Tribal - Protocol

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 17th, 2023 - August 4th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/01/2023
0.2	Document Edit	08/02/2023
0.3	Document Edit	08/04/2023
0.4	Draft Review	08/05/2023
0.5	Draft Review	08/07/2023
1.0	Remediation Plan	08/23/2023
1.1	Remediation Plan Review	08/28/2023
1.2	Remediation Plan Review	08/28/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
Luis Buendia	Halborn	Luis.Buendia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Tribal engaged Halborn to conduct a security assessment on their smart contracts beginning on July 17th, 2023 and ending on August 4th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to verify the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the Tribal team.

1.3 SCOPE

The security assessment was scoped to the following repository of smart contracts:

Commit ID: 6d2bdd564c5bd41f1b25f18bcc493cbcf5324c24

REMEDIATION COMMITS:

- Repository: Tribal Protocol
  - Commit IDs:
    - e8c509b2015ecfc602cf4a456cf78ce60069fef3
    - 48f177f95c5201e89e8a732aa30bc387ea93164f
    - 9d31d4d32bde19931c1ce6ecbe1c85d8d1e73418
    - 1e7a17894688608bd5680c617b5718e0dede0e14
    - 4372de644171592392b90335faf24ba2b08dd4be
    - eb38a85bdf38f55b620ad12fba2f89602a60e90a
    - ba5ebcab30bb820f657d3e6a7ee0eab53bac0900
    - 1e43f5a59ab052cf540b926e9402ae31d7c14e90
    - b319e8a6540373dc57e245afb9ecfb7379049382

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet environment. (Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	4	3	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) DRAIN FUNDS VIA executeRolloverAndBurn FUNCTION	Critical (10)	SOLVED - 08/14/2023
(HAL-02) BORROWER ADDRESS NOT ENFORCED TO BE WHITELISTED IN AUTHORITY	High (7.8)	SOLVED - 08/09/2023
(HAL-03) BORROWER CAN NOT RECOVER DEPOSIT IN FUNDING FAILED STAGE	Medium (5.6)	SOLVED - 08/09/2023
(HAL-04) trancheCoveragesWads NOT VALIDATED CAN PREVENT DEFAULT	Medium (5.6)	SOLVED - 08/10/2023
(HAL-05) POOL CAN BE FUNDED OVER maxFundingCapacity	Medium (5.6)	SOLVED - 08/23/2023
(HAL-06) adminOpenPool CAN ALSO BE EXECUTED BY LENDER/BORROWER/OWNER	Medium (5.6)	SOLVED - 08/10/2023
(HAL-07) CONSIDER USING whenNotPaused MODIFIER ON LENDING POOL	Low (3.1)	SOLVED - 08/10/2023
(HAL-08) nextLender AND nextAddress RETURN INCORRECT VALUE	Low (3.1)	ACKNOWLEDGED
(HAL-09) IMPLEMENTATION CONTRACTS CAN BE INITIALIZED	Low (3.1)	SOLVED - 08/10/2023
(HAL-10) GAS OPTIMIZATION	Informational (0.0)	SOLVED - 08/10/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) DRAIN FUNDS VIA executeRolloverAndBurn FUNCTION - CRITICAL(10)

Description:

The executeRolloverAndBurn function of the TrancheVault.sol contract transfers the funds of a lender from the tranche of an old pool to the tranche of a newer pool. This function is part of the rollover process, and it is intended to be called from the newer tranche.

The function does not contain any access control. This allows to drain the funds from the vault of any lender that deposited previously.

Code Location:

Listing 1: TrancheVault.sol

```
function executeRolloverAndBurn(address lender, uint256 rewards)
↳ external returns (uint256) {
    TrancheVault newTranche = TrancheVault(_msgSender());
    uint256 assets = approvedRollovers[lender][address(newTranche)
↳ ] + rewards;
    SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()),
↳ address(newTranche), assets);
    uint256 shares = convertToAssets(assets - rewards);
    _burn(lender, shares);
    return assets;
}
```

Proof Of Concept:

The next code snippet triggers the issue, draining the lender's funds from the vault. The scenario assumes the environment to be deployed in advanced.

