Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Lumina

Lumina Dex

Veridise Inc.
January 19, 2025

► **Prepared For:**

Lumina
https://luminadex.com/

► **Prepared By:**

Tyler Diamond
Ben Sepanski

► **Contact Us:**

contact@veridise.com

► **Version History:**

Jan. 19, 2025	V3 - Incorporating issue responses/fixes
Dec. 24, 2024	V2 - Incorporating farm contracts
Dec. 11, 2024	V1
Dec. 09, 2024	Initial Draft

Contents

1 Executive Summary

From Dec. 2, 2024 to Dec. 6, 2024 and Dec. 19, 2024 to Dec. 23, 2024, Lumina engaged Veridise to conduct a security assessment of their Lumina Dex. The first security assessment covered the implementation of a constant product AMM and the factory contract that deploys DEX pools for tokens. Veridise conducted the assessment over 2 person-weeks, with 2 security analysts reviewing the project over 1 week on commit `ad04eba`[*]. The second security assessment covered farming contracts designed to incentivize liquidity providers to fund pools in return for additional rewards. Veridise performed this assessment over 6 person-days, with the same 2 security analysts reviewing the project over 3 days on commit `057cd8d`[†]. The review strategy involved a thorough code review in addition to internals of the o1js used.

Project Summary. The first security assessment covered the o1js implementation of a constant-product AMM. There were 3 smart contracts involved in the scope of the review. The `Pool` and `PoolTokenHolder` contracts implement the details of swapping tokens in addition to providing and removing liquidity to the pool. Lastly, the `PoolFactory` contract implements the logic for deploying the aforementioned contracts and the permissions that pools are comprised of, in addition to verifying whether the caller has the right to deploy a given pool.

Unlike synchronous blockchain execution environments, o1js code executes entirely asynchronously, and then is uploaded as `AccountUpdates` to the Mina blockchain with proof of correct execution. In order to maintain the core AMM invariants during swaps, users commit locally to a number of input tokens, a number of output tokens, and the amounts used for the pool balances. At transaction execution time, the network checks these amounts against inequality constraints designed to ensure that the product of the token balances does not decrease, maintaining solvency for the pool. A similar approach is taken for deposits and withdrawals.

Note that this has the beneficial side effect of committing to slippage at transaction creation-time, forcing users to exactly specify and commit to the amount of slippage their transaction will experience if successful. See related issue V-LUM-VUL-010 for more details on other consequences of this design.

The second security assessment reviewed four farming contracts. The first two contracts, `Farm` and `FarmTokenHolder`, allows users to deposit `Pool` liquidity tokens for non-transferable `Farm` tokens at a 1-1 exchange rate. The `Pool` tokens may only be deposited in an owner-specified time period, but the `Farm` tokens may be redeemed at any point to reclaim the `Pool` liquidity tokens.

The second two contracts, `FarmReward` and `FarmRewardTokenHolder`, distribute rewards (in either MINA or a fungible token). The funds for these rewards must be provided by a centralized entity. The deployer is responsible for determining which accounts receive rewards and how large their rewards are. See V-LUM-VUL-005 for more details.

[*] https://github.com/Lumina-DEX/lumina-v1-core/ad04eba
[†] https://github.com/Lumina-DEX/lumina-v1-core/057cd8d

Code Assessment. The Lumina Dex developers provided the source code of the Lumina Dex contracts for the code review. The source code appears to be mostly original code written by the developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. However, the intentions of the functionality of the smart contracts are not documented and the Veridise analysts relied on the source code for their understanding.

The source code contained a test suite, which the Veridise security analysts noted contained tests for the deployment and ownership logic. However, the results of swaps and liquidity provisioning were not tested to be valid.

Summary of Issues Detected. The security assessment uncovered 15 issues, 2 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-LUM-VUL-001 details that liquidity providers in a pool do not receive their corresponding shares and V-LUM-VUL-002 shows that tokens are limited in the number of pools they may be involved in. The Veridise analysts also identified 2 medium-severity issues, including V-LUM-VUL-004 describing that signatures on pool creation are not pinned to any specific tokens, as well as 2 low-severity issues, 4 warnings, and 5 informational findings. Currently, Lumina has fixed 11 issues, including all issues of high or critical severity. Additionally, Lumina provided several mitigations to the centralization risks of the protocol described in (V-LUM-VUL-006). Note that some risks related to upgradeability are inherent to the Mina network, as an owner must be trusted to correctly upgrade a contract during a hard fork.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Lumina Dex.

