SMART CONTRACTS REVIEW

November 22th 2023 | v. 1.0

Security Audit Score

PASS

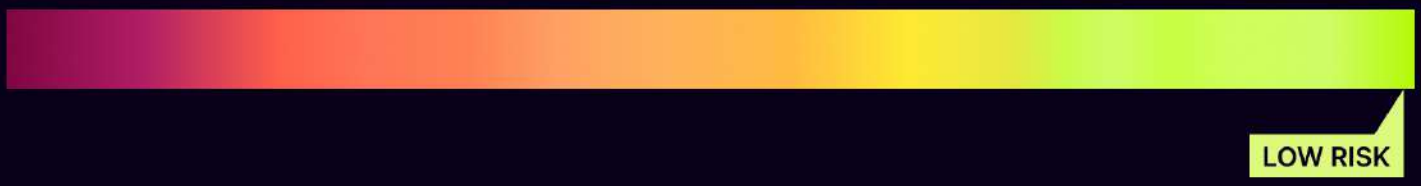
Zokyo Security has concluded that these smart contracts passed a security audit.

TECHNICAL SUMMARY

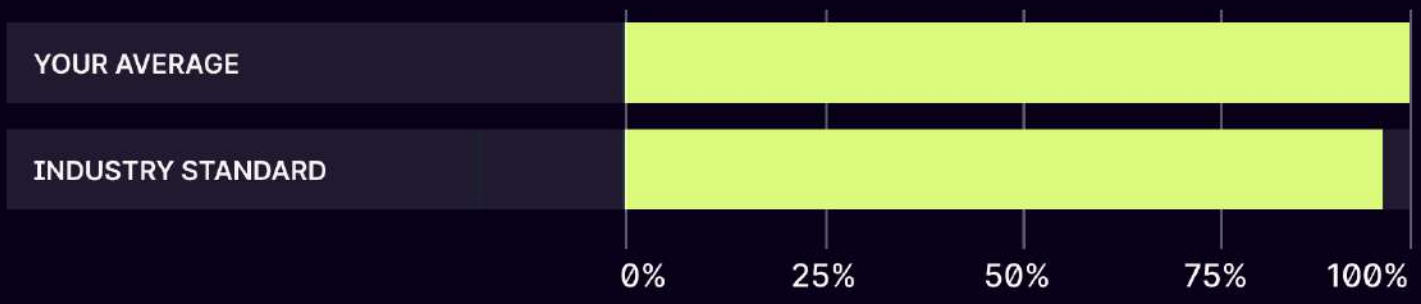
This document outlines the overall security of the Defactor Pools smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Defactor Pools smart contracts codebase for quality, security, and correctness.

Contract Status



Testable Code



100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Defactor team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Defactor Pools repository:
Repo: https://github.com/defactor-com/pools-blockchain/

Initial commit. master branch: fa088da841420889d4c56425b8de7d863150df17
Final commit, master branch: df875640566a7321439697e0ce0d0662a9255489

Within the scope of this audit, the auditors reviewed the following contract(s):

- Aurus
- AurusStorage

Since the initial commit, the contracts have been renamed by the Defactor team as follows:

- ERC20CollateralPool
- ERC20CollateralPoolStorage

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Defactor Pools smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract by industry leaders.	04	Thorough manual review of the codebase line by line.

ZOKYO AUDIT SCORING DEFACTOR

1. Severity of Issues:
   - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
   - High: Important issues that can compromise the contract in certain scenarios.
   - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
   - Low: Smaller issues that might not pose security risks but are still noteworthy.
   - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- Critical issues: 2 issue (resolved): 0 points deducted
- High issues: 2 issue (resolved): 0 points deducted. While the issue is verified, it still raises concerns from auditor's side regarding the absence of the reliable oracle
- Medium issues: 2 issue (resolved): 0 points deducted
- Low issues: 5 issue (resoled): 0 points deducted
- Informational issues: 18 issues (16 resolved, 2 verified): -2 points deducted due to that (as still even verified issues create a concerns regarding the existing solution), to the failed check against the backdoor and including the lack of documentation.

Thus, 100 - 2 = 98

Executive Summary

The Zokyo Security team has conducted an audit of the Defactor Pools protocol. This protocol is a decentralized lending platform that allows users to participate in lending, borrowing, repayment, pool liquidation, and claiming rewards for staking the lending token. The lending token is assumed to be USDC, and the collateral tokens are SILVER and GOLD.

This audit aimed to scrutinize the protocol's security, validate the safety of the funds stored on smart contracts, and verify the accuracy of the logic used in calculations and token distribution. Also, the audit involved thoroughly examining the smart contracts, checking them against a list of common vulnerabilities, the auditors' internal checklist, and gas optimization.

The audit revealed various issues, categorized as follows: 2 critical, 2 high, 2 medium, and several low and informational. The critical issues were primarily associated with the miscalculation of remaining interest when the pool was liquidated and the potential for underflow if a single borrow was initiated but not repaid before the pool's closure. The Defactor developers have successfully fixed these issues by implementing accurate calculations for the remaining interest.

One of the high issues was connected to an inaccurate calculation of the collateral value during the pool liquidation, which could have led to the liquidator not receiving the collateral funds. An incorrect value validation was also detected as a high issue, which could allow the creation of pools with a past end time only. The Defactor team has addressed these issues to ensure the protocol's robustness.

The first of the two medium-priority issues was a lack of validation during the initialization process. The second medium-priority issue involved the necessity to liquidate the pool multiple times in some scenarios when trying to claim rewards. Low and information issues, on their part, were connected to non-explicitly return logic paths, unused code, hard-code, deprecated functionality, namings, suggestions for gas optimization, and other minor issues. For a comprehensive list of all identified issues, please refer to the 'Complete Analysis' section.

The protocol's overall security is high enough. The contract is well-written and tested. Auditors have thoroughly audited all the contracts and functions of the protocol, both

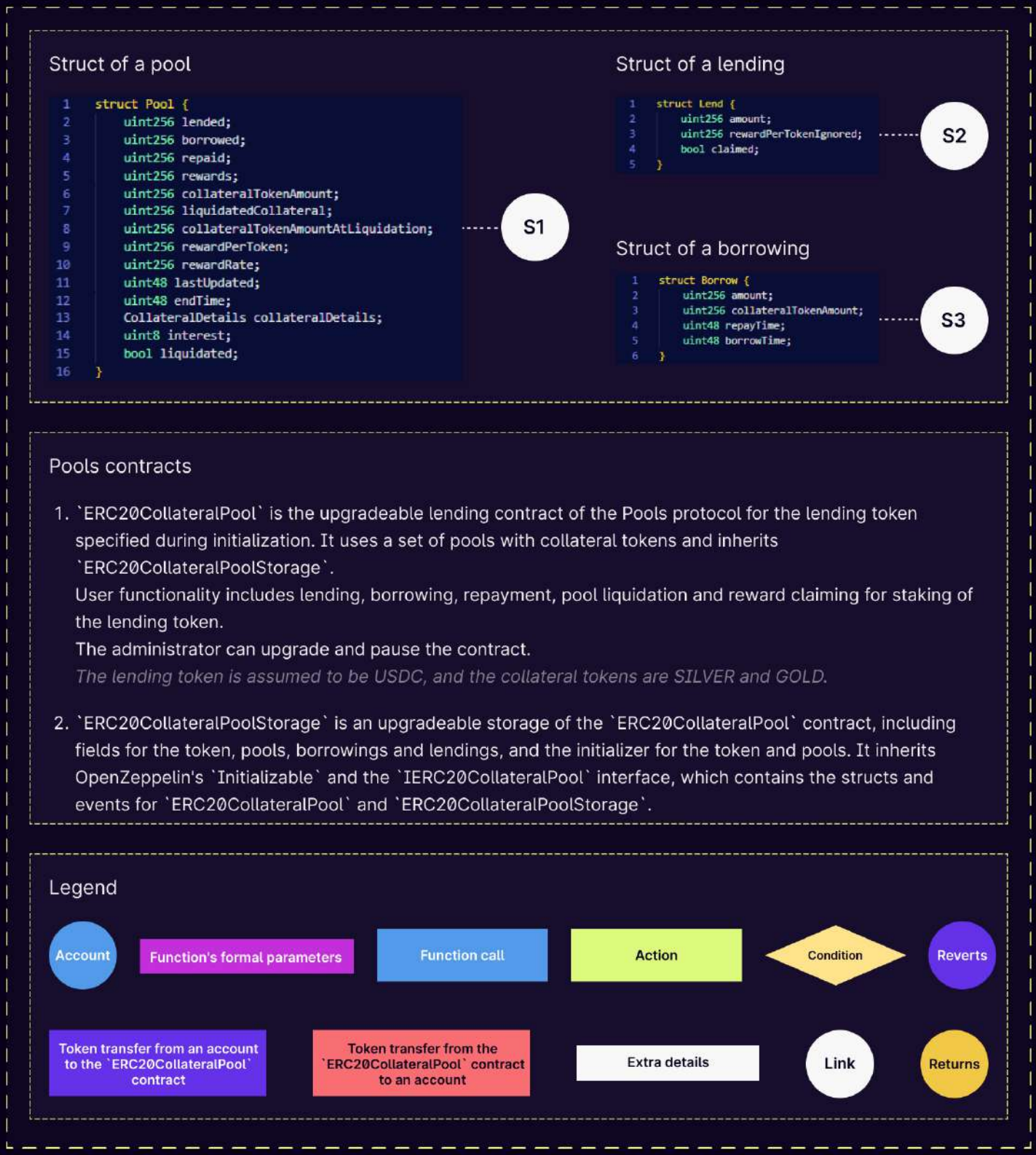
manually and with the aid of unit and scenario tests. Tests written by the Defactor team were also scrutinized, and the auditing team added additional value validations. All core functionalities were carefully audited, including pool initialization, lending, borrowing, repaying, claiming rewards, claiming unliquidated collateral, liquidation, and all related calculations. However, the auditors still have concerns regarding the pool liquidation logic. We recommend implementing specific monitoring for the protocol to ensure quick reactions to stressful events.