Listing 2: Test.t.sol

```
function test_drainFunds() public {
    deployPool();
    address pool = pools[0];
    MockERC20(usdc).mint(borrower, 1000 ether);
    MockERC20(usdc).mint(lender, 1000 ether);

    vm.startPrank(borrower);
    ERC20(usdc).approve(pool, type(uint256).max);
    LendingPool(pool).borrowerDepositFirstLossCapital();
    vm.stopPrank();

    LendingPool(pool).adminOpenPool();

    address traVault = LendingPool(pool).trancheVaultAddresses(0);
    console.log("Vault",ERC20(usdc).balanceOf(traVault));
    console.log("lender",ERC20(usdc).balanceOf(lender));
    console.log("borrower",ERC20(usdc).balanceOf(borrower));

    vm.startPrank(lender);
    ERC20(usdc).approve(traVault, type(uint256).max);
    TrancheVault(traVault).deposit(10 ether, lender);
    vm.stopPrank();

    console.log("Vault",ERC20(usdc).balanceOf(traVault));
    console.log("lender",ERC20(usdc).balanceOf(lender));
    console.log("borrower",ERC20(usdc).balanceOf(borrower));

    vm.startPrank(borrower);
    TrancheVault(traVault).executeRolloverAndBurn(borrower, 10
↳ ether);
    vm.stopPrank();

    console.log("Vault",ERC20(usdc).balanceOf(traVault));
    console.log("lender",ERC20(usdc).balanceOf(lender));
    console.log("borrower",ERC20(usdc).balanceOf(borrower));
}
```

At the end of the execution it is possible to observe the logs with the balance of the vault decreased, and the balance of the borrower increased.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:C/R:N/S:U (10)

Recommendation:

Consider adding a permission modifier that restricts this function execution from non-trusted sources.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: b319e8a6540373dc57e245afb9ecfb7379049382

4.2 (HAL-02) BORROWER ADDRESS NOT ENFORCED TO BE WHITELISTED IN AUTHORITY - HIGH (7.8)

Description:

The system is designed to interact with a central contract that handles the roles of all users addresses. However, this condition is not true for Borrower addresses. Although there is a defined role for it, and the Authority contract supports it, the address introduced on the deployPool function parameters is not validated. So it is possible to introduce and borrower address that has not the appropriate permission role on the Authority contract.

The LendingPool.sol contract handles this address in its current implementation by using a modifier that just checks for the borrower address that was introduced on the initialize parameter.

Code Location:

Listing 3: LendingPool.sol

```
modifier onlyPoolBorrower() {
    _onlyPoolBorrower();
    _;
}

function _onlyPoolBorrower() internal view {
    require(_msgSender() == borrowerAddress, "LP003"); // "
 ↳ LendingPool: not a borrower"
}
```

Proof Of Concept:

The next code snippet is used to illustrate the described issue. Please note that the line that adds the borrower to the authority contract is commented.