Careful event handling. Most likely, farmings rewards will be determined based on information provided to an off-chain entity listening for on-chain events. Care should be taken to properly handle edge cases such as burning or minting zero amounts, users who leave their `Pool` tokens in a `Farm` beyond the end of the farming period, or users who intentionally burn `Farm` tokens without redeeming their `Pool` tokens (see related issues V-LUM-VUL-003 and V-LUM-VUL-009). Further, frontend developers should note that some events have dual purposes (e.g. a `ClaimEvent` may be emitted when someone claims rewards *or* when the contract owner withdraws reward funds) and ensure to handle all cases properly.

In general, it is a common o1js design pattern to break up a single workflow into several `@method` invocations. Since the invoked `@methods` can be called directly by the user instead of by the intended caller contract, off-chain listeners must carefully analyze the code before assuming any two events will always be emitted together.

Additional testing. Issues like V-LUM-VUL-001 may be caught quickly using testing. In general, all workflows (swapping, depositing, and withdrawing) should be tested in both happy and unhappy paths in a variety of scenarios. This should include non-trivial sequences of deposits, withdrawals, and swaps, tests validating the AMM invariant holds, and tests validating the exact amounts input and output from each operation. In addition, these tests should be performed over each configuration of the pool, both for MINA pools and token-only pools.

Use provable initialization. Using provable initialization could bypass the need for many repeated checks by performing them only at initialization. For example, checking the validity of the pair of tokens is repeatedly performed in most pool functions. Furthermore, provable initialization would decrease the trust assumptions on the deployer, and reduce the need for manual deployment/permissions validation. See V-LUM-VUL-008 for more details.

Documentation. Not all functions provide `natspec` comments. Additionally, the protocol requirements and intended functionality are not explicitly documented. These should be documented for both end-user understanding and future maintainability of the project. Moreover, the contracts assume certain properties of the actors that they interact with. For example, the usage of `AccountUpdate.createSigned()` leads to the requirement that only EOAs can interact with the `Pool`. These assumptions should be avoided so that the protocol is more composable, or clearly documented for users of the project.

Function Naming. Due to the architecture of o1js, many operations are split across several contracts and functions. The developers should consider re-architecting function names to more narrowly specify their functionality. For example, to withdraw tokens, a user calls `PoolTokenHolder.withdrawLiquidityToken()`. This sends the user the `token0` they are owed, and calls `PoolTokenHolder.subWithdrawLiquidity()` on the token holder for `token1`. Next, `subWithdrawLiquidity()` sends the user the `token1` they are owed, and makes a final call to `Pool.checkLiquidityToken()`, which burns the user's liquidity tokens. Redefining the function names to clearly indicate what part of which high-level operation they implement (e.g. `withdrawLiquidityToken()`, `withdrawLiquidityToken1Only()`, and `burnLiquidityTokens()`) could improve readability.

Use the fungible token standard. Lastly, using the fungible token standard[‡] for the liquidity provider tokens will enable those tokens to have more composability with other protocols.

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.

[‡] `https://github.com/MinaFoundation/mina-fungible-token`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Lumina Dex(Pool)	`ad04eba`	o1js	Mina
Lumina Dex (Farm)	`057cd8d`	o1js	Mina

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Dec. 2–Dec. 6, 2024	Manual	2	2 person-weeks
Dec. 19–Dec. 23, 2024	Manual	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	1	1	1
Medium-Severity Issues	2	2	1
Low-Severity Issues	2	2	0
Warning-Severity Issues	4	4	3
Informational-Severity Issues	5	5	5
TOTAL	15	15	11

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Maintainability	3
Logic Error	2
Missing/Incorrect Events	2
Authorization	2
Access Control	1
Denial of Service	1
Gas Optimization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Is the math of swapping and liquidity provisioning correctly implemented and constrained?
- Is there any way for an attacker to steal funds?
- Are the permissions of the deployed contracts correctly configured?
- Can multiple pools be deployed for the same pair of tokens?
- Are `AccountUpdates` correctly generated and verified?
- Are common smart contract bugs like access control, arithmetic errors, or frontrunning issues present?
- Can an attacker cause denial of service issues for the protocol?
- Are common ZK bugs like under-constrained or over-constrained circuits present?
- Can anyone bypass the farm's replay protection by transferring the anti-replay tokens?
- Is it possible to redeem farm tokens for pool tokens without emitting a burn event on the depositor account?
- Can users redeem rewards more than once?
- Can users obtain reward funds without being included in the owner-specified Merkle tree?
- Can an attacker shut down or otherwise deny service to the farming contracts?
- Can user funds be locked in the pool or farm contracts?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a thorough review by human experts.

The scope of the first security assessment is limited to the `contracts/src/pool` folder of the source code provided by the Lumina Dex developers, which contains the smart contract implementation of the Lumina Dex. The scope of the second security assessment is limited to the `contracts/src/farming` folder of the same repository, which contains the implementation of the farming contracts.

