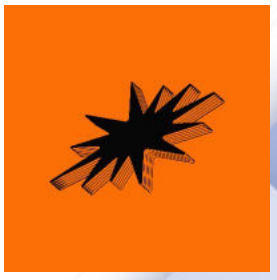
Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Sprinter Liquidity Mining

Veridise Inc.

February 25, 2025

► **Prepared For:**

Sygma-Labs
https://buildwithsygma.com

► **Prepared By:**

Ajinkya Rajput
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Feb. 25, 2025	V2
Feb. 24, 2025	V1
Feb. 19, 2025	Initial Draft

Contents

1 Executive Summary

From Feb. 12, 2025 to Feb. 17, 2025, Sygma-Labs engaged Veridise to conduct a security assessment of their Sprinter Liquidity Mining protocol. The security assessment covered contracts that implemented Liquid Staking where users can deposit tokens for a specified period of time and get LP tokens in return. Veridise conducted the assessment over 8 person-days, with 2 security analysts reviewing the project over 4 days on commit `a7b0c26`. 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 Mining protocol implements contracts that allow users to provide liquidity to solvers and earn rewards. Users receive two tokens as rewards i.e. LPTokens and ScoreTokens. Users can lock in their stake for different time periods depending on the tier they choose. After the staking period is over, the users can unstake and receive their LPTokens. LPTokens are standard ERC4626 share tokens and accrue profits over time and can be redeemed to receive the assets. ScoreTokens are minted to users on staking and are used to calculate a different reward based on total stake the user has placed over all the transactions.

Code Assessment. The Sprinter Liquidity Mining developers provided the source code of the Sprinter Liquidity Mining contracts for the code review. The source code mostly inherits from OpenZeppelin's ERC-20, ERC-2612 and ERC-4626 contracts. The derived contracts override a few functions and add some functions so that these contracts can interact with other sub-components. The overrides and additional functions added to the derived contracts appear to be mostly original code written by Sygma-Labs 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 Mining 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 4 warnings, and 2 informational findings. The Sprinter Liquidity Mining developers have acknowledged and fixed the issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Sprinter Liquidity Mining protocol. The implementation tries to extend the OpenZeppelin contracts and re-implements some contracts. The design could be simplified by using the features provided by the base contracts and only adding necessary functionality.

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 Mining	`a7b0c26`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 12–Feb. 17, 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	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	1	1
Warning-Severity Issues	4	4	3
Informational-Severity Issues	2	2	2
TOTAL	7	7	6

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Data Validation	1
Missing/Incorrect Events	1
Logic Error	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Do the overridden functions return the correct values?
- Do the token contracts use IERC20Permit correctly?
- Do the derived contracts violate any invariants assumed by the base contracts?
- Are all the variables validated before use?

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, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy, unchecked return variables, and uninitialized variables.
- *Fuzzing/Property-based Testing.* Security analysts leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, the desired behavior of the protocol was formulated as [V] specifications and then tested using Veridise's OrCa fuzzing framework to determine if a violation of the specification can be found.

Scope. The scope of this security assessment is limited to following the `contracts/` folder of the source code provided by the Sprinter Liquidity Mining developers, which contains the smart contract implementation of the Sprinter Liquidity Mining.

During the security assessment, the Veridise security analysts referred to the excluded files but assumed that they have been implemented correctly.

- `Rebalancer.sol`

The contracts in scope also interact with other contracts that are not in scope. We assume that the contracts not in scope are implemented correctly.

Methodology. Veridise security analysts reviewed the reports of previous audits for Sprinter Liquidity Mining, inspected the provided tests, and read the Sprinter Liquidity Mining 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 Mining 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-SSL-VUL-001	Missing address validation	Low	Fixed
V-SSL-VUL-002	Lack of safety check causes generic revert . . .	Warning	Fixed
V-SSL-VUL-003	Transfer disabled in derived contract	Warning	Fixed
V-SSL-VUL-004	Wrong amount passed to StakeLockedEvent	Warning	Fixed
V-SSL-VUL-005	ScoreTokens cannot be burnt	Warning	Acknowledged
V-SSL-VUL-006	Unused program construct	Info	Fixed
V-SSL-VUL-007	Unclear function naming	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-SSL-VUL-001: Missing address validation

Severity	Low	**Commit**	a7b0c26
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	4504252		

There are a number of addresses that are used by contracts in the protocol without any input validation.

