization rate too high to withdraw the amount requested given current size of the lending pool.

Aave `unlockTokens` return is correctly not matched with amount requested: https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/yield/AaveYield.sol#L217

But, for example, `withdrawAll` ignores the fact that some funds can remain in the strategy and deletes the use entry after one withdraw attempt: https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L294 https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L312

`switchStrategy` removes the old entry completely: https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L181

Recommended Mitigation Steps

For both `withdrawAll` and `switchStrategy` the immediate fix is to account for tokens received in both cases, which are `_amount` after `unlockTokens` for `withdrawAll` and `_tokensReceived` for `switchStrategy`.

More general handling of the liquidity issues ideally to be addressed architecturally, given the potential issues with liquidity availability any strategy withdrawals can be done as follows:

1. Withdraw what is possible on demand, leave the amount due as is, i.e. do not commit to completing the action in one go and notify the user the action was partial (return actual amount)
2. Save to query and repeat for the remainder funds on the next similar action (this can be separate flag triggered mode)

ritik99 (Sublime) disagreed with severity:

> The above issue requires making a few assumptions - (i) the underlying yield protocol does not have sufficient reserves to facilitate the withdrawal of a single user, (ii) the user attempts to withdraw all their assets during such times of insufficient reserves.

> We agree that the above could be a possibility, but would be unlikely. The underlying yield protocols undergo an interest rate spike during high utilization ratios to bring reserves back to normal levels, and some revert if they cannot withdraw the necessary amount (for eg, **Compound**). During live deployment, only those strategies that work expectedly would be onboarded, while others wouldn't (for eg, Aave as a strategy wouldn't be integrated until their **wrappers for aTokens** are ready for use). Hence we suggest reducing severity to (2) medium-risk

> also similar to #144

Oxean (judge) commented:

> While I understand the argument regarding this being an unlikely scenario, I don't believe that is a sufficient reason to downgrade the issue give the impact to a user and the lost funds.

> ```
> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.
> ```

> In this scenario - Assets are at a direct risk.

[H-08] Possibility to drain SavingsAccount contract assets

Submitted by kemmio

Impact

A malicious actor can manipulate switchStrategy() function in a way to withdraw tokens that are locked in SavingsAccount contract (the risk severity should be reviewed)

Proof of Concept

Firstly an attacker need to deploy a rogue strategy contract implementing IYield.getSharesForTokens() and IYield.unlockTokens() functions and calling switchStrategy() with _currentStrategy = ROGUE*CONTRACT*ADDRESS (_newStrategy can be any valid strategy e.g. NoYield)

https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L160

```
require(_amount != 0, 'SavingsAccount::switchStrategy Amount mus
```

Bypass this check by setting _amount > 0, since it will be overwritten in line https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L162

```
_amount = IYield(_currentStrategy).getSharesForTokens(_amount, _
```

getSharesForTokens() should be implemented to always return 0, hence to bypass the overflow in lines https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L164-L167

```
balanceInShares[msg.sender][_token][_currentStrategy] = balance1
_amount,
'SavingsAccount::switchStrategy Insufficient balance'
);
```

since balanceInShares[msg.sender][_token][_currentStrategy] == 0 and 0-0 will not overflow

The actual amount to be locked is saved in line https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L169

```
uint256 _tokensReceived = IYield(_currentStrategy).unlockTokens(
```

the rouge unlockTokens() can check asset balance of the contract and return the full amount

After that some adjustment are made to set approval for the token or to handle native assets case https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L171-L177

```
uint256 _ethValue;
if (_token != address(0)) {
    IERC20(_token).safeApprove(_newStrategy, _tokensReceived);
} else {
    _ethValue = _tokensReceived;
}
_amount = _tokensReceived;
```

Finally the assets are locked in the locked strategy and shares are allocated on attackers acount https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L179-L181

```
uint256 _sharesReceived = IYield(_newStrategy).lockTokens{value:

balanceInShares[msg.sender][_token][_newStrategy] = balanceInSha
```