Listing 4: Test.t.sol

```
    function deployPool() private {
        authority.addLender(lender);
       // authority.addBorrower(borrower);

        uint256[] memory trancheAPRsWads = new uint256[](1);
        uint256[] memory trancheBoostedAPRsWads = new uint256[](1)
↳ ;
        uint256[] memory trancheBoostRatios = new uint256[](1);
        uint256[] memory trancheCoveragesWads = new uint256[](1);

        trancheAPRsWads[0] = 0.1 ether;
        trancheBoostedAPRsWads[0] = 0.1 ether;
        trancheBoostRatios[0] = 2e12;
        trancheCoveragesWads[0] = 1 ether;
        LendingPool.LendingPoolParams memory params =  LendingPool
↳ .LendingPoolParams({
            name: "LP-1",
            token: "T-LP-1",
            stableCoinContractAddress: usdc,
            platformTokenContractAddress: address(pToken),
            minFundingCapacity: 10000e6,
            maxFundingCapacity: 12000 ether,
            fundingPeriodSeconds: 60 days,
            lendingTermSeconds: 90 days,
            borrowerAddress: borrower,
            firstLossAssets: 10e6,
            borrowerTotalInterestRateWad: 0.15 ether,
            repaymentRecurrenceDays: 30,
            gracePeriodDays: 5,
            protocolFeeWad: 0.1 ether,
            defaultPenalty: 0.5 ether,
            penaltyRateWad: 0.02 ether,
            tranchesCount: 1,
            trancheAPRsWads: trancheAPRsWads,
            trancheBoostedAPRsWads: trancheBoostedAPRsWads,
            trancheBoostRatios: trancheBoostRatios,
            trancheCoveragesWads: trancheCoveragesWads
        });
        uint256[] memory fundingSplitWads = new uint256[](1);
        fundingSplitWads[0] = 1 ether;
        pools.push(poolfactory.deployPool(params, fundingSplitWads));
    }

    function test_borrow() public {
        deployPool();
        address pool = pools[0];
        MockERC20(usdc).mint(borrower, 1000 ether);
        MockERC20(usdc).mint(lender, 15000 ether);
        MockERC20(usdc).mint(lender2, 15000 ether);

        vm.startPrank(borrower);
        ERC20(usdc).approve(pool, type(uint256).max);
        LendingPool(pool).borrowerDepositFirstLossCapital();
        vm.stopPrank();

        LendingPool(pool).adminOpenPool();
        address traVault = LendingPool(pool).trancheVaultAddresses(0);

        vm.startPrank(lender);
        ERC20(usdc).approve(traVault, type(uint256).max);
        TrancheVault(traVault).deposit(10000 ether, lender);
        vm.stopPrank();


        LendingPool(pool).adminTransitionToFundedState();

        vm.startPrank(borrower);
        LendingPool(pool).borrow();
        vm.stopPrank();
```

LendingPool.sol contract is no required to be whitelisted on the authority contract.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:M/R:N/S:C (7.8)

Recommendation:

Consider implementing mechanisms that do not allow the correct authorized addresses to interact with the system as specified in the documentation.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: e8c509b2015ecfc602cf4a456cf78ce60069fef3

4.3 (HAL-03) BORROWER CAN NOT RECOVER DEPOSIT IN FUNDING FAILED STAGE - MEDIUM (5.6)

Description:

The Funding Failed stage is just achieved when the owner or administrator executes the adminTransitionToFundedState function and the total collected assets does not cover the minimum funding capacity. Before arriving to that state, the borrower should have executed the borrowerDepositFirstLossCapital function, where it deposits in the pool part of the collateral used for the loan.

Nonetheless, when the pool enters the Funding Failed stage, lenders can withdraw their assets, however there is no way for the borrower to recover the deposit, locking its funds permanently in the pool.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Consider implementing a function that allows the borrower to recover the deposit in case of getting to the Funding Failed stage.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: 48f177f95c5201e89e8a732aa30bc387ea93164f

4.4 (HAL-04) trancheCoveragesWads NOT VALIDATED CAN PREVENT DEFAULT - MEDIUM (5.6)

Description:

The trancheCoveragesWads array parameter specified on the call to the deployPool function is not validated to meet a specific length or be withing a certain range of values. This array is used in the _transitionToDefaultedStage function, where it is used to calculate the amount of corresponding assets to repay in case of default to the different actors on the pool.

To do that calculation, it is iterated using the length of the trancheVaultAddresses array. However, it is not validated that the length is the same. Moreover, it is multiplied by the availableAssets to calculate the assetsToSend on each iteration. If the length of the array is smaller than the number of tranches or its values are bigger or equal than 10**18, then this function will always revert and prevent the transition to the default stage.

Code Location:

Listing 5: LendingPool .sol (Line 425)

```
function _transitionToDefaultedStage() internal {
    defaultedAt = uint64(block.timestamp);
    currentStage = Stages.DEFAULTED;
    _claimInterestForAllLenders();
    // TODO: update repaid interest to be the total interest paid
 ↳ to lenders
    // TODO: should the protocol fees be paid in event of default
    uint availableAssets = _stableCoinContract().balanceOf(address
 ↳ (this));

    for (uint i; i < trancheVaultAddresses.length; i++) {
        TrancheVault tv = trancheVaultContracts()[i];
        uint assetsToSend = (trancheCoveragesWads[i] * availableAssets) / WAD;
        uint trancheDefaultRatioWad = (assetsToSend * WAD) / tv.totalAssets();

        if (assetsToSend > 0) {
            SafeERC20.safeTransfer(_stableCoinContract(), address(tv), assetsToSend);
        }
        availableAssets -= assetsToSend;
        tv.setDefaultRatioWad(trancheDefaultRatioWad);
        tv.enableWithdrawals();
    }

    emit PoolDefaulted(defaultedAt);
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Consider validating the length of the trancheCoveragesWads parameter as well as the internal values to ensure the system does not revert if transitioning to the next state.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: 9d31d4d32bde19931c1ce6ecbe1c85d8d1e73418

4.5 (HAL-05) POOL CAN BE FUNDED OVER maxFundingCapacity - MEDIUM (5.6)

Description:

The maxFundingCapacity parameter sent to the deployPool function is validated to be more or equal to the minFundingCapacity parameter. The maximum capacity for each vault is calculated by multiplying it on the corresponding number of the fundingSplitWads array, sent as parameter to the deployPool function. The vaults check correctly on each deposit or mint that the total assets are not over the maxFundingCapacity.

However, it is not verified that the sum of all fundingSplitWads is equal to 10 ** 18. This allows the maximum funding capacity of the vaults to be higher than the maximum funding capacity of the pool.

Code Location:

Listing 6: PoolFactory .sol

```
function _deployTrancheVaults(
    LendingPool.LendingPoolParams calldata params,
    uint[] calldata fundingSplitWads,
    address poolAddress,
    address ownerAddress
) internal onlyOwner returns (address[] memory
↳ trancheVaultAddresses) {
    require(params.tranchesCount > 0, "Error TrancheCount must be
↳ gt 0");
    trancheVaultAddresses = new address[](params.tranchesCount);

    for (uint8 i; i < params.tranchesCount; ++i) {
        address impl = trancheVaultImplementationAddress;
        trancheVaultAddresses[i] = Clones.cloneDeterministic(impl,
↳   bytes32(nonces[impl]++));

        emit TrancheVaultCloned(trancheVaultAddresses[i], impl);

        TrancheVault(trancheVaultAddresses[i]).initialize(
            poolAddress,
            i,
            params.minFundingCapacity.mulDiv(fundingSplitWads[i],
 ↳ WAD),
            params.maxFundingCapacity.mulDiv(fundingSplitWads[i],
 ↳ WAD),
            string(abi.encodePacked(params.name, " Tranche ",
 ↳ Strings.toString(uint(i)), " Token")),
            string(abi.encodePacked("tv", Strings.toString(uint(i)
 ↳ ), params.token)),
            params.stableCoinContractAddress,
            address(authority)
        );
        TrancheVault(trancheVaultAddresses[i]).transferOwnership(
 ↳ ownerAddress);
    }
}
```

Listing 7: LendingPool .sol

```
function adminTransitionToFundedState() external onlyOwnerOrAdmin
 ↳ atStage(Stages.OPEN) {
    if (collectedAssets >= minFundingCapacity) {
        _transitionToFundedStage();
    } else {
        _transitionToFundingFailedStage();
    }
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Consider validating the elements inside the fundingSplitWads array to ensure they have the appropriate values. Furthermore, it can be considered to check the collectedAssets on the adminTransitionToFundedState are not bigger than maxFundingCapacity.

Another approach can be to store the values of the fundingSplitWads multiplied by the maxFundingCapacity on the _deployTrancheVaults function and check after the loop that the obtained summed value is actually equal or less than maxFundingCapacity.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: 1e7a17894688608bd5680c617b5718e0dede0e14

4.6 (HAL-06) adminOpenPool CAN ALSO BE EXECUTED BY LENDER/BORROWER/OWNER - MEDIUM (5.6)

Description:

The function adminOpenPool on the LendingPool.sol contract opens the pool once the borrower has deposited the collateral required on the on-chain components. As the name and comments over it suggest, it should be executed by an admin or owner role.

Nevertheless, the access control modifier of the function is onlyWhitelisted. This modifier also allows borrower and lender whitelisted across the whole system to execute the function and transit to open stage.