`ManagedToken` In the constructor of the `ManagedToken` contract, a `manager` address is taken as input from the user. This input is used to set the `MANAGER` field without any validation. If that field were to be set to zero, it would render the contract unusable.

```
constructor(string memory name_, string memory symbol_, address manager)
    ERC20(name_, symbol_)
    ERC20Permit(name_)
{
    MANAGER = manager; // No sanitization on manager input
}
```

Snippet 4.1: Snippet from `ManagedToken` constructor

`SprinterLiquidityMining` In the constructor of the `SprinterLiquidityMining`, an address to a `ILiquidityHub` is taken as input from the user. This input is used without validation to set the `LIQUIDITY_HUB` field of the contract. Without any sanitization, this address could be the zero address.

```
constructor(address owner_, address liquidityHub, Tier[] memory tiers_)
  LiquidityMining(
      "Sprinter USDC LP Score",
      "sprUSDC-LP-Score",
      owner_,
      address(ILiquidityHub(liquidityHub).SHARES()),
      tiers_
  )
{
  LIQUIDITY_HUB = ILiquidityHub(liquidityHub); // liquidityHub is not validated
}
```

Snippet 4.2: Snippet from `SprinterLiquidityMining` constructor

`LiquidityHub` In the `initialize` function of `LiquidityHub`, special roles are granted to input `admin` and `adjuster` accounts. However, these addresses are never validated.

```
function initialize(
    IERC20 asset_,
    address admin,
    address adjuster,
    uint256 newAssetsLimit
) external initializer() {
    ERC4626Upgradeable.__ERC4626_init(asset_);
    require(
        IERC20Metadata(address(asset_)).decimals() <= IERC20Metadata(address(SHARES))
    .decimals(),
        IncompatibleAssetsAndShares()
    );
    // Deliberately not initializing ERC20Upgradable because its
    // functionality is delegated to SHARES.
    _grantRole(DEFAULT_ADMIN_ROLE, admin); // No validation
    _grantRole(ASSETS_ADJUST_ROLE, adjuster); // No validation
    _setAssetsLimit(newAssetsLimit);
}
```

Snippet 4.3: Snippet from `LiquidityHub` initialization function

Impact The impacts of the issue are detailed above

Recommendation Include some form of input sanitization in the constructor of these contracts.

Developer Response The developers have added necessary checks to `ManagedToken` and `LiquidityHub`. For `SprinterLiquidityMining`, they have explained that a null address `liquidityHub` contract will revert on the call to `SHARES`, which we agree with.

4.1.2 V-SSL-VUL-002: Lack of safety check causes generic revert messages

Severity	Warning	**Commit**	a7b0c26
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	7b281a6		

It is desirable to ensure that the protocol gives users useful error messages whenever a transaction reverts. In that regard, there are several locations in the project where operations may revert without offering a good explanation to the user.

`LiquidityMining.sol` In the `LiquidityMining._stake(...)` function, an array is read without any bounds check on the index. Out-of-bounds memory accesses revert in Solidity so there is no danger, however the message associated with the revert will not be meaningful in the context of the project.

```
function _stake(address from, address scoreTo, uint256 amount, uint256 tierId)
    internal {
    require(amount > 0, ZeroAmount());
    require(miningAllowed, MiningDisabled());
    Stake memory currentStake;
    Tier memory tier = tiers[tierId]; // No check to see if tierId is in bounds
    // ...
}
```

Snippet 4.4: Snippet from the`LiquidityMining._stake(...)` function

In the `LiquidityMining.unstake(...)` function, funds are transferred from the contract to an input address after unstaking them. The issue is that the balance of the contract is not checked, which could lead to a generic error message if attempting to transfer more funds than the contract owns.

```
function unstake(uint256 id, address to) external {
  uint256 amount = _unstake(_msgSender(), id, to);
  STAKING_TOKEN.safeTransfer(to, amount);
}
```

Snippet 4.5: Snippet from the `LiquidityMining.unstake(...)` function

Impact There is no risk introduced by the issue, however the lack of proper error messages makes the contracts less usable

Recommendation It is recommended to add manual checks where reverts are possible and to emit an informative error to the user in case of a problematic input.

