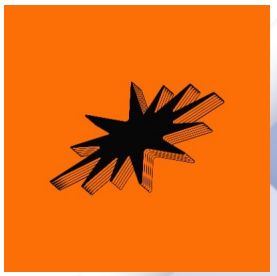
Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Sprinter Liquidity Pool

Veridise Inc.

Mar 11, 2025

► **Prepared For:**

Sygma-Labs
https://buildwithsygma.com

► **Prepared By:**

Ajinkya Rajput
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Mar. 11, 2025 Initial Draft

Contents

1 Executive Summary

From Mar. 04, 2025 to Mar. 07, 2025, Sygma-Labs engaged Veridise to conduct a security assessment of their Sprinter Liquidity Pool protocol. The security assessment covered contracts that implemented liquidity pools and a rebalancer. The liquidity pools manage the assets deposited in the protocol. The rebalancer contracts move assets across chains. Veridise conducted the assessment over 8 person-days, with 2 security analysts reviewing the project over 4 days on commit `d98bb99`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. Sprinter is an intent-based solver that works across multiple chains. The Sprinter Liquidity Pool protocol implements contracts that allow users to provide liquidity to solvers and earn rewards. The assets in the protocol are managed by Liquidity Pools that are deployed on all chains. `LiquidityPool` and `LiquidityPoolAave` are two types of liquidity pools in the scope of the audit. The `LiquidityPool` holds the assets with itself while the `LiquidityPoolAave` immediately forwards the assets to the AAVE protocol and acts as an interface to AAVE Pools. Part of the rewards earned by the protocol are distributed among the liquidity providers. The `Rebalancer` contract transfers assets across supported chains. A rebalance is triggered by a privileged address based on off chain logic. The protocol interacts with out of scope off-chain components and third party protocols for rebalancing and borrowing.

Code Assessment. The Sprinter Liquidity Pool developers provided the source code of the Sprinter Liquidity Pool contracts for the code review.

The source code appears to be mostly original code written by the Sprinter Liquidity Pool developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Sprinter Liquidity Pool developers provided project requirement documents and API design documents.

The source code contained a test suite, which the Veridise security analysts noted tested for wide variety of situations.

Summary of Issues Detected. The security assessment uncovered 7 issues, one of which is a high severity issue. The issue V-SLP-VUL-001 points out missing parameters in an on-chain signature check. However, the developers assure that the off-chain components prevent from this issue from being exploited. The assessment also uncovered 4 warnings and 1 informational finding. The Sprinter Liquidity Pool developers have acknowledged 4 issues without fixes and fixed 3 issues.

Recommendations. After conducting the protocol assessment, the security analysts had some suggestions to improve the Sprinter Liquidity Pool protocol. The protocol makes heavy usage of privileged roles which are trusted to perform security critical actions such as reflecting asset changes from the pool contracts to the hub contracts. The logic controlling these actions is deferred to an off-chain component of the project. When dealing with cross-chain applications, this becomes an inevitability; however, the analysts recommend moving as much of this logic as possible to on-chain code. Additionally, the analysts recommend clearly documenting areas of the protocol that can be influenced by these off-chain components. For example, share prices of the hub contracts are not directly updated by changes to the balance of the pool contracts but instead by a privileged actor controlled by off-chain code.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Sprinter Liquidity Pool	`d98bb99`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 04–Mar. 07, 2025	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	1	0
Warning-Severity Issues	4	4	2
Informational-Severity Issues	1	1	1
TOTAL	7	7	3

Table 2.4: Category Breakdown.

Name	Number
Data Validation	2
Logic Error	2
Access Control	2
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Sprinter Liquidity Pool's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Can an attacker steal funds from the liqudity pool?
- Can the MPC signatures be misused?
- Are all the variables validated before use?
- Are all the privileged functions protected by adequate access control?
- Is the interaction with CCTP correct?
- Are the source and destination being correctly validated when transferring tokens cross-chain?
- Is the interaction with AAVE Pool correct?
- Is the protocol vulnerable to front running attacks?
- Is the protocol vulnerable to re-entrancy attacks?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard. This tool is designed to find instances of common smart contract vulnerabilities, such as reentrancy, unchecked return variables, and uninitialized variables.