Proof of Concept

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
contract Attacker{
    function getSharesForTokens(uint256 amount, address token) e
        return 0;
    }
    function unlockTokens(address token, uint256 amount) externa
        uint256 bal;
        if(token == address(0))
            bal = msg.sender.balance;
        else
            bal = IERC20(token).balanceOf(msg.sender);
        return bal;
    }
}
```

Recommended Mitigation Steps

Add a check for _currentStrategy to be from strategy list like the one in line https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/SavingsAccount/SavingsAccount.sol#L159

```
require(IStrategyRegistry(strategyRegistry).registry(_newStrateg
```

ritik99 (Sublime) disputed:

> The savings account contract doesn't hold any tokens, so it is not possible to lock tokens in a new strategy, hence this attack will not work. Nevertheless it is something we will explore further to limit unexpected state changes

Oxean (judge) commented:

> Based on the review of the warden I believe this is a valid attack path. This line would need to change to the amount of tokens that are to be "stolen" but otherwise this does seem accurate.

```
bal = IERC20(token).balanceOf(msg.sender);
```

[H-09] `PriceOracle` Does Not Filter Price Feed Outliers

Submitted by leastwood

Impact

If for whatever reason the Chainlink oracle returns a malformed price due to oracle manipulation or a malfunctioned price, the result will be passed onto users, causing unintended consequences as a result.

In the same time it's possible to construct mitigation mechanics for such cases, so user economics be affected by sustainable price movements only. As price outrages provide a substantial attack surface for the project it's worth adding some complexity to the implementation.

Proof of Concept

https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/PriceOracle.sol#L149-L161

```
    function getLatestPrice(address num, address den) external view
        uint256 _price;
        uint256 _decimals;
        (_price, _decimals) = getChainlinkLatestPrice(num, den);
        if (_decimals != 0) {
            return (_price, _decimals);
        }
        (_price, _decimals) = getUniswapLatestPrice(num, den);
        if (_decimals != 0) {
            return (_price, _decimals);
        }
        revert("PriceOracle::getLatestPrice - Price Feed doesn't exi
    }
```

The above code outlines how prices are utilised regardless of their actual value (assuming it is always a non-zero value).

Recommended Mitigation Steps

Consider querying both the Chainlink oracle and Uniswap pool for latest prices, ensuring that these two values are within some upper/lower bounds of each other. It may also be useful to track historic values and ensure that there are no sharp

changes in price. However, the first option provides a level of simplicity as UniswapV3's TWAP implementation is incredibly resistant to flash loan attacks. Hence, the main issue to address is a malfunctioning Chainlink oracle.

ritik99 (Sublime) disputed:

> The described suggestion is fairly complex - besides the increase in code complexity, we'd also have to decide the bounds within which the Uniswap and Chainlink oracles should report prices that won't be trivial. We've also noted in the assumptions section of our contest repo that oracles are assumed to be accurate

0xean (judge) commented:

> " We expect these feeds to be fairly reliable." - Based on this quote, I am going to leave this open at the current risk level. These are valid changes that could significantly reduce the risk of the implementation and unintended liquidations.

> Fairly reliable != 100% reliable

[H-10] Wrong implementation of `NoYield.sol#emergencyWithdraw()`

Submitted by WatchPug, also found by 0x1f8b

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/yield/NoYield.sol#L78-L83

```
    function emergencyWithdraw(address _asset, address payable _wall
        require(_wallet != address(0), 'cant burn');
        uint256 amount = IERC20(_asset).balanceOf(address(this));
        IERC20(_asset).safeTransfer(_wallet, received);
        received = amount;
    }
```

`received` is not being assigned prior to L81, therefore, at L81, `received` is `0`.

As a result, the `emergencyWithdraw()` does not work, in essence.

Recommendation

Change to:

```
function emergencyWithdraw(address _asset, address payable _wall
    require(_wallet != address(0), 'cant burn');
    received = IERC20(_asset).balanceOf(address(this));
    IERC20(_asset).safeTransfer(_wallet, received);
}
```

ritik99 (Sublime) confirmed

Oxean (judge) commented:

> upgrading to High sev based on assets being “lost” directly. IE the emergency function will not work.

> 3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).

[H-11] Unable To Call `emergencyWithdraw` ETH in `NoYield` Contract

Submitted by leastwood

Impact

The `emergencyWithdraw` function is implemented in all yield sources to allow the `onlyOwner` role to drain the contract's balance in case of emergency. The contract considers ETH as a zero address asset. However, there is a call made on `_asset` which will revert if it is the zero address. As a result, ETH tokens can never be withdrawn from the `NoYield` contract in the event of an emergency.

Proof of Concept

Consider the case where `_asset == address(0)`. An external call is made to check the contract's token balance for the target `_asset`. However, this call will

revert as `_asset` is the zero address. As a result, the `onlyOwner` role will never be able to withdraw ETH tokens during an emergency.

```
function emergencyWithdraw(address _asset, address payable _wall
    require(_wallet != address(0), 'cant burn');
    uint256 amount = IERC20(_asset).balanceOf(address(this));
    IERC20(_asset).safeTransfer(_wallet, received);
    received = amount;
}
```

Affected function as per below: https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/yield/NoYield.sol#L78-L83

Recommended Mitigation Steps

Consider handling the case where `_asset` is the zero address, i.e. the asset to be withdrawn under emergency is the ETH token.

ritik99 (Sublime) confirmed Oxean (judge) commented:

> Upgrading to Sev 3 in line with #4 / #115 as this results in funds being stuck in the contract.

Medium Risk Findings (8)

[M-01] Ether can be locked in the `PoolFactory` contract without a way to retrieve it

Submitted by broccolirob

If a borrower calls the `createPool` function with a non-zero value, but also includes an ERC20 token address for `_collateralToken`, then the Ether value sent will be locked in the `PoolFactory` contract forever.

- createPool L260-317

In the `_createPool` function, a `_collateralToken` address other than the zero address will set the `amount` variable to zero. That `amount` variable will be passed to `create2` which will send 0 wei to the newly created `Pool` contract.

```
349
350 uint256 amount = _collateralToken == address(0) ? _collatera
```

Impact

A borrower can accidentally lock Ether in the `PoolFactory` without the ability to retrieve it.

Proof of Concept

A borrower reuses a script they made to create a pool and deposit collateral. They intend to deposit Ether as collateral so they send value with the transaction, but forget to change the `_collateralToken` address to address(0). The `Pool` contract will be deployed using the `_collateralToken`, and will lock the Ether sent in the `PoolFactory`

Tools Used

Manual analysis and Hardhat.

Recommended Mitigation Steps

If msg.value is greater than 0, make sure the `_collateralToken` address is set to address(0).

ritik99 (Sublime) disputed and disagreed with severity:

> We will add this check but the scenario laid out is more about sanity checks on the side of the end-user. Assets are not stolen or compromised directly but because of user error. Such cases are better handled via UI/UX. We would suggest a (1) Low rating given the likelihood

Oxean (judge) commented:

> Marking down to medium risk based on the c4 documentation and some external requirements on how this would have to occur.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-02] CreditLine.liquidate doesn't transfer borrowed ETH to a lender

Submitted by hyh, also found by 0x0x0x

Impact

Funds that are acquired from a liquidator and should be sent to a lender are left with the contract instead. The funds aren't lost, but after the fact mitigation will require manual accounting and fund transfer for each CreditLine.liquidate usage.

Proof of Concept

ETH sent to CreditLine.liquidate by an external liquidator when `autoLiquidation` is enabled remain with the contract and aren't transferred to the lender:
https://github.com/code-423n4/2021-12-sublime/blob/main/contracts/CreditLine/CreditLine.sol#L1015

Recommended Mitigation Steps

Add transfer to a lender for ETH case:

Now:

```
    if (_borrowAsset == address(0)) {
            uint256 _returnETH = msg.value.sub(_borrowTokens, 'Insuf
            if (_returnETH != 0) {
                    (bool success, ) = msg.sender.call{value: _retur
                    require(success, 'Transfer fail');
            }
    }
```

To be:

```
    if (_borrowAsset == address(0)) {
            uint256 _returnETH = msg.value.sub(_borrowTokens, 'Insuf

            (bool success, ) = _lender.call{value: _borrowTokens}('
            require(success, 'liquidate: Transfer failed');

            if (_returnETH != 0) {
                    (success, ) = msg.sender.call{value: _returnETH}
                    require(success, 'liquidate: Return transfer fai
            }
    }
```

ritik99 (Sublime) confirmed

[M-03] Collateral can be deposited in a finished pool

Submitted by pedroais

Proof of Concept

The depositCollateral function doesn't check the status of the pool so collateral can be deposited in a finished loan. This can happen by mistake and all funds will be lost.

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/Pool/Pool.sol#L207

Recommended Mitigation Steps

Require loan status to be collection or active in the depositCollateral function.

ritik99 (Sublime) disagreed with severity:

> We will add a check for this. The issue however stems from user error. Sending assets to an address without proper checks does not constitute an attack path imo. We would suggest a rating of (1) Low or (0) non-critical given the low likelihood and the impact of the attack (only the user making the incorrect transaction is affected)

Oxean (judge) commented:

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> This definitely qualifies as “external requirements” and a simple check would assist in avoid it.

[M-04] Unlinked address can link immediately again

Submitted by gpersoon

Impact

After a master calls unlinkAddress() to unlink an address, the address that has just been unlinked can directly link again without permission. The address that is just unlinked can call linkAddress(masterAddress) which will execute because pendingLinkAddresses is still set. Assuming the master has unlinked for a good reason it is unwanted to be able to be linked again without any permission from the master.

Note: a master can prevent this by calling cancelAddressLinkingRequest(), but this doesn’t seem logical to do

Proof of Concept

https://github.com/code-423n4/2021-12-sublime/blob/e688bd6cd3df7fefa3be092529b4e2d013219625/contracts/Verification/Verification.sol#L129-L154

```
        function unlinkAddress(address _linkedAddress) external {
            address _linkedTo = linkedAddresses[_linkedAddress].mast
            require(_linkedTo != address(0), 'V:UA-Address not linke
            require(_linkedTo == msg.sender, 'V:UA-Not linked to ser
            delete linkedAddresses[_linkedAddress];
           ...
    }
```

```
        function linkAddress(address _masterAddress) external {
            require(linkedAddresses[msg.sender].masterAddress == adc
            require(pendingLinkAddresses[msg.sender][_masterAddress]
            _linkAddress(msg.sender, _masterAddress);
        }

    function cancelAddressLinkingRequest(address _linkedAddress) ext
            ...
            delete pendingLinkAddresses[_linkedAddress][msg.sender];
```

Recommended Mitigation Steps

Add something like to following at the end of linkAddress:

```
    delete pendingLinkAddresses[msg.sender][_masterAddress];
```

ritik99 (Sublime) confirmed

[M-05] Extension voting threshold check needs to rerun on each transfer

Submitted by cmichel

The `Extension` contract correctly reduces votes from the `from` address of a transfer and adds it to the `to` address of the transfer (in case both of them voted on it before), but it does not rerun the voting logic in `voteOnExtension` that actually grants the extension. This leads to issues where an extension should be granted but is not:

POC

- `to` address has 100 tokens and votes for the extension
- `from` address has 100 tokens but does not vote for the extension and transfers the 100 tokens to `to`
- `to` now has 200 tokens, `removeVotes` is run, the `totalExtensionSupport` is increased by 100 to 200. In theory, the threshold is reached and the vote should pass if `to` could call `voteOnExtension` again.

- But their call to `voteOnExtension` with the new balance will fail as they already voted on it (`lastVotedExtension == _extensionVoteEndTime`). The extension is not granted.

Impact

Extensions that should be granted after a token transfer are not granted.

Recommended Mitigation Steps

Rerun the threshold logic in `removeVotes` as it has the potential to increase the total support if `to` voted for the extension but `from` did not.

ritik99 (Sublime) confirmed

[M-06] `NoYield.sol` Tokens with fee on transfer are not supported

Submitted by WatchPug

There are ERC20 tokens that charge fee for every `transfer()` or `transferFrom()` .

In the current implementation, `NoYield.sol#lockTokens()` assumes that the received amount is the same as the transfer amount, and uses it to calculate `sharesReceived` amounts.

As a result, in `unlockTokens()` , later users may not be able to successfully withdraw their tokens, as it may revert at L141 for insufficient balance.

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/yield/NoYield.sol#L93-L106

```
    function lockTokens(
        address user,
        address asset,
        uint256 amount
```

```
) external payable override onlySavingsAccount nonReentrant retu
    require(amount != 0, 'Invest: amount');
    if (asset != address(0)) {
        IERC20(asset).safeTransferFrom(user, address(this), amou
    } else {
        require(msg.value == amount, 'Invest: ETH amount');
    }
    sharesReceived = amount;
    emit LockedTokens(user, asset, sharesReceived);
}
```

https://github.com/code-423n4/2021-12-sublime/blob/9df1b7c4247f8631647c7627a8da9bdc16db8b11/contracts/yield/NoYield.sol#L134-L144

```
function _unlockTokens(address asset, uint256 amount) internal r
    require(amount != 0, 'Invest: amount');
    received = amount;
    if (asset == address(0)) {
        (bool success, ) = savingsAccount.call{value: received}(
        require(success, 'Transfer failed');
    } else {
        IERC20(asset).safeTransfer(savingsAccount, received);
    }
    emit UnlockedTokens(asset, received);
}
```

Recommendation

Consider comparing before and after balance to get the actual transferred amount.

ritik99 (Sublime) acknowledged and disagreed with severity

[M-07] AaveYield: Misspelled external function name making functions fail

Submitted by 0xngndev

Impact

In `AaveYield.sol` the functions:

- `liquidityToken`
- `_withdrawETH`
- `_depositETH`

Make a conditional call to `IWETHGateway(wethGateway).getAWETHAddress()`

This function does not exist in the `wethGateway` contract, causing these function to fail with the error `"Fallback not allowed"`.

The function they should be calling is `getWethAddress()` without the "A".

Small yet dangerous typo.

Mitigation Steps

Simply modify:

```
IWETHGateway(wethGateway).getAWETHAddress()
```

to:

```
IWETHGateway(wethGateway).getWETHAddress()
```

In the functions mentioned above.

ritik99 (Sublime) confirmed:

> We were using an older version of the contracts that had this definition, will be updated accordingly

[M-08] Missing approve(0)

Submitted by sirhashalot, also found by Jujic, and sirhashalot

Impact

There are 3 instances where the `IERC20.approve()` function is called only once without setting the allowance to zero. Some tokens, like USDT, require first reducing the address' allowance to zero by calling `approve(_spender, 0)`. Transactions will revert when using an unsupported token like USDT (see the `approve()` function requirement at line 199).

Proof of Concept

- CreditLine/CreditLine.sol:647
- CreditLine/CreditLine.sol:779
- yield/AaveYield.sol:324

Note: the usage of `approve()` in yield/CompoundYield.sol (lines 211-212), in yield/YearnYield.sol (lines 211-212), and in yield/AaveYield.sol (lines 297-298) do not need modification since it they already use the recommended approach. Additionally the usage of `approve()` in yield/AaveYield.sol:307 likely does not need modification since that approve function only handles ETH.

Recommended Mitigation Steps

Use `approve(_spender, 0)` to set the allowance to zero immediately before each of the existing `approve()` calls.

ritik99 (Sublime) confirmed Oxean (judge) commented:

> moving to medium risk as the availability of the protocol is affected.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Low Risk Findings (24)

- **[L-01] Wrong usage of `OracleLibrary.getQuoteAtTick()` breaks `PriceOracle.sol`** *Submitted by WatchPug*

- [L-02] Deprecated safeApprove() function *Submitted by sirhashalot, also found by robee, WatchPug, and 0x1f8b*
- [L-03] `initializer` functions can be front run *Submitted by WatchPug, also found by cmichel, robee, and leastwood*
- [L-04] CreditLine.liquidate allows for price manipulated liquidation *Submitted by hyh*
- [L-06] PoolFactory and CreditLine updateSavingsAccount will break the system in production as savings account hold current user records *Submitted by hyh*
- [L-07] Extension voting power can be flashloaned *Submitted by cmichel*
- [L-08] Pool direct savingsaccount deposits fail when no strategy set *Submitted by cmichel*
- [L-09] Self-transfer leads to wrong withdrawable repayments *Submitted by cmichel, also found by gpersoon*
- [L-10] Collateral deposit does not support fee-on-transfer tokens *Submitted by cmichel*
- [L-11] calculateInterest() comments missing input parameter *Submitted by sirhashalot*
- [L-12] `delete` doesn't delete mapping in struct *Submitted by sirhashalot*
- [L-13] Two Steps Verification before Transferring Ownership *Submitted by robee*
- [L-14] Overflow in _repay() *Submitted by gpersoon*
- [L-15] transferTokens should use _from instead of msg.sender *Submitted by gpersoon*
- [L-16] Missing timelock for critical contract setters of privileged roles (Price Oracles) *Submitted by defsec*
- [L-17] `unlockShares` wrong comment *Submitted by cmichel*
- [L-18] `Pool.sol` should use the Upgradeable variant of OpenZeppelin Contracts *Submitted by WatchPug*
- [L-19] CreditLine.borrow accepts ETH transfers *Submitted by TomFrenchBlockchain*

- [L-20] Contracts allow sending ETH on calls which does not expect it *Submitted by TomFrenchBlockchain, also found by pmerkleplant*
- [L-21] `poolSizeLimit` does not account for differing unit values between borrow assets *Submitted by TomFrenchBlockchain*
- [L-22] Unnecessary receive() *Submitted by Jujic*
- [L-23] Improper Validation Of Chainlink's `latestRoundData` Function *Submitted by leastwood, also found by Jujic*
- [L-24] Natspec not matching function's logic *Submitted by 0xngndev*
- [L-05] Lack Of Precision *Submitted by robee*

Non-Critical Findings (18)

- [N-01] Named return issue *Submitted by robee*
- [N-02] Event missing when removing a vote in extensions *Submitted by pedroais*
- [N-03] Repayments._transferTokens doesn't check msg.value in ETH case *Submitted by hyh*
- [N-04] approve return values not checked *Submitted by cmichel*
- [N-05] missing nonreentrant modfier *Submitted by certora*
- [N-06] `SavingsAccount.sol` Wrong `amount` in `Transfer` events *Submitted by WatchPug*