Also, during the last audit iteration, the Defactor team provided several updates to the protocol:

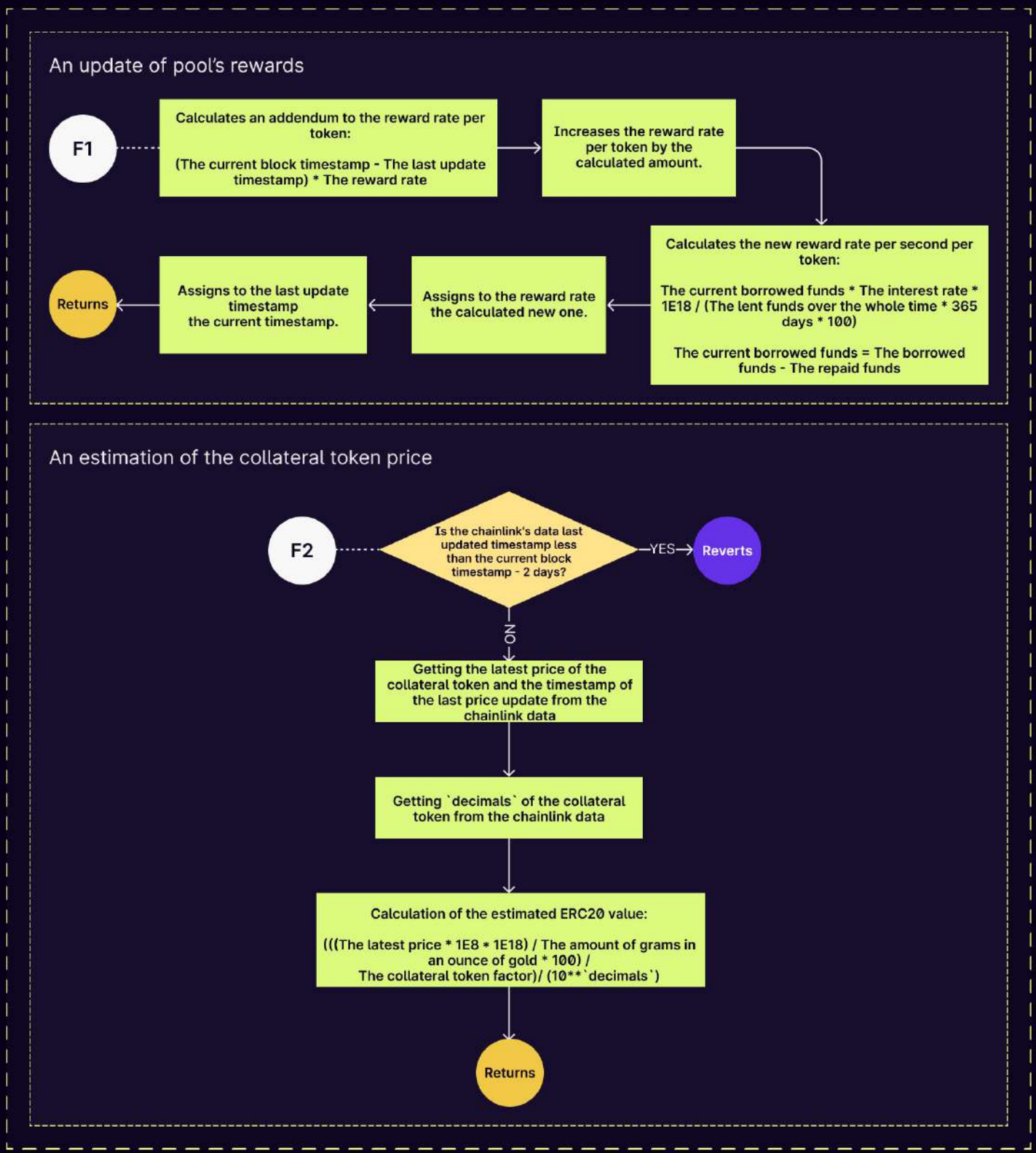
- they renamed the contracts;
- they updated the pool liquidation functionality. Now, claiming rewards and liquidation are calling separately, eliminating the need to liquidate the pool multiple times;
- they updated the calculation of the collateral token price and removed hard-coded addresses and prices by integrating the Chainlink Aggregator v3;
- they updated the calculation of the liquidatable amount by incorporating the liquidation fee and the liquidation protocol fee;
- they added a pause functionality for the core functions of the contract.

The auditing team reviewed these updates and their impact on the smart contract's functioning. An additional set of tests confirmed that the issues were resolved correctly. The final assessment reflects that all the issues have been addressed or verified. However, it should be noted that the design allows anyone to complete the liquidation of the entire pool. Additionally, the contract did not pass the check against backdoors, as the contracts are upgradeable. Despite being a common approach, it still creates a controllable backdoor, which is noteworthy for a contract handling funds.

