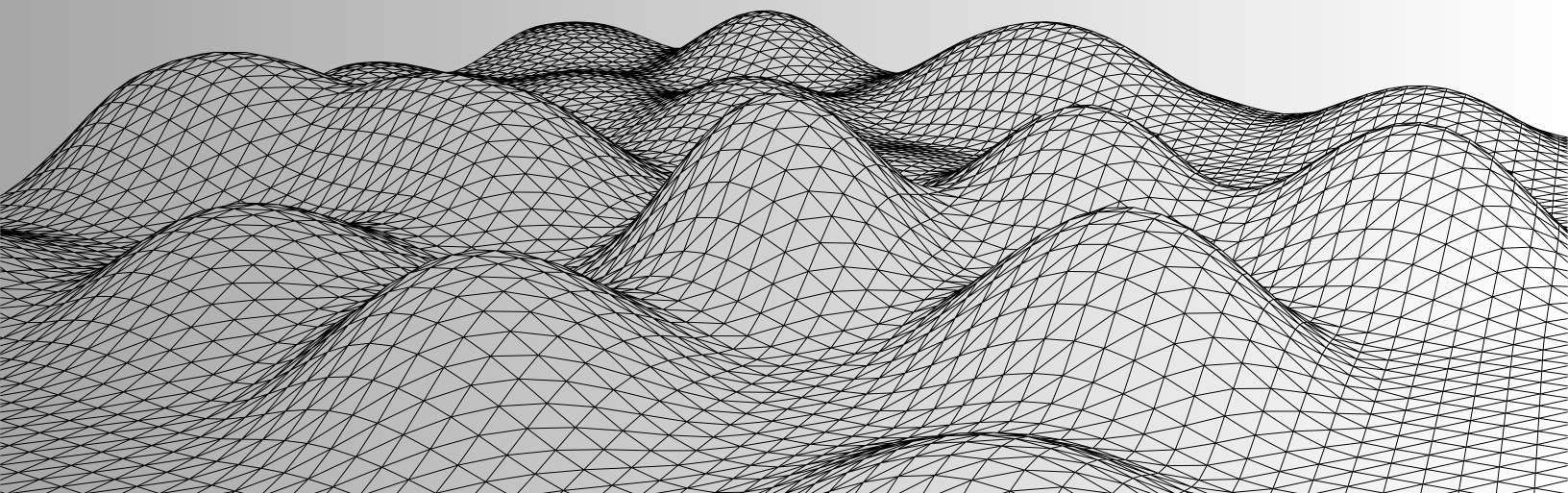
BlockGenesys

Smart Contract
Security Audit Report

A51 Finance
Concentrated Liquidity Tool

29 March, 2024



Introduction

This report may include confidential information regarding the IT systems and intellectual property of the Client, along with details on potential vulnerabilities and their possible exploitation. Public disclosure of this report is only permitted with prior consent from the Client. The Client must authorize any further distribution or publication of this report.