Scope. The scope of this security assessment is limited to the following files provided by the Sprinter Liquidity Pool developers, which contains the smart contract implementation of the Sprinter Liquidity Pool.

- `Rebalancer.sol`
- `LiquidtyPool.sol`
- `LiquidityPoolAave.sol`

The protocol interacts with out of scope off-chain components. The auditors did not make any assumptions about the off-chain components. The developers provided the auditors an informal specification of the off chain components. The Veridise team assumes that these components are implemented correctly. The contracts in scope also interact with other contracts and third party protocols that are not in scope. The Veridise team assumes that these contracts are implemented correctly and do not make any assumptions about these protocols. The analysts

only verify the external calls made to these third party protocols based on their publicly available documentation.

Methodology. Veridise security analysts reviewed the reports of previous audits for Sprinter Liquidity Pool, inspected the provided tests, and read the Sprinter Liquidity Pool documentation. They then began a review of the code assisted by both static analyzers and automated testing.

During the security assessment, the Veridise security analysts regularly interacted with the Sprinter Liquidity Pool developers to ask questions about the code. The Veridise security analysts also referred to test cases in `tests/` directory to understand the intended behavior of contracts.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SLP-VUL-001	swapInputData not part of signature	High	Acknowledged
V-SLP-VUL-002	Withdrawals can possibly use stale . . .	Low	Acknowledged
V-SLP-VUL-003	Insufficient access control	Warning	Acknowledged
V-SLP-VUL-004	Missing validations	Warning	Fixed
V-SLP-VUL-005	Repay may pay more than the borrowed . . .	Warning	Fixed
V-SLP-VUL-006	Centralization Risk	Warning	Acknowledged
V-SLP-VUL-007	Code quality	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-SLP-VUL-001: swapInputData not part of signature

Severity	High	Commit	d98bb99
Type	Data Validation	Status	Acknowledged
File(s)	LiquidityPool.sol		
Location(s)	borrowAndSwap()		
Confirmed Fix At	9f5220c		

The protocol allows cross chain transfers by borrowing on a destination chain. On the destination chain, users can swap their tokens with a single call by calling `borrowAndSwap()` in the `LiquidityPool` contract as shown below. This function first checks the MPC signature and borrows the `borrowToken` to the address of `msg.sender`, then calls the function `swap()` on `msg.sender` and finally transfers in the `swapInputData.fillToken` and approves the original target to get the `fillAmount` of `fillTokens`.

However, `swapInputData` is not the part of the `signature`. This makes the function vulnerable to an attacker that has access to the `signature`

```
function borrowAndSwap(
    address borrowToken,
    uint256 amount,
    SwapParams calldata swapInputData,
    address target,
    bytes calldata targetCallData,
    uint256 nonce,
    uint256 deadline,
    bytes calldata signature
) external override whenNotPaused() {
    _validateMPCSignatureWithCaller(borrowToken, amount, target, targetCallData,
    nonce, deadline, signature);
    _borrow(borrowToken, amount, msg.sender);
    // Call the swap function on caller
    IBorrower(msg.sender).swap(swapInputData.swapData);
    IERC20(swapInputData.fillToken).safeTransferFrom(msg.sender, address(this),
    swapInputData.fillAmount);
    IERC20(swapInputData.fillToken).forceApprove(target, swapInputData.fillAmount);
    // - Invoke the recipient's address with calldata provided in the MPC signature
    to complete
    // the operation securely.
    (bool success,) = target.call(targetCallData);
    require(success, TargetCallFailed());
}
```

Snippet 4.1: Snippet from `borrowAndSwap()` in `LiquidityPool.sol`

Impact Consider the following attack scenario

1. The user who possesses the signature calls `borrowAndSwap()` with all arguments same as those signed except the `swapInputData`

2. For the `swapInputData` argument, the user provides `fillToken` that are tokens of lesser value and sets the `fillAmount` to the minimum amount of approval allowed by `fillToken`
3. The attack will be successful if the `target.call(targetCallData)` does not revert.

Effectively the security of the protocol becomes the responsibility of the function called by `target.call(targetCallData)`.