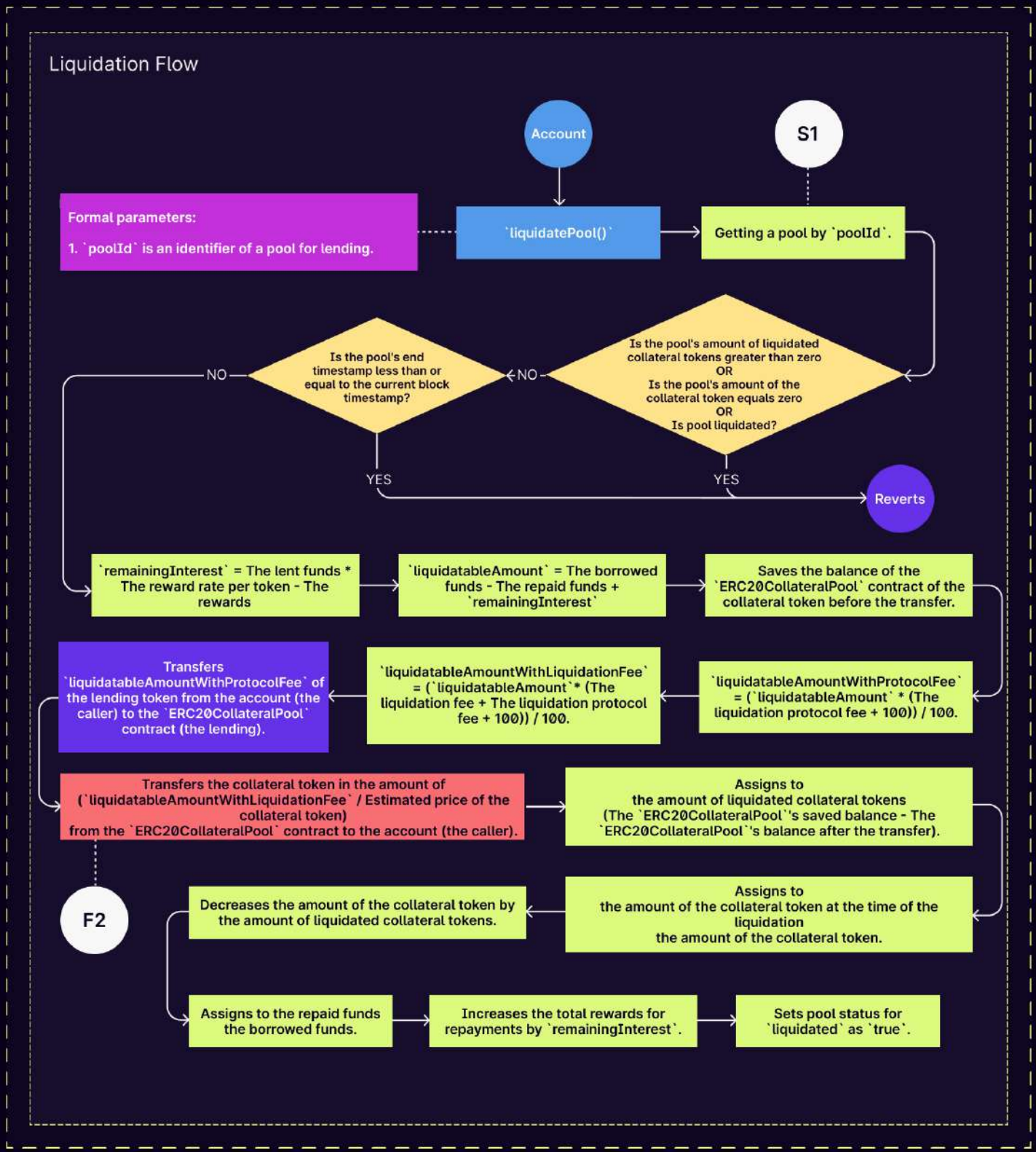
Defactor (Pools) project diagram



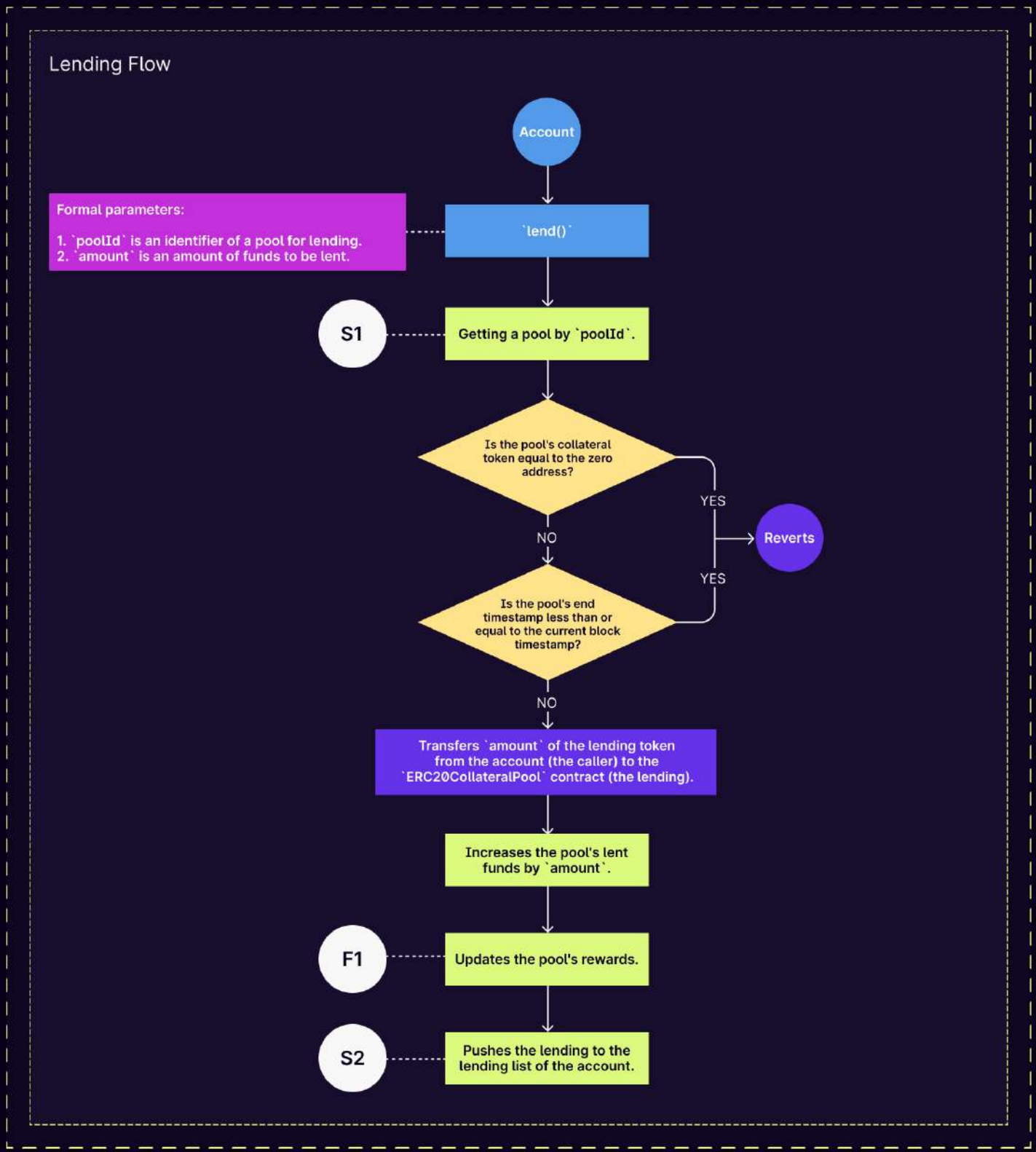
Defactor (Pools) project diagram



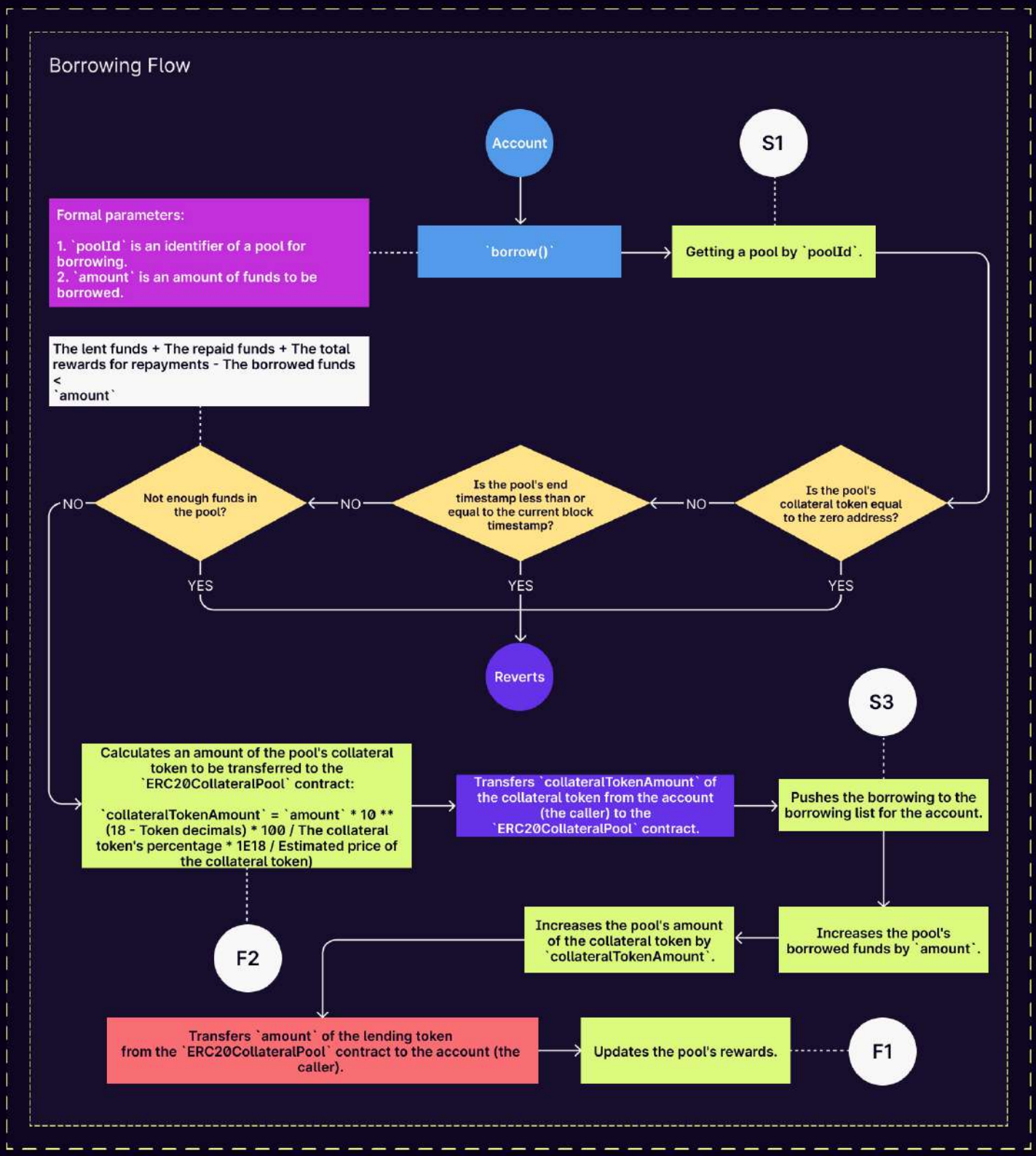
Defactor (Pools) project diagram



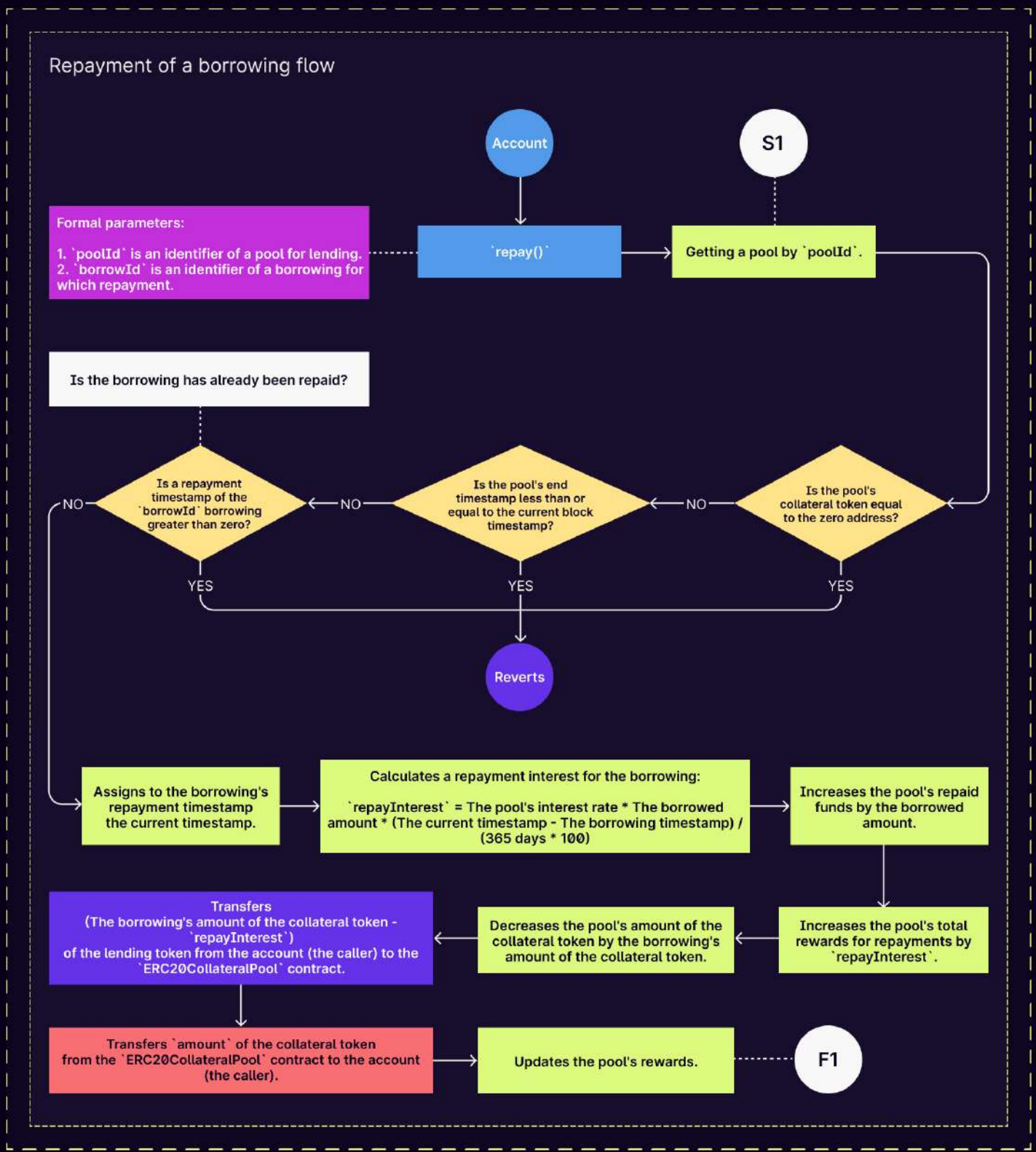