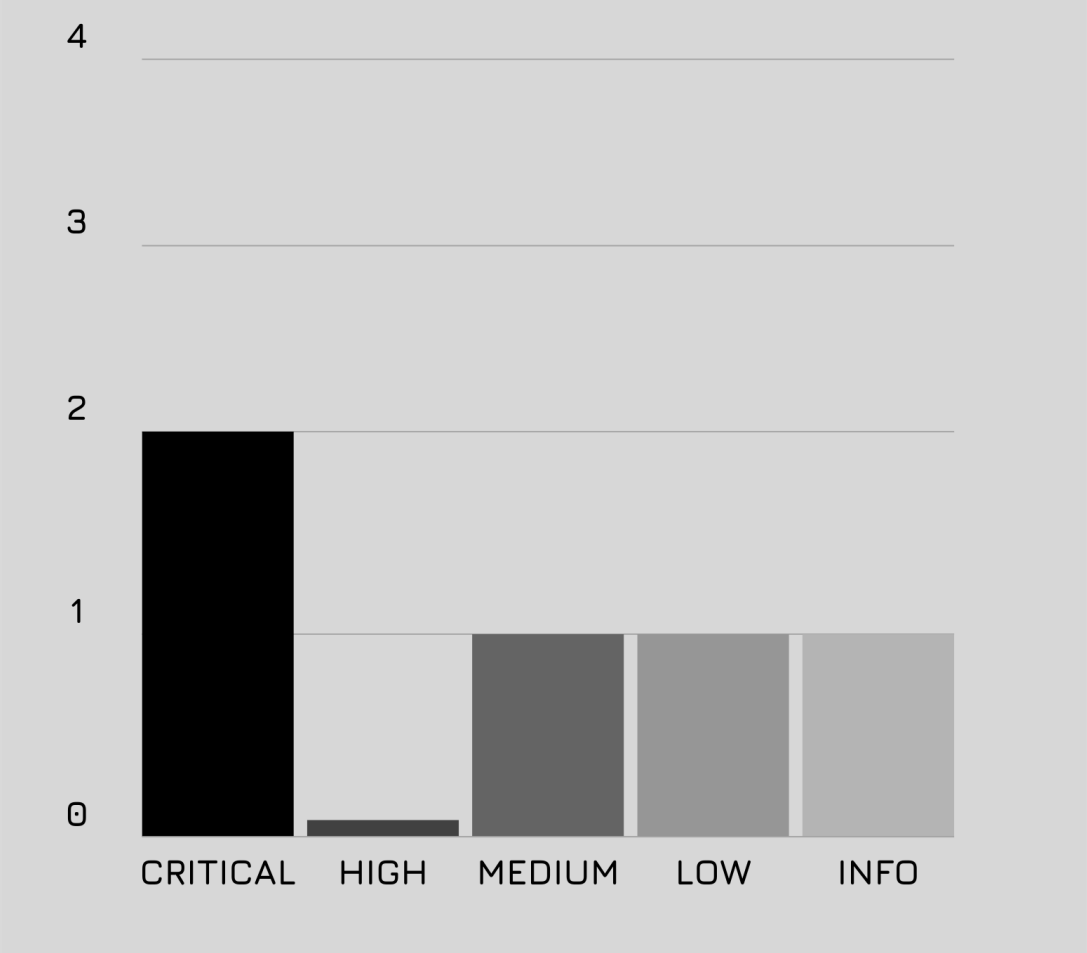
Name	A51 Finance (Concentrated Liquidity Tool)
Website	a51.finance
Repository/Source	Private Repository
Platform	L1
Network	Ethereum and EVM-Compatible Chains
Languages	Solidity
Initial Commit	-
Final Commit	-
Changelog	4 March, 2024 - Preliminary Report 29 March, 2024 - Final Report

Table Of Contents

Executive Summary

This initial audit report provides a detailed overview of the findings from our security audit of A51 Finance (Concentrated Liquidity Tool)

Issues Overview



Project Overview

A51 Finance's Concentrated Liquidity Tool is a DeFi application built on the Polygon Network, leveraging QuickSwap to allow users to contribute tokens to liquidity pools. What sets this tool apart are modular strategies that govern liquidity provision, enabling liquidity providers (LPs) to optimize yields through automated management. Depending on their preferences, users can select between basic or advanced modes for liquidity management, with the strategies adjusting automatically to maximize returns.

This dual-mode functionality helps users effectively manage their funds while taking advantage of concentrated liquidity on AMMs. By offering greater control over capital deployment and risk exposure, the tool ensures both novice and experienced users can tailor liquidity strategies to align with their goals.

Audit Checklist

During our comprehensive audit of Concentrated Liquidity Tool, we will be assessing the following potential vulnerabilities commonly encountered in Ethereum smart contracts:

- Reentrancy Attacks
- Integer Overflow and Underflow
- Insecure Initialization
- Missing Return Value Checks for External Calls
- Improper Access Control
- Denial of Service (DoS)
- Race Conditions
- Timestamp Dependence
- Front-Running
- Oracle Manipulation
- Price Feed Accuracy
- Token Decimal Handling Errors
- Funds Misdirection
- Authentication and Authorization Checks
- Logic Errors in Round Management
- Arithmetic Underflow or Overflow in Token Calculations
- Multisig Wallet Security
- Event Emission Omissions

Scope

Critical Areas of Focus

Liquidity Management and Strategy Modules

The audit will prioritize evaluating the modules that govern liquidity for maximum yield. This includes validating that the basic and advanced modes manage users' funds effectively and align with the defined parameters. Ensuring the strategies optimize liquidity based on market conditions and user settings will be a key focus to maintain the integrity of the platform.

Interaction with QuickSwap and AMMs

Given that A51 Finance integrates with QuickSwap on the Polygon network, the audit will scrutinize the concentrated liquidity tool's interactions with AMMs to ensure safe, seamless, and correct token deposits. This includes verifying correct handling of price slippage, pool interactions, and ensuring liquidity addition and withdrawal processes function securely without exposing users to risks.