Code Location:

Listing 8: LendingPool .sol

```
/** @notice Marks the pool as opened. This function has to be called by *owner* when
 * - sets openedAt to current block timestamp
 * - enables deposits and withdrawals to tranche vaults
 */
function adminOpenPool() external onlyWhitelisted atStage(Stages.FLC_DEPOSITED) {
    openedAt = uint64(block.timestamp);
    currentStage = Stages.OPEN;

    for (uint i; i < trancheVaultAddresses.length; i++) {
        trancheVaultContracts()[i].enableDeposits();
        trancheVaultContracts()[i].enableWithdrawals();
    }

    emit PoolOpen(openedAt);
}
```

Listing 9: AuthorityAware .sol

```
function _onlyWhitelisted() internal view {
    require(
        owner() == msg.sender ||
            authority.isWhitelistedBorrower(msg.sender) ||
            authority.isWhitelistedLender(msg.sender) ||
            authority.isAdmin(msg.sender),
        "AA:W" // "AuthorityAware: caller is not a whitelisted
↳ borrower or lender"
    );
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Consider changing the modifier for onlyOwnerOrAdmin.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: 4372de644171592392b90335faf24ba2b08dd4be

4.7 (HAL-07) CONSIDER USING whenNotPaused MODIFIER ON LENDING POOL - LOW (3.1)

Description:

The LendingPool.sol contract implements the pause and unpause functions. However, it does not block the access to any functions when being paused.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:L/R:N/S:U (3.1)

Recommendation:

Consider using the whenNotPaused modifier on critical functions to prevent any undesired execution. Furthermore, consider implementing a mechanism that also pauses the vaults when the pool is paused and disabling deposits and withdrawals. Following this recommendation avoids needing to pause the components individually, creating a more secure and robust environment.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: eb38a85bdf38f55b620ad12fba2f89602a60e90a

4.8 (HAL-08) nextLender AND nextAddress RETURN INCORRECT VALUE - LOW (3.1)

Description:

The nextLender and nextAddress functions of the PoolFactory.sol contract attempt to return the next address of the corresponding implementation that the system will clone.

However, instead of nextLenders or nextTranches, they use the current nonce value plus one. The nonce mapping is incremented always after being used for cloning, so the current value is the one that will be used for generating the next address. However, it is important to mention that both functions are not used in the protocol. The function used are nextLenders and nextTranches that return the correct values.

Code Location:

Listing 10: PoolFactory .sol

```
function nextLender() public view returns(address) {
    return nextAddress(poolImplementationAddress);
}
```

Listing 11: PoolFactory .sol

```
function nextAddress(address impl) public view returns(address) {
    return Clones.predictDeterministicAddress(impl, bytes32(nonces
  ↳ [impl] + 1));
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:L/R:N/S:U (3.1)

Recommendation:

Consider using only the nonce to return the next generated address. Moreover, if these functions are not of any use, consider removing them.

Remediation Plan:

ACKNOWLEDGED: The Tribal team acknowledged this finding.

4.9 (HAL-09) IMPLEMENTATION CONTRACTS CAN BE INITIALIZED - LOW (3.1)

Description:

The implementation contracts of the protocol that are used by proxies do not disable the initialize function. The identified contracts that do not disable the initialize function of the implementation are:

- Authority.sol
- PoolFactory.sol
- FeeSharing.sol
- Staking.sol
- LendingPool.sol
- TrancheVault.sol

An uninitialized contract can be taken over by an attacker, which map impact the proxy or be used for social engineering.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:L/R:N/S:U (3.1)

Recommendation:

Consider adding the _disableInitializers function call on the constructor of each implementation.

Listing 12

```
constructor() {
    _disableInitializers();
}
```

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: ba5ebcab30bb820f657d3e6a7ee0eab53bac0900

4.10 (HAL-10) GAS OPTIMIZATION - INFORMATIONAL (0.0)

Description:

The LendingPool.sol contract calls the internal trancheVaultContracts function several times inside loops across the whole contract to obtain the address of the tranches. Note that every time this function is called, all stored tranche addresses are accessed and returned into an array. It is not required to call this function inside loops, just call it once before and store the results in a memory array.