Defactor (Pools) project diagram



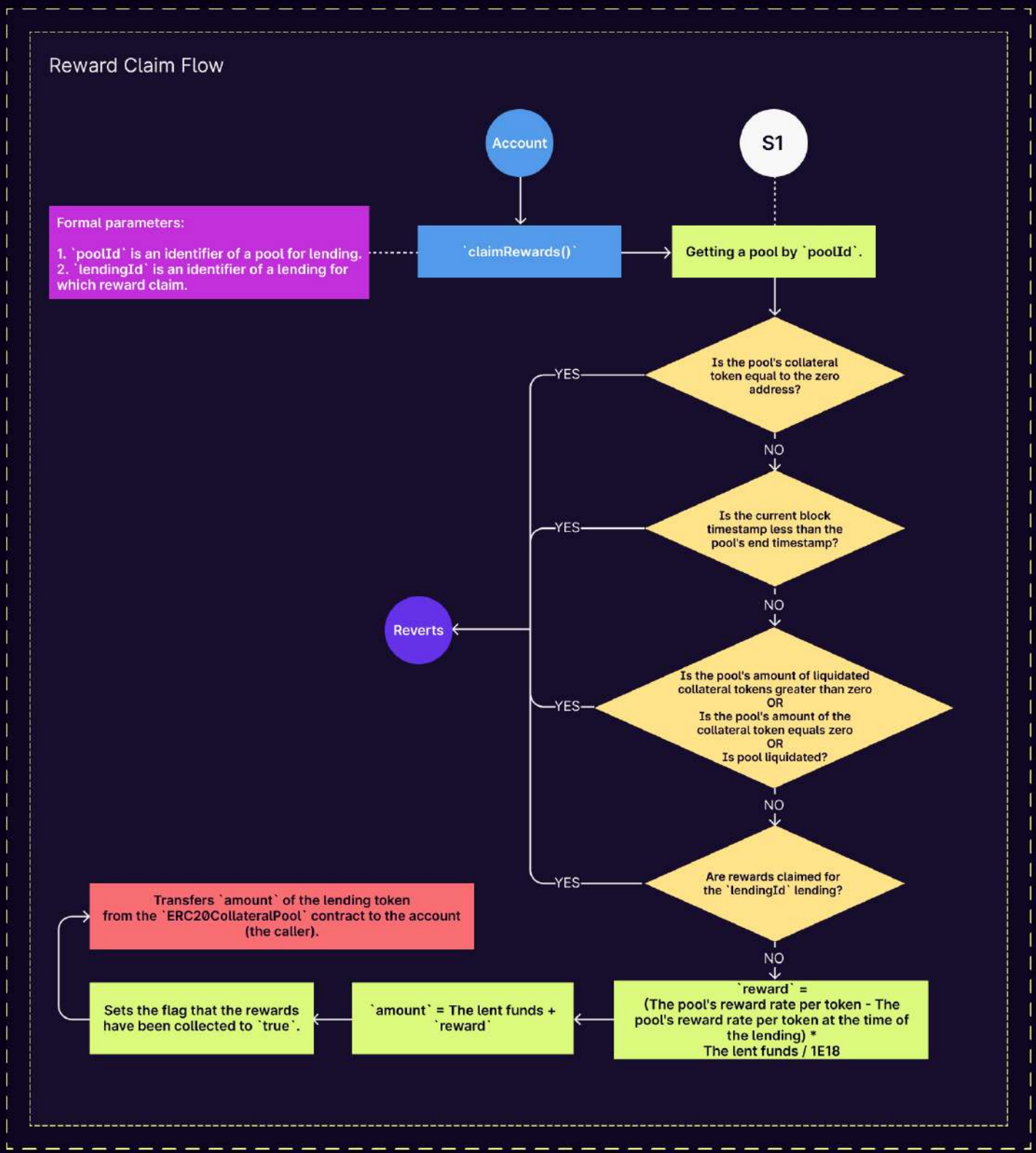
Defactor (Pools) project diagram



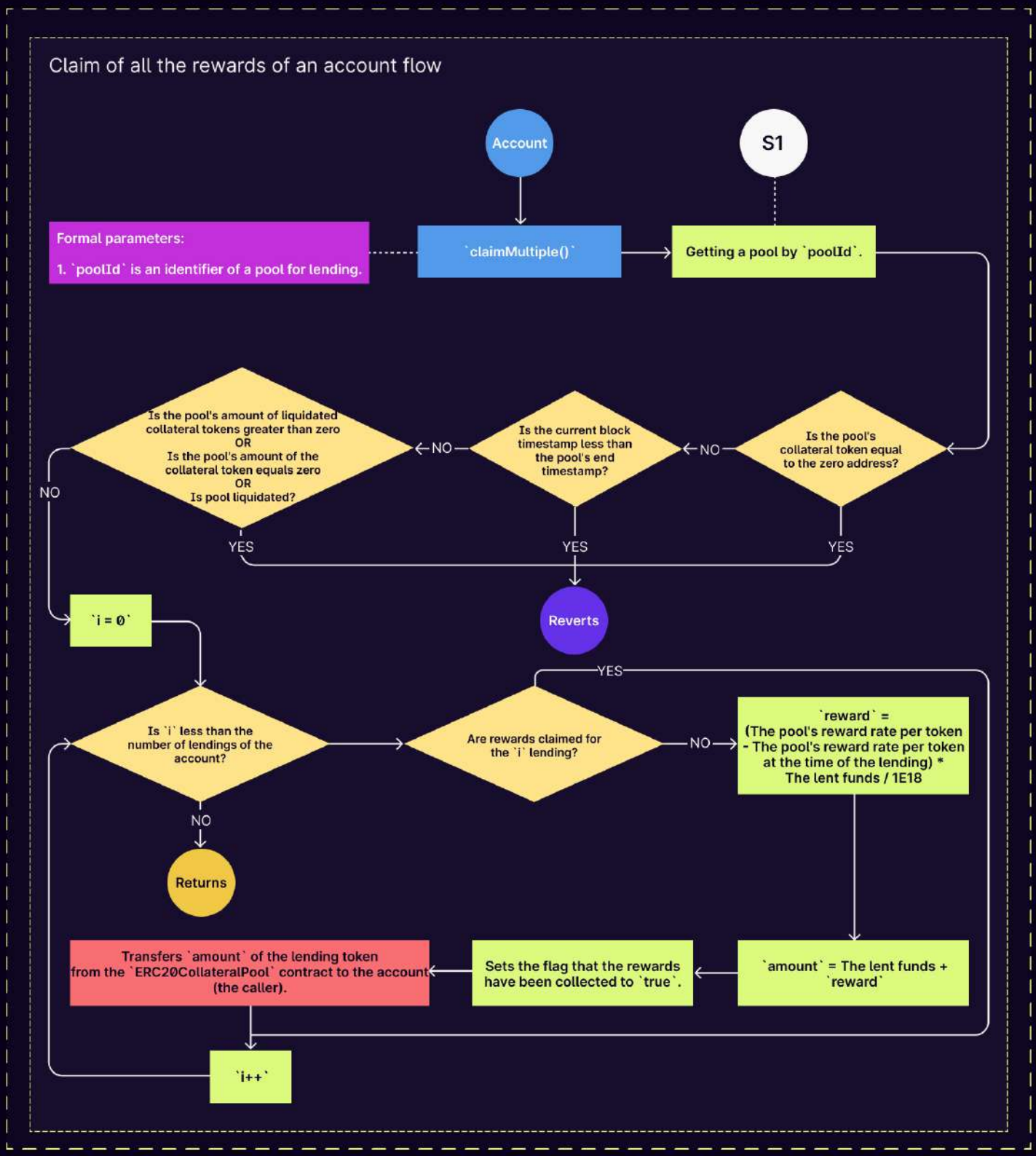
Defactor (Pools) project diagram



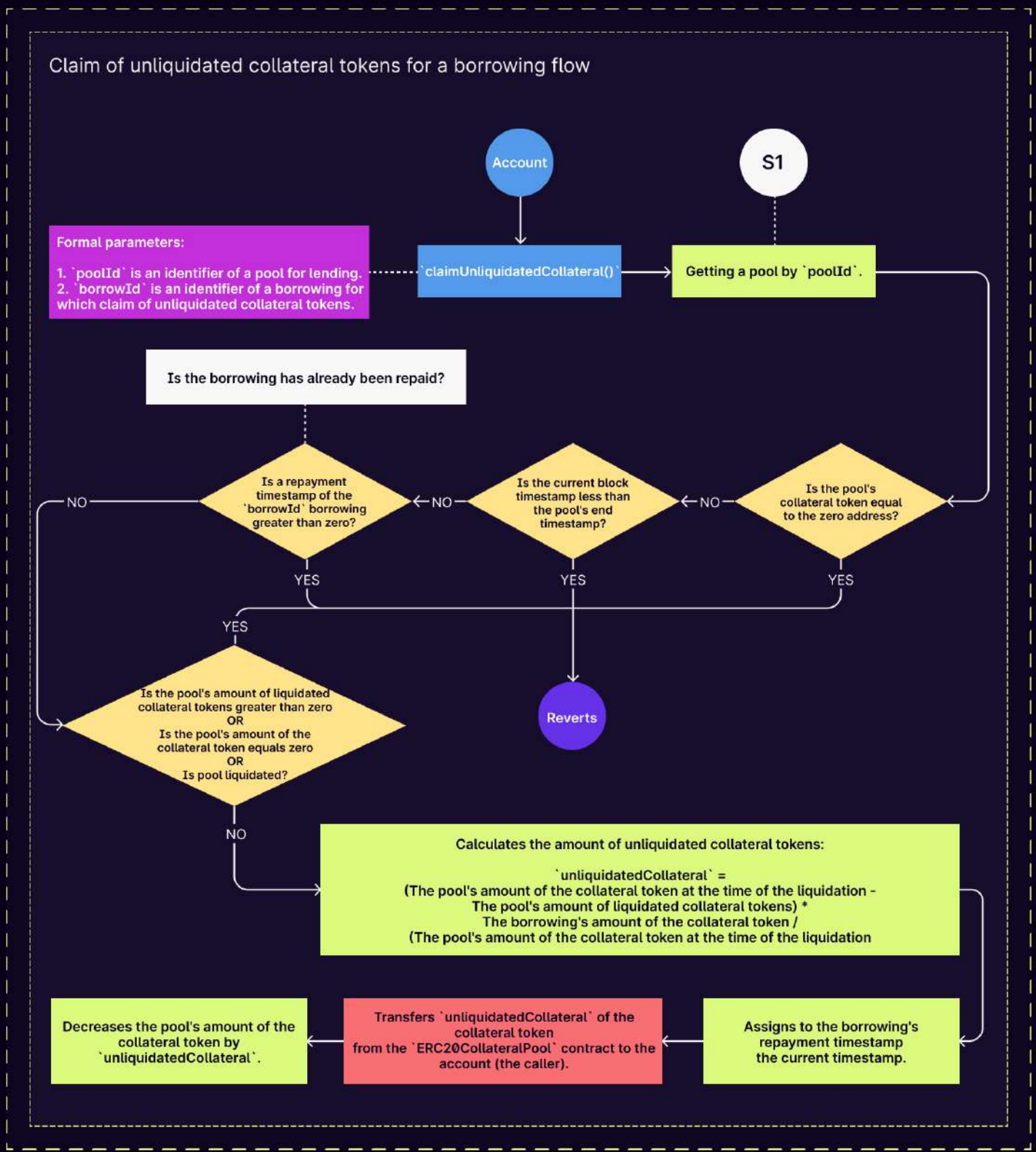
Defactor (Pools) project diagram



Defactor (Pools) project diagram



Defactor (Pools) project diagram



STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as "Resolved" or "Unresolved" depending on whether they have been fixed or addressed. The issues that are tagged as "Verified" contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.