Fund Management and Security Protocols

The audit will examine how user funds are deposited, managed, and distributed within the liquidity pools. Special attention will be given to reentrancy vulnerabilities, secure fund transfers, and address validation to prevent unauthorized withdrawals. Ensuring that funds are managed efficiently within the ecosystem is essential for trust and security.

Access Control and Administrative Functions

The audit will evaluate the access control mechanisms implemented across the protocol to ensure that only authorized personnel can perform sensitive actions, such as updating strategies, pausing liquidity operations, or modifying key parameters. Special attention will be given to the segregation of duties to avoid misuse of administrative privileges and to protect the protocol from potential threats.

Out of Scope

The scope of this audit does not extend to the user interface or any front-end integration with the Concentrated Liquidity Tool contracts. It also needs to examine external smart contracts or the interaction of the contract with other decentralized applications beyond the direct functionality provided. Furthermore, the audit does not include the deployment process or network-specific considerations such as gas optimization strategies.

Methodology

The audit process commenced with a reconnaissance phase to develop a fundamental understanding of the contract's intended functionality and security posture. This was followed by a filtered code review technique, focusing on the assumptions developed from the documentation provided by the client. The manual review phase sought to identify logical flaws and potential vulnerabilities within the codebase. This phase was complemented by recommendations for code optimizations and the implementation of security best practices, alongside identifying and addressing any false positives detected by automated tools.

This narrative provides a structured approach to the security audit of the Concentrated Liquidity Tool contracts, emphasizing the prevention of security issues and ensuring that the contract functions without adverse effects on users.

Security Review Report

Findings Count

Issues	Severity Level	Open	Resolved	Acknowledged
2	Critical	-	2	-
0	High	-	-	-
1	Medium	-	1	-
1	Low	-	1	-
1	Info	-	-	1
5	TOTAL	0	4	1

Summary of Findings

#	Findings	Risk	Status
1	Insufficient input validation can lock user funds into strategy	Critical	Fixed
2	TWAP Manipulation and Liquidity Risk.	Critical	Fixed
3	Incomplete Implementation in shiftLeft and shiftRight.	Medium	Fixed
4	Insufficient Error Messaging in require Statements for Authorization Checks.	Low	Fixed
5	Implementing Dual-Signature Fail-Safe Mechanism for Emergency Withdrawals.	Info	Acknowledged

Comprehensive Review of Findings

Issue No	01
Issue	Insufficient input validation can lock user funds into strategy.
Status	Fixed
Severity	Critical

Description

In the CLTBase contract, there is a potential vulnerability where the strategy owner can renounce ownership by setting the owner address to zero. This action could lead to a critical issue during fee transfer in the withdraw function. If the withdraw function attempts to transfer fees to the renounced strategy owner, the ERC20 token transfer will fail because tokens cannot be transferred to the zero address according to the ERC20 standard. As a result, if the fee transfer fails, user funds may become trapped in the strategy contract indefinitely without an emergency withdrawal mechanism in place. It's essential to consider this behavior when designing the system and ensure that appropriate checks and validations are in place to handle ownership changes.

Impact

- User funds may become stuck in the strategy contract if the withdrawal function fails to transfer fees to the renounced strategy owner.
- Lack of an emergency withdrawal mechanism means users may be unable to retrieve their funds in case of contract failure or unexpected circumstances.

Proof of Concept

```
function test_renounceStrategy() public {
        //deposit
        loopedDepositsToStrategy();
        loopedDepositsToUniswap();

        //swap
        uint256 swapAmount = 1e30;
        for (uint256 i = 0; i < 10; i++) {
            swapsByContract(swapAmount);
        }
        console.log("10 users swapped in univ3 pool");

        //renounce
        bytes32 strategyID = getStrategyID(address(this), 1);
        //setting some fee
        ICLTBase.PositionActions memory actions = createStrategyActions(1, 3, 0, 3, 0, 0);
        //sending zero address as strategy owner
        base.updateStrategyBase(strategyID, address(0), 1, 1, actions);

        // update strategy fee
        base.getStrategyReserves(getStrategyID(address(this), 1));
        base.getStrategyReserves(getStrategyID(address(this), 2));
        base.getStrategyReserves(getStrategyID(address(this), 3));

        //withdraw
        try this.loopedWithdrawFromStrategy() {
            emit CallSucceeded();
        } catch Error(string memory reason) {
            emit CallFailed(reason);
        }
    }
```

Proposed Solution

Place a zero address check while changing the new strategy. Additionally, it is suggested to introduce a well-thought emergency withdrawal mechanism (as mentioned in 4 below) in case funds are stuck in any strategy.