Code Location:

Listing 13: LendingPool .sol

```
function adminOpenPool() external onlyWhitelisted atStage(Stages.
    FLC_DEPOSITED) {
    openedAt = uint64(block.timestamp);
    currentStage = Stages.OPEN;

    for (uint i; i < trancheVaultAddresses.length; i++) {
        trancheVaultContracts()[i].enableDeposits();
        trancheVaultContracts()[i].enableWithdrawals();
    }

    emit PoolOpen(openedAt);
}
```

Listing 14: LendingPool .sol

```
function _transitionToFundedStage() internal {
    fundedAt = uint64(block.timestamp);
    currentStage = Stages.FUNDED;

    for (uint i; i < trancheVaultContracts().length; i++) {
        TrancheVault tv = trancheVaultContracts()[i];
        tv.disableDeposits();
        tv.disableWithdrawals();
```

```
            tv.sendAssetsToPool(tv.totalAssets());
        }

        emit PoolFunded(fundedAt, collectedAssets);
}
```

Listing 15: LendingPool .sol

```
function _transitionToFundingFailedStage() internal {
        fundingFailedAt = uint64(block.timestamp);
        currentStage = Stages.FUNDING_FAILED;

        for (uint i; i < trancheVaultAddresses.length; i++) {
            trancheVaultContracts()[i].disableDeposits();
            trancheVaultContracts()[i].enableWithdrawals();
        }
        emit PoolFundingFailed(fundingFailedAt);
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider calling the function once and storing the results in a memory array to traverse it in the loop.

Remediation Plan:

SOLVED: The Tribal team solved the issue in the following commit ID: 1e43f5a59ab052cf540b926e9402ae31d7c14e90

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

- No major issues found by Slither.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

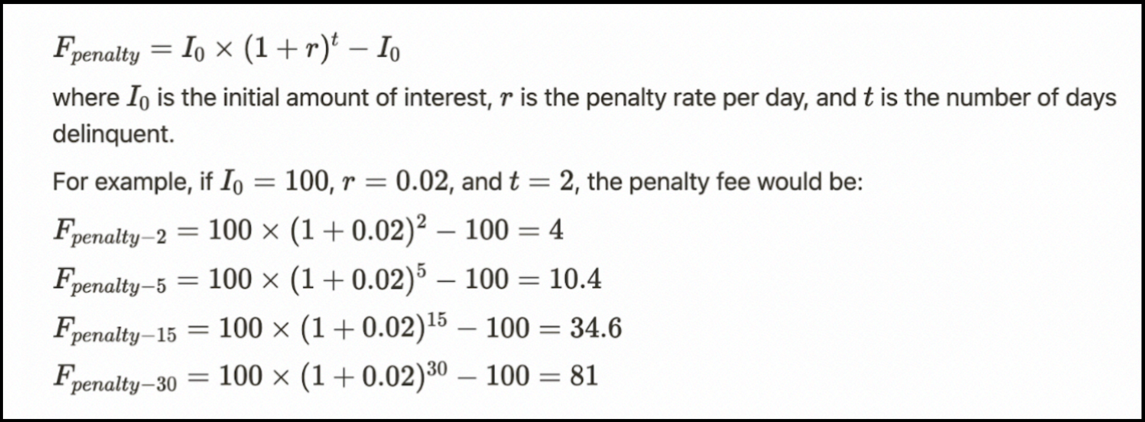
- No major issues were found by MythX.

WHITEPAPER FORMULAS VERIFICATION

This section contains all the formulas explained in the documentation and studies the implementation of it, considering if the code does match with the provided description. This analysis verifies the consistency of the logic regarding the expected functionality.

6.1 Penalties formula

The first formula to verify is the interest repayment calculation for the borrower. The formula presented in the documentation can be observed in the next picture.



$$F_{penalty} = I_0 \times (1+r)^t - I_0$$

where I_0 is the initial amount of interest, r is the penalty rate per day, and t is the number of days delinquent.

For example, if $I_0 = 100$, $r = 0.02$, and $t = 2$, the penalty fee would be:

$$F_{penalty-2} = 100 \times (1+0.02)^2 - 100 = 4$$

$$F_{penalty-5} = 100 \times (1+0.02)^5 - 100 = 10.4$$

$$F_{penalty-15} = 100 \times (1+0.02)^{15} - 100 = 34.6$$

$$F_{penalty-30} = 100 \times (1+0.02)^{30} - 100 = 81$$

The code corresponding to this formula is:

Listing 16: PoolCalculations .sol

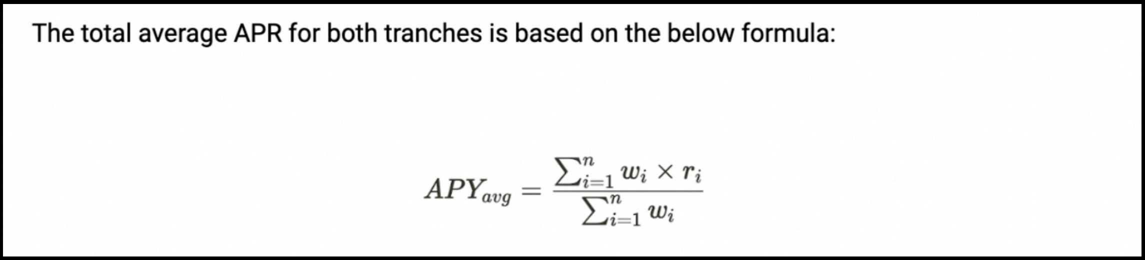
```
function borrowerPenaltyAmount(LendingPool lendingPool) public view returns (uint) {
    uint poolBalance = lendingPool.poolBalance();
    uint poolBalanceThreshold = lendingPool.poolBalanceThreshold();
    uint collectedAssets = lendingPool.collectedAssets();
    uint allLendersEffectiveAprWad = lendingPool.allLendersEffectiveAprWad();
    uint penaltyRateWad = lendingPool.penaltyRateWad();