Recommendation The `swapInputData` should be part of the signature.

Developer Response The developers acknowledged the issue and informed us that the off chain MPC signer generates signatures only containing targets for which this issue does not exists. They have also added comments to both borrow functions to specify the intended behavior.

4.1.2 V-SLP-VUL-002: Withdrawals can possibly use stale conversion rate

Severity	Low	**Commit**	d98bb99
Type	Logic Error	**Status**	Acknowledged
File(s)	`LiquidityPool.sol`		
Location(s)	_withdrawProfitLogic()		
Confirmed Fix At	N/A		

The `LiquidityPool` contract manages the tokens on each of the supported chains. This contract is also responsible for collecting the profits earned by the protocol as shown below.

```
function withdrawProfit(
    address[] calldata tokens,
    address to
) external override onlyRole(WITHDRAW_PROFIT_ROLE) whenNotPaused() {
    _withdrawProfit(tokens, to);
}
```

Snippet 4.2: Snippet from `withdrawProfit()` in `LiquidityPool.sol`

The profits are collected when an address with `WITHDRAW_PROFIT_ROLE` calls the `withdrawProfit()` external function. The collected profits are distributed among the liquidity providers via the `LiquidityHub` contract.

The `LiquidityHub` contract implements ERC4626 standard that mints shares. According to the standard, the conversion rate between shares and assets depends on the total number of assets in the vault and the total shares minted. The `LiquidityHub` contract maintains the total assets in a storage variable `totalAssets`. The profits are added to `LiquidityHub` by calling the `adjustTotalAssets` function.

The logic to call the `withdrawProfit` and adding it to `LiquidityHub` is off chain.

Impact The liquidity providers may receive their profits based on a stale conversion rate if they withdraw their liquidity before the profits are added to the `LiquidityHub`. Since the logic is off chain it is not possible to specify the complete impact.

Recommendation Document the schedule for the transfer of the profits to `LiquidityHub` so users can plan their unstaking

Developer Response The developers responded

> Due to the multichain nature of the project, the conversion rate is always stale. We will consider having a schedule for updates, though initially we are planning to only adjust conversion rate when meaningful amount of profit is made.

4.1.3 V-SLP-VUL-003: Insufficient access control

Severity	Warning	**Commit**	d98bb99
Type	Access Control	**Status**	Acknowledged
File(s)	`LiquidityPool.sol`		
Location(s)	depositWithPool.sol		
Confirmed Fix At	N/A		

`LiquidityPool.depositWithPull` The `LiquidityPool` contract features a `depositWithPull` function that operates similarly to the standard `deposit` function for adding assets to the pool. The key distinction is that `depositWithPull` first withdraws assets from the caller using the ERC-20 `safeTransferFrom` function. According to developer discussions, this function is intended for admin use to distribute earned profits to the pool. However, the function lacks access controls, creating a risk that end users who call it by mistake would permanently lose their tokens with no recovery method.

```
function depositWithPull(uint256 amount) external override {
        // pulls USDC from the sender
        ASSETS.safeTransferFrom(msg.sender, address(this), amount);
        _deposit(msg.sender, amount);
}
```

Snippet 4.3: Snippet from `LiquidityPool.depositWithPull()`

`LiquidityPoolAave.repay` In the `LiquidityPoolAave` contract there is a `repay` function. This function is used in order to repay outstanding debts on the underlying Aave pool. However, there is no access control on this function, meaning that anybody can trigger a repayment, spending the pool's tokens.

```
function repay(address[] calldata borrowTokens) external override {
    // Repay token to aave
    bool success;
    for (uint256 i = 0; i < borrowTokens.length; i++) {
        success = _repay(borrowTokens[i]) || success;
    }
    require(success, NothingToRepay());
}
```

Snippet 4.4: Snippet from `LiquidityPoolAave.repay()`

Impact The impacts are detailed above.

Recommendation It is recommended to add access-control to the above functions.

Developer Response The developers have noted the following about the issue:

1. Users are not expected to interact with the contract without provided UI. Otherwise they should be competent enough to not send their funds wherever. Besides, contract should allow donations.
2. Repay will be executed often, and skipping the access control makes it cheaper to execute while not posing additional risk as the caller cannot affect the repayment process, they could only supply gas for it.