Issue No	02
Issue	TWAP Manipulation and Liquidity Risk
Status	Fixed
Severity	Critical

Description

A significant vulnerability in the ALP tool's handling of liquidity strategies, specifically within the ShiftLiquidity function of the CLTBase contract. The core of the issue lies in the absence of a checkDeviation safeguard, which opens up the potential for frontrunning and price manipulation attacks in volatile pools.

An attacker, by manipulating the current tick through strategic trades right before executing key functions such as deposit, withdraw, or shiftLiquidity, can adversely affect the protocol's operations. For instance, in a volatile pool, a malicious actor can execute a large trade that drastically moves the current tick, making the protocol compute an incorrect amount for token conversion, leading to the deposit or withdrawal of funds at a manipulated and unfavorable rate. This kind of attack not only disrupts the intended liquidity provision strategy but also exposes users to potential financial losses due to Miner Extractable Value (MEV) attacks.

Impact

The absence of the checkDeviation modifier in critical user-facing functions (deposit, withdraw) and admin/operator functions (shiftLiquidity) significantly heightens the risk of adverse user experiences and financial losses. Users could find their transactions executing at rates far removed from fair market values, resulting in direct financial losses or inefficient capital allocation. Furthermore, these vulnerabilities can be exploited by attackers to perform MEV attacks, extracting value from users' transactions through frontrunning or sandwich attacks, ultimately eroding trust in the platform and potentially leading to substantial liquidity withdrawal.