- [N-07] `getInterestOverdue` reverts rather than returning 0 when there is no overdue interest *Submitted by TomFrenchBlockchain*
- [N-08] No validation of protocol fee fraction *Submitted by sirhashalot, also found by cmichel*
- [N-09] Not verified function inputs of public / external functions *Submitted by robee, also found by 0x1f8b, and WatchPug*
- [N-10] CreditLine.sol assumes 365 day year *Submitted by sirhashalot*
- [N-11] Comments inconsistency for _id *Submitted by sirhashalot*
- [N-12] Magic number 30 could be a constant *Submitted by sirhashalot*
- [N-13] Change state mutability in NoYield.sol *Submitted by p4st13r4*

- [N-14] Best Practice: Contract file name should follow coding conventions *Submitted by WatchPug*
- [N-15] `idealCollateralRatio` is confusingly named *Submitted by TomFrenchBlockchain*
- [N-16] Argument order for SavingsAccount approval functions is odd *Submitted by TomFrenchBlockchain*
- [N-17] Duplicated code in Yield contracts *Submitted by TomFrenchBlockchain*
- [N-18] Typo in liquidateCancelPenalty natspec *Submitted by TomFrenchBlockchain*

Gas Optimizations (32)

- [G-01] Gas: Upgrading solc version and removing SafeMath *Submitted by 0xngndev, also found by WatchPug, 0x0x0x, Jujic, and defsec*
- [G-02] Fix Unused Variables and Function Parameters *Submitted by 0xngndev, also found by p4st13r4*
- [G-03] Unnecessary uint zero initialization *Submitted by sirhashalot, also found by 0x0x0x, WatchPug, cmichel, pmerkleplant, and robee*
- [G-04] Prefix increments are cheaper than postfix increments *Submitted by robee, also found by 0x0x0x and WatchPug*
- [G-05] Unnecessary Reentrancy Guards *Submitted by robee*
- [G-06] Remove unused local variables *Submitted by pmerkleplant*
- [G-07] Gas Optimization: Struct layout *Submitted by gzeon*
- [G-08] Use of _msgSender() *Submitted by defsec*
- [G-09] Gas: Use `else if` in `withdrawLiquidity` *Submitted by cmichel*
- [G-10] `AaveYield.getTokensForShares()`, `AaveYield.getSharesForTokens()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-11] `Pool.sol#withdrawBorrowedAmount()` Validation of pool status can be done earlier to save gas *Submitted by WatchPug*
- [G-12] `10**30` can be changed to `1e30` and save some gas *Submitted by WatchPug*

- [G-13] Inline unnecessary function can make the code simpler and save some gas *Submitted by WatchPug*
- [G-14] Cache storage variables in the stack can save gas *Submitted by WatchPug*
- [G-15] Remove unnecessary variables can make the code simpler and save some gas *Submitted by WatchPug*
- [G-16] LinkedAddress struct can be packed to save an SSTORE *Submitted by TomFrenchBlockchain*
- [G-17] Unnecessary zero approvals in yield contracts *Submitted by TomFrenchBlockchain*
- [G-18] Check on `poolConstants.loanWithdrawalDeadline` for liquidation is unnecessary *Submitted by TomFrenchBlockchain*
- [G-19] Flattening nested mappings can save gas *Submitted by TomFrenchBlockchain*
- [G-20] Use one require instead of several *Submitted by Jujic*
- [G-21] Redundant use safeMath *Submitted by Jujic, also found by WatchPug*
- [G-22] Reduce length of require error messages to save in deployment costs *Submitted by 0xngndev, also found by Jujic, WatchPug, robee, and sirhashalot*
- [G-23] Gas: Inlining logic that's used only once in the contract *Submitted by 0xngndev*
- [G-24] Gas saving using delete *Submitted by 0x1f8b*
- [G-25] Gas saving removing safe math *Submitted by 0x1f8b*
- [G-26] Gas saving by duplicate check *Submitted by 0x1f8b*
- [G-27] Gas optimization *Submitted by 0x1f8b*
- [G-28] Gas saving by struct reorganization *Submitted by 0x1f8b, also found by TomFrenchBlockchain*
- [G-29] Not needed zero address check *Submitted by 0x0x0x*
- [G-30] Loops can be implemented more efficiently *Submitted by 0x0x0x, also found by WatchPug, pmerkleplant, and robee*
- [G-31] Unnecessary array boundaries check when loading an array element twice *Submitted by robee, also found by pmerkleplant*

- [G-32] Credit Line acceptance logic can be simplified to avoid SLOAD in some cases *Submitted by TomFrenchBlockchain, also found by WatchPug*

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