    if (poolBalance >= poolBalanceThreshold) {
        return 0;
    }

    uint dailyLendersInterestAmount = (collectedAssets *
↳ allLendersEffectiveAprWad) / WAD / 365;
    uint balanceDifference = poolBalanceThreshold - poolBalance;
    uint daysDelinquent = balanceDifference /
↳ dailyLendersInterestAmount;

    if (daysDelinquent == 0) {
        return 0;
    }

    uint penaltyCoefficientWad = _wadPow(WAD + penaltyRateWad,
↳ daysDelinquent);

    uint penalty = (balanceDifference * penaltyCoefficientWad) /
↳ WAD - balanceDifference;
    return penalty;
}
```

r is equal to penaltyRateWad defined by admin parameter when creating the pool
- I is the initial amount of interest that is expressed as balanceDifference
The balanceDifference is calculated as poolBalanceThreshold - poolBalance
poolBalanceThreshold is the amount of interest that should be repaid subtracted to the first loss deposited. On the other hand, the poolBalance represents the effective amount that is in the pool, this means, the firstLossAsset plus the repaid interests. This represents the difference between what the pool should have and what it really has. As a result, this is used to calculate the time, t
- t is calculated as the amount that the pool should have divided by the interests that lenders should receive on a daily basis. This results in the total number of days that the interests have not been paid.

So, the formula does correspond with the smart contract implementation, holding the explained over the documentation.

6.2 APR average for tranches formula

The formula detailed on the documentation to calculate the APY/APR of the tranches can be observed on the next screenshot.



The total average APR for both tranches is based on the below formula:

$$APY_{avg} = \frac{\sum_{i=1}^{n} w_i \times r_i}{\sum_{i=1}^{n} w_i}$$

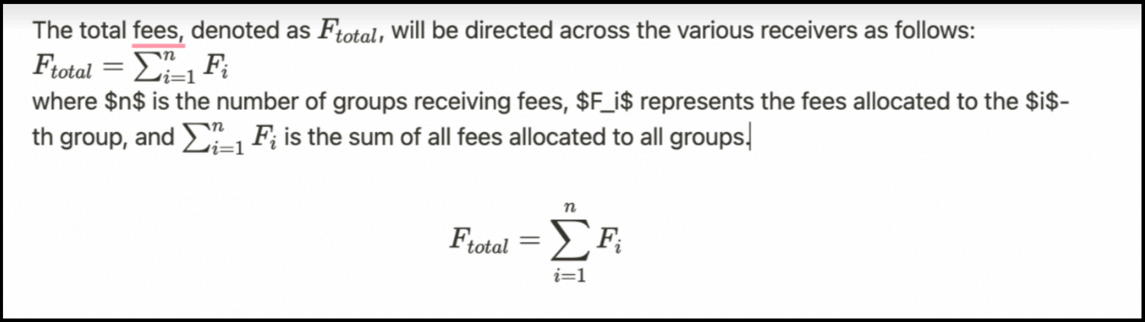
This formula can be decomposed on the numerator as the sum of the multiplication of the next values for each tranche:
- w that is the weight of the corresponding tranche
- r represents the APY of the tranche

And the previous number is the divided by the sum of all weights of each tranche.

Although it has not been observed a direct correspondence to this formula in the source code, there are several calculations that are performed to calculate the effective APR of the lenders. It is important to notice that the APR calculations on the code take into account the boosted yield tokens. All initial parameters for the calculations are provided by admin or governance to ensure a proper functionality of the protocol.