Developer Response The developers have added a safety check to the `_stake(...)` function. For `unstake(...)`, they have acknowledged the issue and provided sound reasoning for why a check is not required.

4.1.3 V-SSL-VUL-003: Transfer disabled in derived contract

Severity	Warning	**Commit**	a7b0c26
Type	Maintainability	**Status**	Fixed
File(s)	`LiquidityMining.sol`		
Location(s)	transfer()		
Confirmed Fix At	4504252		

The `LiquidityMining` contract is derived from OpenZeppelins ERC20 contract and adds additional functions like `depositAndStake()` and `unstakeAndWithdraw()` that interact with `LiquidityHub` among other things. On every stake this contract mints tokens that are referred in this issue as `scoreTokens`. The `scoreTokens` represent the total stake a user has submitted in all their interactions with the contract. This happens in the `_stake()` internal function as shown below.

Also, the `scoreTokens` are not burnt while unstaking.

```
function _stake(address from, address scoreTo, uint256 amount, uint256 tierId)
    internal {
    require(amount > 0, ZeroAmount());
    require(miningAllowed, MiningDisabled());
    Stake memory currentStake;
    Tier memory tier = tiers[tierId];
    currentStake.amount = amount;
    currentStake.period = tier.period;
    currentStake.until = timeNow() + tier.period;
    currentStake.multiplier = tier.multiplier;
    stakes[from].push(currentStake);
    uint256 addedScore =
        currentStake.amount * uint256(tier.multiplier) /
        uint256(MULTIPLIER_PRECISION);
    _mint(scoreTo, addedScore);

    emit StakeLocked(from, scoreTo, amount, amount, currentStake.until, addedScore);
}
```

Snippet 4.6: Snippet from `_stake()` from `LiquidityMining.sol`

The `SprinterLiquidityMining` contract derives from the `LiquidityMining` contract described above and disables the transfers of `scoreTokens` by reverting on `transfer()`, `approve()`, and `transferFrom()`. And overriding the `_update()` function to allow for minting only. Disabling of transfers is important for the intended behavior of the `scoreToken`.

However, the code for minting is present in the base contract i.e. `LiquidityMining` and the code for disabling the transfers is implemented in the derived contract.

Impact It may happen that some other contract may want to implement functionality similar to `SprinterLiquidityMining` that derives from `LiquidityMining`. The responsibility of disabling transfers is with the derived contracts which is easy to miss. This may lead to maintainability issues.

Recommendation Disable the transfers in the base `LiquidityMining` contract itself or move the call to `_mint()` to the `SprinterLiquidity` contract

Developer Response The developers have added a burn function in the base contract and disabled it in the derived `SprinterLiquidityMining` contract

4.1.4 V-SSL-VUL-004: Wrong amount passed to StakeLockedEvent

Severity	Warning	**Commit**	a7b0c26
Type	Missing/Incorrect Eve	**Status**	Fixed
File(s)	`LiquidityMining.sol`		
Location(s)			
Confirmed Fix At	4504252		

When users lock in their stake a `StakeLocked` event is emitted inside `_stake()` function. The `StakeLocked` event takes arguments as shown below

```
event StakeLocked(
    address from,
    address to,
    uint256 amount,
    uint256 totalAmount,
    uint32 until,
    uint256 addedScore
);
```

Snippet 4.7: Snippet from `StakeLockedEvent`

```
function _stake(address from, address scoreTo, uint256 amount, uint256 tierId)
    internal {
    require(amount > 0, ZeroAmount());
    require(miningAllowed, MiningDisabled());
    Stake memory currentStake;
    Tier memory tier = tiers[tierId];
    currentStake.amount = amount;
    currentStake.period = tier.period;
    currentStake.until = timeNow() + tier.period;
    currentStake.multiplier = tier.multiplier;
    stakes[from].push(currentStake);
    uint256 addedScore =
        currentStake.amount * uint256(tier.multiplier) /
        uint256(MULTIPLIER_PRECISION);
    _mint(scoreTo, addedScore);

    emit StakeLocked(from, scoreTo, amount, amount, currentStake.until, addedScore);
}
```

Snippet 4.8: Snippet from `_stake()` from LiquidityMining.sol

The fourth argument is named as `totalAmount`. However the same amount argument is passed to third and fourth argument.