Low

The issue has minimal impact on the contract's ability to operate.

Informational

The issue has no impact on the contract's ability to operate.

COMPLETE ANALYSIS

CRITICAL-1 | RESOLVED

Incorrect calculation of remaining interest.

Aurus.sol: _liquidatePool(), line 308.
The following formula is used for calculating the remaining interest:
*pool.lended * pool.rewardPerToken - pool.interest*.
Where:
pool.lended – total amount of USDC lended to the pool.
pool.rewardPerToken – total reward per lended token.
pool.interest - an interest coefficient to calculate how many tokens should the borrowers repay.
Based on the test cases provided by the Defactor team, the interest is typically set at 10, 15, or 17. Therefore, subtracting it from the multiplication result doesn't have a significant impact. As a result, the calculated variable *remainingInterest* will encompass the entire interest, including the portion that borrowers have repaid. This situation leads to the function requesting more tokens from the liquidator than should.

Recommendation:
Subtract the amount of already repaid interest, not the coefficient. Based on the contract's logic, the repaid interest is held in the variable *'pool.rewards'*.

Post-audit:
The subtracted value now is already repaid interest - 'pool.rewards' variable, so the remaining interest is correctly calculated.

Underflow possible when no borrows were repaid.

Aurus.sol: _updateRewards(), line 254, 255.
_liquidatePool(), line 308.
In the scenario where only one borrow was committed, and no borrow was repaid until the pool is closed, an underflow will occur in line 308 if a lender tries to claim rewards. This is because of the value of the *pool.rewardPerToken* will be 'zero' in this case. The underflow happens due to calculations in the *_updateRewards()* function. The *_calculateRewardPerToken()* function, which depends on the *pool.rewardRate()* value, is called before this value is assigned. This sequence of events leads to the issue.

Recommendation:

This issue will be fixed once **Critical-1** is resolved.

Post-audit:
The remaining interest is calculated correctly since **Critical-1** is fixed; thus, there is no underflow possibility now.

Inaccurate calculation of collateral value.

Aurus.sol: _liquidatePool(), line 320.
The following formula *liquidatableAmount / _estimateErc20Value(pool)* calculates the amount of collateral from USDC. Since *liquidatableAmount* is in USDC with 6 decimals and the result of *_estimateErc20Value(pool)* is in collateral with 18 decimals, such division can potentially return 0 or an inexact value. As a result, the liquidator won't receive enough collateral funds.

Recommendation:
Calculate the amount of collateral by considering the accuracy of arithmetical operations in Solidity. For example, there are already implemented operations in the function *calculateCollateralTokenAmount():*

1. Firstly, the amount of USDC is normalized: *normalizedUSDCAmount = _normalize(amount, USDC.decimals()).*
2. Secondly, the normalized USDC value is additionally multiplied by 1e18 to ensure the correct number of decimals in the result: *(1e18 * collateralTokenUSDCValue) / _estimateErc20Value(pool).*

Implement a similar calculation in _liquidatePool() when calculating the value of collateral from USDC.

Post-audit:
The function `*_estimateErc20Value()*` was redesigned and now returns not the fixed price of the collateral token but the correctly estimated and normalized price of the collateral token using Chainlink data, so its usage is correct now.

Incorrect validation of the value that allows to create only pools with end time in the past.

ERC20CollateralPool.storage.sol, line 25.
Validation of the pool's end time is incorrect. Instead of preventing the creation of pools with an end time in the past, this check incorrectly prevents the creation of pools with an end time in the future. The following validation:
`if (_pools[i].endTime > block.timestamp)
revert PoolEndsAtIsInThePast();`
should be replaced by:
`if (_pools[i].endTime < block.timestamp)
revert PoolEndsAtIsInThePast();`

Recommendation:
Replace validation so it would prevent the creation of pools with an end time in the past.

Post-audit:
The pool's end time is now correctly checked, initialization of the pool requires its end time to be in the future and not in the past.

MEDIUM-1 | RESOLVED

Lack of validation during initialization.

There are a number of validations missing during contract initialization.

- Aurus.storage.sol: __AurusStorage_init().

The formal parameter `_usdc` is not checked for equality to the zero address. Also, the token's decimals should be validated for compliance with the requirements of the implemented logic. For example, equality of six or a specific range.
The fields of the `_pools`'s elements should be validated according to the logic. Among other things, this includes checking that `endTime` is greater than the current block timestamp, validating `collateralToken`, and validating ranges for the percentage and interest rate.

- Aurus.sol: __Aurus_init():

There is no verification that the `_admin` address is not equal to the zero address.
Validating parameters according to logic is one of the core parts of ensuring safe use.

Recommendation:
Add validation of formal parameters according to the required logic.

MEDIUM-2 | RESOLVED

Token loss risk due to the necessity to call the liquidation function multiple times.

Aurus.sol: _liquidatePool(), line 324.
Due to the inaccurate calculation described in **High-1**, there is a severe risk that the final value of the *pool.liquidatedCollateral* in the result of the following calculations *pool.liquidatedCollateral = balanceBefore - balanceAfter* will be 'zero' since the balance didn't change. Because of it, the validation in line 305 will be unable to catch work properly, and users, while trying to claim their rewards or unliquidated collateral, will be forced to pay extra *liquidatableAmount*.

Recommendation:
After fixing **High-1**, implement additional validation to prevent users from losing extra tokens.

LOW-1 | RESOLVED

Return variable can remain unassigned.

Aurus.sol: _estimateErc20Value().
This function does not explicitly return a value to all non-reverting code paths so that an unnamed return variable can remain unassigned. The current implementation leads to potentially incorrect behavior on the part of the logic, causing reverting with a panic error.

Recommendation:
Add a reversal with a meaningful error for unacceptable values, **OR** add an explicit return with value to all logic paths.

Post-audit: The method's functionality was revised; for now, all the values for non-reverting code paths are returnable.

LOW-2 | RESOLVED

Dead pause functionality.

Aurus.sol.
The contract inherits OpenZeppelin's module `PausableUpgradeable`, but its functionality is not implemented, and its functions (and modifiers) are not used. Thus, there is no emergency stop mechanism to freeze all token transfers in case of a large bug or another unexpected behavior.

Recommendation:
Correctly implement the pause mechanism, including `_pause()` and `_unpause()` functions with access for an authorized account.

LOW-3 | RESOLVED

Unused access control mechanism functionality.

Aurus.sol.
The contract inherits and initializes the OpenZeppelin's module `AccessControlUpgradeable`, but the role `DEFAULT_ADMIN_ROLE` is not used anywhere after initialization and other roles are not implemented. Unused code may indicate incompleteness of the code base, which is critical for production.

Recommendation:
Implement role-based access control mechanism **OR** remove the module.

LOW-4 | RESOLVED

Unused library.

Aurus.sol.
The library `OracleLibrary` is imported, but its functionality is not used anywhere. Unused code may indicate incomplete functionality, which is critical for production.

Recommendation:
Complete the implementation of the functional component **OR** remove the library.

LOW-5 | RESOLVED

Unused constants.

Aurus.sol: UNISWAP_ROUTER, UNISWAP_QUOTER.
These constants are declared, but not used anywhere in the logic. Unused code may indicate that the contract logic is incomplete, which is critical for production.

Recommendation:
Complete the functionality **OR** remove the unused constants.

INFORMATIONAL-1 | RESOLVED

Hard-coded ETH price for collateral tokens.

Aurus.sol: _estimateErc20Value().
Constant price values in Ether for collateral assets are specified on lines 297 and 301. This approach is potentially dangerous because it cannot adapt to changes associated with asset price fluctuations.

Recommendation:
Use verified oracles or protocols, such as Chainlink's price feeds, to determine asset prices where possible, **OR** create variables for asset prices and setter functions, updating prices from reliable sources in a timely manner with verified solutions.

INFORMATIONAL-2 | RESOLVED

Hard-coded addresses from a testing network in production code.
Aurus.sol: _estimateErc20Value().

Although the hardcoded addresses on lines 296 and 300 belong to the Goerli testnet, they are used in production code. This may indicate that the code base is incomplete. It is worth noting that if address constants are used, it is worth commenting on which network an address belongs to.