Proof of Concept

We tried to depict a front-running scenario in our local environment which is far different from a real world scenario where the functionality works but with unfavorable results for users' funds.

```
function testFrontrunning() public {
    vm.prank(specialUsers[1]);
    base.deposit(
      ICLTBase.DepositParams({
        strategyId: getStrategyID(address(this), 1),
        amount0Desired: 100 ether,
        amount1Desired: 100 ether,
        amount0Min: 0,
        amount1Min: 0,
        recipient: specialUsers[1]
      })
    );
    emit log("Deposit 1st user thru A51 strategy");
    emit log_named_uint("1st user balanceOf t0", token0.balanceOf(specialUsers[1]));
    emit log_named_uint("1st user balanceOf t1", token1.balanceOf(specialUsers[1]));

    (, int24 currTick,,,,,) = pool.slot0();
    emit log("\nFor the record: ");
    emit log_named_int("\tCurrent Tick from slot0 BEFORE swaps: \t", currTick);

    emit log_named_uint("\t Pool T0 balance: \t", token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t", token1.balanceOf(address(pool)));

    emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
    emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));

    emit log("\n====================First Deposit====================\n");
    for (uint256 i = 0; i < 2; i++) {
      vm.warp(block.timestamp + 1 minutes);
      //swap t0 for t1
      vm.prank(user3[1]);

      router.exactInputSingle(
        ISwapRouter.ExactInputSingleParams({
          tokenIn: address(token0),
          tokenOut: address(token1),
          fee: 500,
```

```
            recipient: user3[1],
            deadline: block.timestamp + 1 hours,
            amountIn: 49.999 ether,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        })
    );
}
console.log("\n Record: 2 x 1 ether swaps for t0->t1 \n");

(, currTick,,,,) = pool.slot0();
emit log("For the record: ");
emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t", currTick);

emit log_named_uint("\t Pool T0 balance: \t", token0.balanceOf(address(pool)));
emit log_named_uint("\t Pool T1 balance: \t", token1.balanceOf(address(pool)));

emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));

emit log("\n====================First Swap====================\n");

vm.prank(specialUsers[2]);
(uint256 tokenIdOf2ndDepositor,, uint256 amount0, uint256 amount1) = base.deposit(
    ICLTBase.DepositParams({
        strategyId: getStrategyID(address(this), 1),
        amount0Desired: 100 ether,
        amount1Desired: 100 ether,
        amount0Min: 0,
        amount1Min: 0,
        recipient: specialUsers[2]
    })
);

vm.warp(block.timestamp + 1 minutes);

emit log("\n Deposit 2nd user thru A51 strategy");
emit log_named_uint("2nd user balanceOf t0", token0.balanceOf(specialUsers[2]));
emit log_named_uint("2nd user balanceOf t1", token1.balanceOf(specialUsers[2]));

emit log_named_uint("\t Pool T0 balance: \t", token0.balanceOf(address(pool)));
emit log_named_uint("\t Pool T1 balance: \t", token1.balanceOf(address(pool)));
```

```
        emit log("\n====================Second Deposit====================\n");

        for (uint256 i = 0; i < 2; i++) {
            vm.warp(block.timestamp + 1 minutes);
            //swap t0 for t1
            vm.prank(user3[1]);
            router.exactInputSingle(
                ISwapRouter.ExactInputSingleParams({
                    tokenIn: address(token1),
                    tokenOut: address(token0),
                    fee: 500,
                    recipient: user3[1],
                    deadline: block.timestamp + 1 hours,
                    amountIn: 75 ether,
                    amountOutMinimum: 0,
                    sqrtPriceLimitX96: 0
                })
            );
        }
        console.log("\n Record: 2 x 1 ether swaps for t0->t1 \n");

        (, currTick,,,,) = pool.slot0();
        emit log("For the record: ");
        emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t", currTick);

        emit log_named_uint("\t Pool T0 balance: \t", token0.balanceOf(address(pool)));
        emit log_named_uint("\t Pool T1 balance: \t", token1.balanceOf(address(pool)));

        emit log_named_uint("Swapper Balance T0", token0.balanceOf(user3[1]));
        emit log_named_uint("Swapper Balance T1", token1.balanceOf(user3[1]));
        emit log("\n====================Second Swap====================\n");

        (, int24 tick,,,,) = pool.slot0();
        int24 tickSpacing = key.pool.tickSpacing();

        tick = utils.floorTicks(tick, tickSpacing);

        ICLTBase.StrategyKey memory newKey = ICLTBase.StrategyKey({ pool: pool,
tickLower: -200, tickUpper: 0 });

        (,,,,,,, ICLTBase.Account memory accountStrategy1) =
```

```
base.strategies(getStrategyID(address(this), 1));

    // invalid ticks stored here because we are trying to add in range liquidity with only 1
asset
    vm.prank(base.owner());
    base.toggleOperator(base.owner());
    vm.prank(base.owner());
    base.shiftLiquidity(
      ICLTBase.ShiftLiquidityParams({
        key: newKey,
        strategyId: getStrategyID(address(this), 1),
        shouldMint: true,
        zeroForOne: false,
        swapAmount: int256(accountStrategy1.balance1 / 2),
        moduleStatus: "",
        sqrtPriceLimitX96: 0
      })
    );

    (, currTick,,,,,) = pool.slot0();
    emit log("For the record: ");
    emit log_named_int("\tCurrent Tick from slot0 AFTER swaps: \t", currTick);

    emit log_named_uint("\t Pool T0 balance: \t", token0.balanceOf(address(pool)));
    emit log_named_uint("\t Pool T1 balance: \t", token1.balanceOf(address(pool)));

    emit log("\n====================First ShiftBase====================\n");
  }
```

Proposed Solution

It is strongly recommended to incorporate a checkDeviation modifier in all critical functions (deposit, withdraw, and shiftLiquidity) to mitigate the risks associated with price manipulation and frontrunning. This safeguard should compare the current tick against a TWAP to ensure that the operation proceeds only if the price deviation is within acceptable limits (aka THRESHOLD).

Issue No	03
Issue	Incomplete Implementation in shiftLeft and shiftRight.
Status	Fixed
Severity	Medium

Description

Both shiftLeft and shiftRight functions appear to be missing return statements for certain execution paths. Specifically, if the current tick is not below the lower tick (for shiftLeft) or above the upper tick (for shiftRight), the function does not return any values, leading to potential undefined behavior or compilation errors.

In the scenario where currentTick is equal to any boundary tick, the shift function in our liquidity management contract fails to execute its intended logic due to its conditional check. As a result, when currentTick matches the boundary of our defined liquidity range, the function bypasses its logic designed to adjust tick positions, inadvertently returning zero (0) for both tickLower and tickUpper due to uninitialized return values. This behavior leads to failed transactions because the adjustment logic is not triggered, and users are not informed of the reason for failure through a clear error message.

Code Affected

```
if (currentTick < key.tickLower) {
    // Logic to adjust tickLower and tickUpper
}
```

Problematic Outcome:

- Transactions Fail Silently: The logic intended to shift liquidity in response to market conditions is skipped, and the function returns zeros without executing any adjustment.
- Lack of Error Messaging: Users receive no feedback or error message indicating why the liquidity adjustment did not occur, leading to confusion.