4.1.4 V-SLP-VUL-004: Missing validations

Severity	Warning	**Commit**	type commit hash here
Type	Data Validation	**Status**	Fixed
File(s)	See Description		
Location(s)	See description		
Confirmed Fix At	2d34795		

The code does not validate a few arguments as shown below

1. `amount` is not validated to be less than or equal to `IERC20(borrowToken).balanceOf(this)` in the function `LiquidityPool._borrow(...)`. This will lead to revert in `forceApprove(...)`, which can lead to confusion for the users
2. Non-zero check for `defaultLTV_` is missing in `LiquidityPoolAave.setDefaultLTV(...)` and `constructor(...)`. This check is important as `defaultLTV` is used when `ltv` is zero in `_checkLTV(...)`
3. Non-zero check is missing for `to` address in `withdrawProfit(...)`. A zero address accidentally passed to `to` argument may lead to tokens being burnt
4. The `ASSETS` balance of token of `LiquidityPoolAave` is not checked to be greater than or equal to amount in the withdraw function. This might lead to confusion as the `withdraw` will fail in `ASSETS.safeTransfer(...)`
5. The return value of `ILiquidityPool(sourcePool).withdraw(address(this), amount)` is not checked in `Rebalancer.initiateRebalance(...)`.

Impact There is no security impact from these missing checks, however they may lead to confusion when interacting with the codebase.

Recommendation It is recommended to implement correct error handling in order to maintain a cleaner and more user-friendly codebase.

Developer Response The developers have taken the following actions for each listed argument:

1. They have noted that UX improvement logic is removed in favor of gas savings for the `_borrow(...)` function
2. This is intended behavior to allow setting the `defaultLTV` to zero in order to only allow borrowing of whitelisted assets.
3. They have added a non-zero check for the `to` address in `withdrawProfit(...)`
4. They have added a check to the token balance of the pool in `_withdrawLogic(...)`
5. They have updated the signature of the `withdraw(...)` function to no longer return a value. The function will now always withdraw the specified amount.

4.1.5 V-SLP-VUL-005: Repay may pay more than the borrowed amount

Severity	Warning	**Commit**	d98bb99
Type	Logic Error	**Status**	Fixed
File(s)	`LiquidityPoolAave.sol`		
Location(s)	_repay()		
Confirmed Fix At	145906e		

The `repay()` function in `LiquidityPoolAave` takes a list of token addresses i.e. `borrowTokens`, and repays these tokens to `AAVE_POOL`. The function calls `_repay()` for each token in `borrowTokens`.

```
function _repay(address borrowToken)
    internal
    returns(bool success)
{
    AaveDataTypes.ReserveData memory borrowTokenData = AAVE_POOL.getReserveData(
    borrowToken);
    if (borrowTokenData.variableDebtTokenAddress == address(0)) return false;
    uint256 totalBorrowed = IERC20(borrowTokenData.variableDebtTokenAddress).
    balanceOf(address(this));
    if (totalBorrowed == 0) return false;
    uint256 borrowTokenBalance = IERC20(borrowToken).balanceOf(address(this));
    if (borrowTokenBalance == 0) return false;
    IERC20(borrowToken).forceApprove(address(AAVE_POOL), borrowTokenBalance);
    uint256 repaidAmount = AAVE_POOL.repay(
        borrowToken,
        borrowTokenBalance,
        2,
        address(this)
    );
    emit Repaid(borrowToken, repaidAmount);
    return true;
}
```

Snippet 4.5: Snippet from `example()`

The `_repay()` function call `AAVE_POOL.repay()` with all of the `borrowToken` balance of `LiquidityPoolAave` contract.

Impact This issue does not have an impact because the AAVE pool only transfers in the amount of debt that it is owed. However this places trust in an external protocol to not withdraw more funds. In order to follow best practices, The protocol should approve only the minimum amount of tokens required to repay the tokens owed to AAVE pool.