Recommendation:
Instead of hard-coded addresses of the test network, create variables for storing the corresponding addresses and setter functions, then set the addresses of the main network.

INFORMATIONAL-3 | RESOLVED

Inaccurate pragma statement.

Aurus.sol, Aurus.storage.sol, Aurus.interface.sol.
The complex pragma statement `pragma solidity ^0.8.17;` is used for the contracts. A contract should be deployed with the same compiler version and options with which it has been most tested. Using a strict pragma with a locked version helps ensure that a contract is not accidentally deployed using a different version. In addition, older versions of Solidity may contain bugs and vulnerabilities and be less optimized regarding gas consumption. Using the latest version of Solidity and specifying the exact pragma is recommended.

Recommendation:
Use the strict pragma statement with the latest version of Solidity (or with the verified version with which a contract has been most tested).

INFORMATIONAL-4 | RESOLVED

Usage of a deprecated functionality.

Aurus.sol: __Aurus_init().
According to the OpenZeppelin documentation, the function `_setupRole()` is deprecated, but it is used when the contract initializes. Usage of deprecated methods is strictly not recommended. It is worth mentioning that the new version of OpenZeppelin's `AccessControl` does not have it at all.

Recommendation:
Use `_grantRole()` instead.

INFORMATIONAL-5 RESOLVED

Arrays can be replaced with mappings.

Aurus.sol, Aurus.storage.sol: pools, borrows, lendings.
These arrays are only used to add new elements to the end of the sequence and read them. There is also no sequence processing specific to arrays and no popping of sequence elements. In such cases, it is better to use mapping to reduce gas consumption.

Recommendation:
Replace these arrays with mappings.

INFORMATIONAL-6 RESOLVED

Lack of documentation.

Aurus.sol, Aurus.storage.sol, Aurus.interface.sol.
In the contracts missing NatSpec documentation for a lot of their externally callable functions, variables with public visibility (that have externally callable getters), events, errors, structs and themselves.
Documentation helps to:

- make the code clearer and more readable, giving a fuller view;
- ensure code quality when written and modified;
- keep developers in sync and minimize unnecessary interaction;
- make it easier to integrate with other protocols.

It is therefore recommended to fully describe their action, calling requirements, features, formal parameters, return values and event emissions, applying NatSpec where required.
It is worth noting that the Solidity documentation recommends contracts to be fully annotated using NatSpec for all public interfaces (everything in the ABI).

Recommendation:
Add the full NatSpec documentation.

Private functions that are used only once.

Aurus.sol: _calculateRewardPerToken(), _calculateRewardRate(), _normalize().
These private methods are used only once in the logic. In such cases, it is recommended to perform a calculation directly at the point of use. This allows to reduce the gas consumption a bit and also increases the readability of the code due to consistency.
It is worth noting that a descriptive comment may be left at the point where the calculations are performed to provide more clarity.

Recommendation:
Perform these calculations directly at the point of use instead of using one-off functions.

Unclear liquidation flow.

Aurus.sol: _liquidatePool().
Function _liquidatePool() allows the first user to call it to liquidate any remaining collateral on the pool. Thus, the function is executed only once during the lifetime of the pool. The function is internal and is called within the functions:

- claimRewards().
- claimMultiple().
- claimUnliquidatedCollateral().

Thus, none of these functions can be executed until it is executed by a user who has approved enough USDC and is willing to liquidate the pool. There are two main concerns here:

1. The user was unaware that he gave enough approval and didn't intend to liquidate the pool since there is no separate external mechanism to execute _liquidatePool(), and the user might just have intended to claim his rewards.
2. The pool is blocked, and no one can claim their rewards until a user is willing to liquidate the whole pool. The partial liquidation is not allowed.

Recommendation:

1. Provide more detailed documentation regarding the liquidation flow. Verify that it works as intended, where a single liquidator should liquidate the whole pool in one operation.
2. Consider providing an external function specifically for liquidation to allow users who are willing to liquidate a pool without claiming any rewards.
3. Consider making an ability to liquidate the pool partially. This will help to ensure several users with moderate amount of funds can liquidate instead of waiting for a single user to liquidate.

Post-audit. Added an external function for liquidation that allows users to liquidate the pool without the need to claim rewards in the same function call.
The Defactor team verified that anyone can liquidate the whole pool; this is by design. However, the auditor's team highly recommends adding specific monitoring for such liquidation. Despite being a healthy mechanism, complete pool liquidation creates a risk of cascade liquidations and influences the whole TVL.

INFORMATIONAL-9 | RESOLVED

Redundant copying to memory.

Aurus.sol:

- calculateRepayInterest() on lines 206, 207;
- calculateCollateralTokenAmount() on line 220;
- _estimateErc20Value() on line 292.

Copying an object from storage to memory is performed, but there is no specific value handling, the values are only read. In such cases, it is recommended to use storage pointers instead of copying to reduce gas consumption.

Recommendation:

Use storage pointers instead of copying to memory in these cases.

INFORMATIONAL-10 RESOLVED

Redundant validation that a pool is not added.

Aurus.sol: borrow(), repay().

At the beginning of these functions, it is checked that the address of the pool's collateral token is not equal to the zero address at the time `_getPool()` is called. However, this check is repeated when calling `calculateCollateralTokenAmount()` and `calculateRepayInterest()` respectively.

In such a case, it is recommended to define private functions for these calculations, not including the call of `_getPool()`, to reduce gas consumption. These functions can then be called in `borrow()`, `calculateCollateralTokenAmount()`, `repay()` and `calculateRepayInterest()` without repeating the check, and the functions `calculateCollateralTokenAmount()` and `calculateRepayInterest()` are changed from `public` to `external`.

Recommendation:

Create the private functions to prevent the rechecking.

INFORMATIONAL-11 RESOLVED

Local variable that are used once.

Aurus.sol:

- claimRewards(): reward;
- claimMultiple(): reward;
- claimUnliquidatedCollateral(): totalUnliquidatedCollateral;
- calculateRepayInterest(): timeBorrowed;
- calculateCollateralTokenAmount(): normalizedUSDCAmount, collateralTokenUSDCValue, collateralTokenAmount;
- _liquidatePool(): balanceAfter.

These local variables are used once. It is recommended to use the expressions directly at the point of use to reduce gas consumption.
It is worth noting that to detail and clarify a particular expression, a comment describing its meaning and components should be added above it.

Recommendation:
Use these expressions directly at the point of use instead of creating one-time variables.

INFORMATIONAL-12 RESOLVED

Initialization of a variable in a loop.

Aurus.sol: claimMultiple() on line 162.
The local variable `amount` is initialized and destroyed in each iteration of the loop. In such cases, it is recommended to declare the variable outside the loop and use assignment to reduce gas consumption.

Recommendation:
Declare the variable outside the loop.

INFORMATIONAL-13 RESOLVED

Excessive method.

Aurus.sol: _getPool().
This method gets a pool from the array by its ID. Then it only reads the `collateralToken` field from the storage once to verify that the pool exists and returns a storage pointer to the pool without performing any other actions. It is recommended to convert the method into a validation method and pass it a ready pointer. For example:

```
function _requireExistingPool(Pool storage pool) private view {
    if (address(pool.collateralToken) == address(0))
        revert PoolDoesNotExist();
}
```

Not copying the pool ID and pointer when returning allows to reduce gas consumption.

Recommendation:
Convert the method to remove unnecessary calculations **OR** use this check directly at the point of use.

INFORMATIONAL-14 RESOLVED

Unnamed constants.

Aurus.sol: calculateRepayInterest() on line 213; calculateCollateralTokenAmount() on line 224; _calculateRewardRate() on line 272.
The unnamed constants "365 days" and "100" are used in the functions. It is recommended to declare appropriate constants with meaningful names in order to maintain consistency and improve code readability.

Recommendation:
Declare constants with meaningful names

INFORMATIONAL-15 | RESOLVED

Confusing naming.

ERC20CollateralPool.interface.sol: line 97.
Error `*error ChainlinkDataToOld()*` naming is confusing due to divergence of meaning. The part of the naming "To" is expected to be "Too" since the error intended to inform that chainlink data is too old.

Recommendation:
Rename error to `*ChainlinkDataTooOld*` for preservation of correct meaning.

INFORMATIONAL-16 | RESOLVED

Copied functionality.

ERC20CollateralPool.sol: liquidatePool(), line 254.
The same check for whether the pool can be liquidated is written in a separate `*_isCompleted()*` function and can be used instead of the current implementation, that copies functionality.