Proposed Solution

Error Handling for Unchanged Ticks: Implement a mechanism to check for and handle scenarios where tick adjustments result in no change (e.g., returning initial values or preventing transaction execution), including providing clear error messages to the user.

Issue No	04
Issue	Insufficient Error Messaging in require Statements for Authorization Checks
Status	Fixed
Severity	Low

Description

The _authorization and _authorizationOfStrategy functions require statements for authorization checks without providing error messages. This absence can confuse users during transaction failures, as no specific reason is disclosed. Additionally, throughout the codebase there exist multiple scenarios where the require statements do not print suitable error messages and thus making the debugging process difficult to track.

Code Affected

```
function _authorization(uint256 tokenID) private view {
    require(ownerOf(tokenID) == _msgSender());
}

function _authorizationOfStrategy(bytes32 strategyId) private view {
    if (strategies[strategyId].isPrivate) {
        require(strategies[strategyId].owner == _msgSender());
    }
}
```

Proposed Solution

Add descriptive error messages to require statements to clarify the cause of transaction rejections.

Issue No	05
Issue	Implementing Dual-Signature Fail-Safe Mechanism for Emergency Withdrawals
Status	Acknowledged
Severity	Info

Description

To enhance the security and trust in our system, especially concerning emergency withdrawal functionalities that are typically reserved for administrators, we propose implementing a dual-signature failsafe mechanism. This mechanism requires both the user's and the system administrator's signatures for withdrawals, ensuring mutual dependence and enhanced trust.

Rationale
Current concerns revolve around the risk associated with granting exclusive emergency withdrawal rights to administrators, which could potentially lead to theft or misuse. By introducing a mechanism where both the user initiates a withdrawal request via a signature and the administrator co-signs this request, we create a balanced trust model. This approach ensures that:

- Users retain control and oversight over their assets.
- Administrators can facilitate emergency withdrawals without unilateral control, mitigating the risk of abuse.

Mechanism
The suggested FailSafe contract utilizes EIP-712 typed data signing to securely process withdrawal requests. The process involves:

- User Request: Users generate a signed withdrawal request containing their tokenID and sender address.
- Administrator Approval: The system verifies the user's signature and requires a subsequent signature from the system administrator, confirming the request.
- Emergency Withdrawal Execution: Upon successful verification of both signatures, the emergency withdrawal logic is executed.

This method ensures that withdrawals in emergency scenarios are executed with consent from both parties, thereby reinforcing security and mutual trust.

Security and Trust Benefits

- Mutual Dependency: By requiring consent from both users and administrators, the system mitigates risks associated with unilateral access to emergency withdrawal functionalities.
- Enhanced Transparency: The dual-signature requirement ensures transparent processing of emergency withdrawals, with traceable approvals from both parties.
- Reduced Risk of Misuse: The added layer of verification by both user and administrator acts as a deterrent against potential misuse or theft, protecting user assets.

Disclaimer

This smart contract audit report ("Report") is provided by BlockGenesys ("Auditor") for the benefit of the client ("Client") and is subject to the following terms and conditions:

The scope of this audit is limited to a review of the smart contract code provided by the Client. The Auditor has conducted a technical code analysis to identify potential security vulnerabilities and deviations from best practices. The Auditor has not undertaken a formal verification or validation of the functional requirements of the smart contract.

The Auditor has performed the audit to the best of their ability, given the current state of knowledge and technology. However, the Auditor does not warrant that the smart contract is free from all vulnerabilities or will operate as intended in all environments.

The Auditor shall not be liable for any direct, indirect, incidental, special, or consequential damages, including but not limited to loss of data, loss of profits, or any other loss arising from the use or inability to use the smart contract.

This Report is intended solely for the use of the Client and may not be used, reproduced, or distributed to any third party without the prior written consent of the Auditor. The Report does not constitute an endorsement or approval of the smart contract by the Auditor.

The Client is responsible for the deployment and operation of the smart contract. The Client agrees to conduct their testing and verification of the smart contract to ensure it meets their requirements and to implement any recommended changes identified in the Report.

The Auditor reserves the right to update or revise the Report if new information or vulnerabilities are discovered. The Client is encouraged to keep their smart contract code up-to-date and to review the security of their smart contract periodically.

By engaging the Auditor's services, the Client acknowledges and agrees to the terms and conditions outlined in this Disclaimer.