6.3 Fees formula

The formula for calculating the total fees obtained is expressed in the documentation as follows:



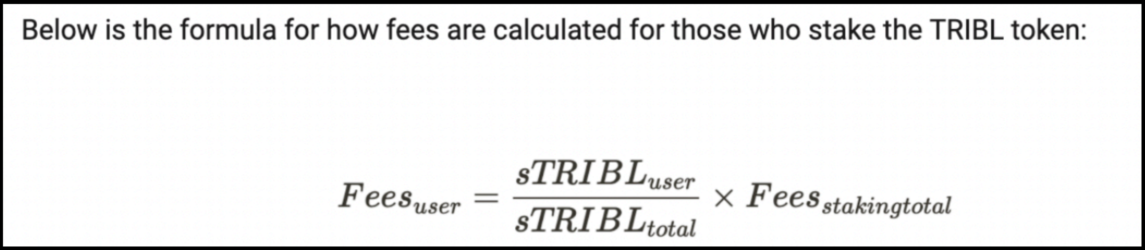
The total fees, denoted as F_{total}, will be directed across the various receivers as follows:
$F_{total} = \sum_{i=1}^{n} F_i$
where n is the number of groups receiving fees, F_i represents the fees allocated to the i-th group, and $\sum_{i=1}^{n} F_i$ is the sum of all fees allocated to all groups.

$$F_{total} = \sum_{i=1}^{n} F_i$$

This formula for the fees is trivial to calculate, as expressing the total fees as the sum of all individual obtained fees is meant to be a tautology in terms of a logical verification. However, it can be observed the distribution implementation on the staking rewards formula analysis.

6.4 Staking rewards formula

The formula detailed in the documentation used to calculate the staking rewards obtained from protocol fees can be observed in the next picture.



Below is the formula for how fees are calculated for those who stake the TRIBL token:

$$Fees_{user} = \frac{sTRIBL_{user}}{sTRIBL_{total}} \times Fees_{stakingtotal}$$

The fee distribution for the Staking contract is defined in code as follows:

Listing 17: FeeSharing .sol

```
function distributeFees() external onlyOwnerOrAdmin {
    uint balance = assetContract.balanceOf(address(this));
    // distribute shares
    for (uint i = 0; i < beneficiaries.length; i++) {
        uint amount = balance.mulDiv(beneficiariesSharesWad[i],
↳ WAD, MathUpgradeable.Rounding.Down);
        if (amount > 0) {
            if (i == 0) {
                address stakingAddress = beneficiaries[i];
                // first beneficiary is the staking contract.
                SafeERC20.safeApprove(assetContract, stakingAddress, amount);
                // Call addReward on the contract
                IStaking(stakingAddress).addReward(amount);
            } else {
                // otherwise just transfer funds to benificiary
                SafeERC20.safeTransfer(assetContract, beneficiaries[i], amount);
            }
        }
    }
}
```

Listing 18: Staking .sol

```
function _calculateRewards(address account) private view returns (uint) {
    uint shares = stakedBalanceOf[account];
    return (shares * (rewardIndex - rewardIndexOf[account])) / WAD;
}
```

THANK YOU FOR CHOOSING

HALBORN