Impact Wrong value is passed to the `StakeLocked` event

Recommendation Pass `balanceOf(from)` as the fourth argument.

Developer Response The developers have removed the `totalAmount` field from the `StakeLocked` event.

4.1.5 V-SSL-VUL-005: ScoreTokens cannot be burnt

Severity	Warning	**Commit**	a7b0c26
Type	Logic Error	**Status**	Acknowledged
File(s)	`LiquidityMining.sol`		
Location(s)	_stake()		
Confirmed Fix At	N/A		

When users lock in their stakes, they receive ERC20 tokens that we refer as `ScoreTokens`. These tokens are minted in the `_stake()` function in LiquidityMining.sol as shown below.

```
function _stake(address from, address scoreTo, uint256 amount, uint256 tierId)
    internal {
  // Start of the function omitted

  _mint(scoreTo, addedScore); // ScoreTokens minted

  emit StakeLocked(from, scoreTo, amount, amount, currentStake.until, addedScore);
}
```

Snippet 4.9: Snippet from `_stake()`

The developers informed the auditors that the `ScoreTokens` are used by the protocol to keep a track of the total stake placed by a users across all their transactions. The balances of these tokens may later be used to distribute an additional reward to users depending on the balance of `ScoreToken` for that user.

However there is currently no way to burn these tokens

Impact If the protocol utilizes balance of `ScoreToken` to distribute rewards, and multiple such rounds of reward distribution occur, there would be a few users that would receive the reward multiple times for the same stake

Recommendation Implement a protected function that allows burning `ScoreTokens`

Developer Response The developers acknowledged the issue but explained that the off chain logic for reward distribution will take care of reward calculation and that the reward distribution will happen only once

4.1.6 V-SSL-VUL-006: Unused program construct

Severity	Info	**Commit**	a7b0c26
Type	Maintainability	**Status**	Fixed
File(s)	`LiquidityMining.sol`		
Location(s)			
Confirmed Fix At	7b281a6		

The `InvalidAddedScore` error is never used by the project.

Impact There is no security impact, however it is recommended to remove unused program constructs to maintain a cleaner codebase.

Developer Response The developers have removed the unused construct.

4.1.7 V-SSL-VUL-007: Unclear function naming

Severity	Info	**Commit**	a7b0c26
Type	Maintainability	**Status**	Fixed
File(s)	`LiquidityHub.sol`		
Location(s)	_getTotals		
Confirmed Fix At	4504252		

The `LiquidityHub` contract includes a function named `_getTotals(...)`. Based on the name, one might expect this to retrieve some information from the contract state. However, this function is actually responsible for clipping the lower bound on the input values `supplyShares` and `supplyAssets`

Impact There is no security impact, however we believe it is important to maintain a clear codebase for the maintainability of the project.

Recommendation Change the name of the function to more accurately reflect its functionality

Developer Response The developers have refactored the function to be clearer.

Glossary

[V] A declarative specification language used to encode properties when testing with Veridise tools. See https://docs.veridise.com/orca/user_guide/v/overview for more information . 4

ERC Ethereum Request for Comment. 17

ERC-20 The famous Ethereum fungible token standard. See https://eips.ethereum.org/EIPS/eip-20 to learn more. 1, 17

ERC-2612 An Ethereum Request for Comment (ERC) describing a permit extension for ERC-20-signed approvals. See https://eips.ethereum.org/EIPS/eip-2612 for the full ERC. 1

ERC-4626 An Ethereum Request for Comment (ERC) describing a tokenized vault representing shares of an underlying ERC20 token. See https://eips.ethereum.org/EIPS/eip-4626 for the full ERC. 1

Ethereum Request for Comment Peer-reviewed proposals describing application-level standards and conventions. Visit https://eips.ethereum.org/erc to learn more. 17

Liquid Staking A tokenized contract representing shares of a staked native currency. See https://chain.link/education-hub/liquid-staking to learn more. 1

OpenZeppelin A security company which provides many standard implementations of common contract specifications. See https://www.openzeppelin.com. 1

OrCa An oracle-guided contract fuzzer by Veridise. See https://docs.veridise.com/orca/ for more information . 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

Slither A static analyzer for Solidity by Crytic, a subsidiary of Trail of Bits. See https://github.com/crytic/slither for more information. 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