Recommendation:
Use `*_isCompleted()*` function in `*liquidatePool()*` instead of copying its functionality.

INFORMATIONAL-17 | VERIFIED

Excessive validation.

ERC20CollateralPool.sol: _getPool(), line 352.
The validation of collateral token address not to be equal to zero address is implemented in the time of initialization of pools in `*ERC20CollateralPool.storage.sol*`, line 27. Thus, the validation in `*_getPool()*` method is unreachable and excessive, but influences the gas consumption.

Recommendation:
Remove excessive validation to reduce gas consumptions.

Post-audit. The validation were moved to the new function `*_requireExistingPool()*`. The Defactor team verified, that this validation is required since the purpose is to check if the pool exists and not if a collateral token is valid.

INFORMATIONAL-18 | RESOLVED

Pause functionality missing.

ERC20CollateralPool.sol: LiquidatePool().
As it was recommended to correctly implement pause functionality in Low-2, since this moment, the `*liquidatePool()*` function has become external and needs a pause mechanism to be implemented, too, to avoid unexpected behaviors and set an ability to token freeze.

Recommendation:
Implement pause mechanism for `LiquidatePool()` function.

	ERC20CollateralPool ERC20CollateralPoolStorage
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Failed

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Defactor in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the Defactor contract requirements for details about issuance amounts and how the system handles these.

ERC20CollateralPool

Initialize
- ✓ Trying to initialize twice
- ✓ Trying to initialize with invalid usdc token (49ms)
- ✓ Trying to initialize with invalid end time
- ✓ Trying to initialize with collateral token zero-address
- ✓ Trying to initialize with admin zero-address

Pause and unpause
- ✓ Pausing and unpausing contract
- ✓ Trying to pause contract by non-admin
- ✓ Trying to unpause contract by non-admin (39ms)

Lend
- ✓ Trying to lend when contract is paused
- ✓ Trying to lend to non-existent pool
- ✓ Trying to lend after pool already closed

Borrow
- ✓ Trying to borrow when contract is paused (45ms)
- ✓ Trying to borrow after pool already closed (227ms)
- ✓ Trying to borrow more than there are funds in pool (73ms)
- ✓ Trying to borrow in pool with not supported token (109ms)

Repay
- ✓ Borrower trying to repay when contract is paused (77ms)
- ✓ Borrower trying to repay after pool is closed (282ms)
- ✓ Borrower trying to repay twice (136ms)

Claim rewards
- ✓ Trying to claim rewards when contract is paused (116ms)
- ✓ Trying to claim rewards before pool is closed (109ms)
- ✓ Trying to claim rewards while pool is not completed (109ms)
- ✓ Trying to claim rewards twice (332ms)
- ✓ Lender claims rewards after pool is closed (176ms)

Claim multiple rewards
✓ Trying to claim multiple rewards when contract is paused (116ms)
✓ Trying to claim multiple rewards before pool is closed (109ms)
✓ Trying to claim multiple rewards while pool is not completed (109ms)
✓ Trying to claim multiple rewards after already claimed one of them (252ms)
✓ Lender claims multiple rewards after pool is closed (745ms)
✓ Lender claims multiple reward with no borrows done (75ms)
✓ Lender should properly claim multiple rewards with one borrow and no repay (142ms)
Claim unliquidated collateral
✓ Trying to claim unliquidated collateral when contract is paused (79ms)
✓ Trying to claim unliquidated collateral before pool is closed (109ms)
✓ Trying to claim unliquidated collateral while pool is not completed (116ms)
✓ Trying to claim unliquidated collateral twice (191ms)
✓ Borrower claims unliquidated collateral after pool is closed (201ms)
Liquidation
✓ Trying to liquidate when contract is paused
✓ Trying to liquidate before pool is closed (95ms)
✓ Trying to liquidate after pool is completed (161ms)
Number of borrows by the address
✓ Get the number of borrows (135ms)
Get pools
✓ Check correctness of pool's creation (152ms)

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% FUNCS	% BRANCH	% LINES
ERC20CollateralPool	100	98.39	100
ERC20CollateralPoolStorage	100	87.5	100
Total %	**100**	**95.83**	**100**

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by the Pools team

As a part of our work assisting Defactor in verifying the correctness of their contract code, our team has checked the complete set of tests prepared by the Defactor team.

We need to mention that the original code has a significant original coverage with testing scenarios provided by the Defactor team. All of them were also carefully checked and additional value validations were added by the team of auditors.

ERC20CollateralPool

Test scenario 1

- ✓ Lender1 lends 1000$ (48ms)
- ✓ Borrower1 borrows 1000$ (437ms)
- ✓ Forward 10 days
- ✓ Lender2 lends 1000$ (61ms)
- ✓ Forward 10 days
- ✓ Borrower1 repays 1000$ + interests (93ms)
- ✓ Rewards should be properly distributed
  - **Pool interests: 10%**
  - **Lender1 Reward: $4.1096**
  - **Lender2 Reward: $1.36986**

Test scenario 2

- ✓ Lender1 lends 1000$ (48ms)
- ✓ Forward 5 days
- ✓ Lender1 lends 1000$ (62ms)
- ✓ Borrower1 borrows 1000$ (412ms)
- ✓ Forward 5 days
- ✓ Lender2 lends 1000$ (54ms)
- ✓ Forward 5 days
- ✓ Lender1 lends 1000$ (48ms)
- ✓ Forward 5 days
- ✓ Borrower1 repays 1000$ + interests (139ms)
- ✓ Forward 10 days
- ✓ Borrower1 borrows 1000$ (66ms)
- ✓ Forward 5 days
- ✓ Borrower1 repays 1000$ + interests (68ms)
- ✓ Rewards should be properly distributed
  - **Pool interests: 15%**
  - **Lender1 Reward: $6.50688**
  - **Lender2 Reward: $1.71233**

Test scenario 3

✓ Lender1 lends 1000$ (42ms)
✓ Borrower1 borrows 50$ (382ms)
✓ Borrower1 borrows 50$ (72ms)

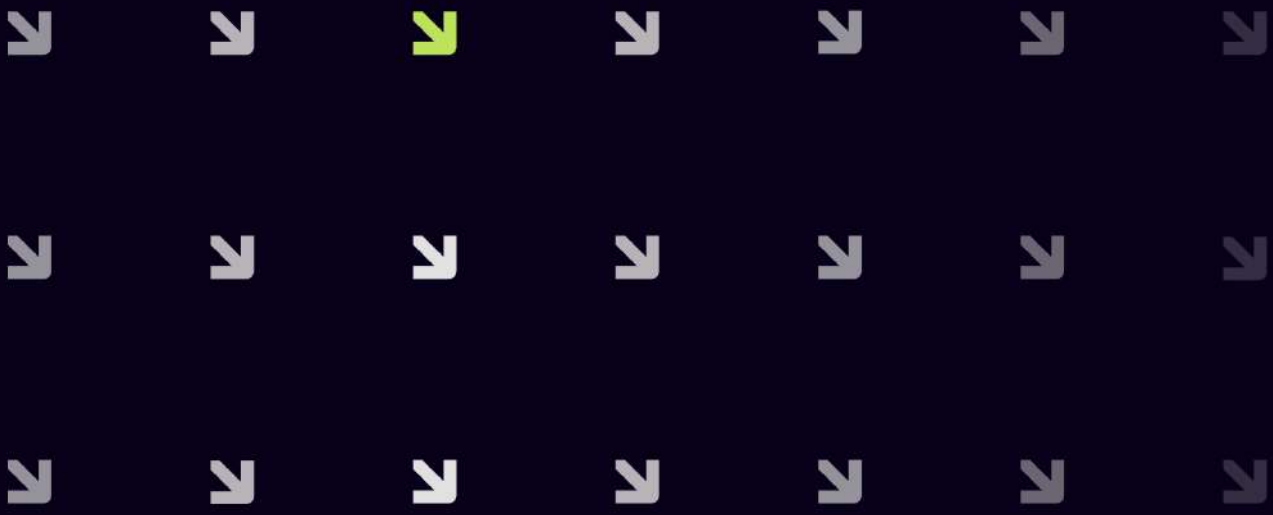
The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% FUNCS	% BRANCH	% LINES
ERC20CollateralPool	100	98.39	100
ERC20CollateralPoolStorage	100	87.5	100
Total %	**100**	**95.83**	**100**

We are grateful for the opportunity to work with the Defactor team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Defactor team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.



zokyo