Recommendation Call `AAVE_POOL.repay()` with minimum of `borrowTokenBalance` and `totalBorrowed`

Developer Response The developers have changed the logic to use the minimum of `borrowTokenBalance` and `totalBorrowed` when approving funds to the Aave pool.

4.1.6 V-SLP-VUL-006: Centralization Risk

Severity	Warning	**Commit**	d98bb99
Type	Access Control	**Status**	Acknowledged
File(s)	See Description		
Location(s)	See description		
Confirmed Fix At	N/A		

Similar to many projects, Sygma's, `LiquidityPool`, `LiquidityPoolAave` and `Rebalancer` declare an administrator role that is given special permissions. Along with the admin role these contracts also declare multiple roles that can perform privileged actions. In particular, these administrators are given the following abilities:

- The `REBALANCER_ROLE` can trigger rebalances
- The `LIQUIDITY_ADMIN_ROLE` can withdraw tokens from the `LiquidityPool`
- The `WITHDRAW_PROFIT_ROLE` can trigger withdrawing profits from the `LiquidityPool` and controls the address to which the profits are transferred.
- The `DEFAULT_ADMIN_ROLE` can change the address of the MPC signer and other important parameters like
  - the minimum health factor
  - the LTV of a token
  - and default LTV

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, an attacker could withdraw all the funds in the liquidity pool.

Recommendation Privileged operations should be operated by a multi-sig contract or decentralized governance system. Non-emergency privileged operations should be guarded by a timelock to ensure there is enough time for incident response. The risks in this issue may be partially mitigated by validating that the protocol is deployed with the appropriate roles granted to timelock and multi-sig contracts.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

1. Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
2. Using separate keys for each separate function.
3. Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
4. Enabling 2FA for key management accounts. SMS should **not** be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
5. Validating that no party has control over multiple multi-sig keys.
6. Performing regularly scheduled key rotations for high-frequency operations.
7. Securely storing physical, non-digital backups for critical keys.

8. Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
9. Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

Developer Response The developers have responded with the following about the various roles in the project:

- `REBALANCER_ROLE` if compromised must only be able to move liquidity to another valid pool. If compromised, the funds could be rebalanced back after rotating the key.
- `LIQUIDITY_ADMIN_ROLE` is only given to the LiquidityHub on the main network, and to Rebalancer on all networks, so liquidity movement is controlled by other code.
- `WITHDRAW_PROFIT_ROLE` risks only the profit made since latest withdrawal, so impact is very limited and cannot put system into debt.
- `DEFAULT_ADMIN_ROLE` can rekt the protocol and will be controlled by a multisig as the most secure actor.

4.1.7 V-SLP-VUL-007: Code quality

Severity	Info	**Commit**	d98bb99
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	879e20a		

A number of small suggestions for better code quality:

- The `BorrowingIsNotPaused` error in `LiquidityPool.sol` is never used.
- The `setMinHealthFactor` would be a more concise name for the `setHealthFactor` function in `LiquitidyPoolAave.sol`
- The `mpcAddress` assignment on line 73 of `LiquidityPoolAave.sol` is redundant with the assignment on line 83 of `LiquidityPool.sol`

Impact There is no security impact, however it is recommended to address these issues in order to maintain a cleaner codebase.

Developer Response The developers have taken the following actions to address the issue:

- They have removed the `BorrowingIsNotPaused` error
- They have renamed the `setHealthFactor` function to `setMinHealthFactor`
- They have removed the assignment to `mpcAddress` in the constructor of `LiquidityPoolAave.sol`

Glossary

AAVE Aave is an Open Source Protocol to create Non-Custodial Liquidity Markets to earn interest on supplying and borrowing assets. To learn more, visit https://aave.com. 1

CCTP A cross-chain transfer protocol, allowing tokens to be moved from a source chain to a destination chain. See https://developers.circle.com/stablecoins/cctp-getting-started for more . 4

reentrancy A vulnerability in which a smart contract hands off control flow to an unknown party while in an intermediate state, allowing the external party to take advantage of the situation. 4

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 17

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See https://docs.soliditylang.org/en/v0.8.19/ to learn more. 17

Vanguard A static analysis tool for Solidity by Veridise. See https://docs.veridise.com/vanguard/ for more information . 4