Methodology. Veridise security analysts reviewed the reports of previous audits for Lumina Dex, inspected the provided tests, and read the Lumina Dex documentation. They then began a review of the code. During the security assessment, the Veridise security analysts communicated with the Lumina Dex developers asynchronously to ask questions about the code.

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-LUM-VUL-001	No liquidity tokens for deposits	Critical	Fixed
V-LUM-VUL-002	Only one pool can be deployed for a token0	High	Fixed
V-LUM-VUL-003	Events may emit incorrect output amounts	Medium	Fixed
V-LUM-VUL-004	Signatures are not unique to tokens	Medium	Acknowledged
V-LUM-VUL-005	Pool creation not as permissioned as intended	Low	Acknowledged
V-LUM-VUL-006	Centralization Risk	Low	Partially Fixed
V-LUM-VUL-007	Ownership transferred in one step	Warning	Fixed
V-LUM-VUL-008	No provable initialization	Warning	Fixed
V-LUM-VUL-009	Events should separate receiver/sender	Warning	Fixed
V-LUM-VUL-010	Pool actions may be DOSed for gas cost of . . .	Warning	Acknowledged
V-LUM-VUL-011	o1js/Token Best Practices	Info	Fixed
V-LUM-VUL-012	Typos and incorrect comments	Info	Fixed
V-LUM-VUL-013	[Upd.] Unused and duplicate program . . .	Info	Fixed
V-LUM-VUL-014	Optimizations	Info	Fixed
V-LUM-VUL-015	New pool event should emit signer	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-LUM-VUL-001: No liquidity tokens for deposits

Severity	Critical	**Commit**	ad04eba
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/src/pool/Pool.ts`		
Location(s)	supply()		
Confirmed Fix At	0d0bb58		

Depositing into the pool is implemented using the `Pool.supply()` function. The function covers four cases: MINA pools vs non-Mina pools (indicated by `isMinaPool`) and the first deposit vs non-first deposits (indicated by `isFirstSupply`).

As shown in the below snippet, the `!isFirstSupply` case declares a new variable named `liquidityAmount` instead of setting the `liquidityAmount` variable used to increase supply. Additionally, the branch does not set `liquidityUser`, which determines the number of liquidity tokens sent to the user.

```
let liquidityAmount = UInt64.zero;
let liquidityUser = UInt64.zero;

// [VERIDISE] Address preparation elided ...

if (isFirstSupply) {
   // [VERIDISE] First supply case elided ...
} else {
   // [VERIDISE] Computation of liquidityToken0/1 elided ...

   const liquidityAmount = Provable.if(liquidityToken0.lessThan(liquidityToken1),
    liquidityToken0, liquidityToken1);
   liquidityAmount.assertGreaterThan(UInt64.zero, "Liquidity out can't be zero");
}

// [VERIDISE] Code transferring token0/token1 elided ...

this.internal.mint({ address: sender, amount: liquidityUser });
this.internal.mint({ address: circulationUpdate, amount: liquidityAmount });
```

Snippet 4.1: Snippet from `supply()`

Impact After the first deposit into a pool, subsequent deposits will not mint tokens to the user or increase the total supply of liquidity tokens.

Recommendation Overwrite `liquidityAmount` and `liquidityUser` in the `else` branch. Add tests validating user balances after deposits/withdrawals in a wide variety of scenarios, including both MINA and non-MINA pools and both first and non-first deposits.

Developer Response The developers have implemented the recommendations.

4.1.2 V-LUM-VUL-002: Only one pool can be deployed for a token0

Severity	High	**Commit**	ad04eba
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/src/pool/PoolFactory.ts`		
Location(s)	createPoolToken()		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/1`		

As can be seen below, the `publicKey` created for two tokens as a unique identifier of the pool is done by concatenating their fields together:

```
const fields = token0.toFields().concat(token1.toFields());
const publicKey = PublicKey.fromFields(fields);
```

Snippet 4.2: Snippet from `contracts/src/pool/PoolFactory.ts:createPoolToken()`

However, this will just set `publicKey` to `token0`, as `CircuitValue.fromFields()` only reads in the amount of fields required for the underlying type.

Impact Any token can only be used as `token0` for a single pool, as other pairs will lead to the `tokenAccount` check inside of `createAccounts()` failing.

Recommendation Consider hashing the values together to create a unique identifier.

Developer Response The developers now hash the concatenation of the token's field representation and map the hash to a group point.

4.1.3 V-LUM-VUL-003: Events may emit incorrect output amounts

Severity	Medium	**Commit**	ad04eba
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	`contracts/src/pool/Pool.ts`		
Location(s)	checkLiquidityToken()		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/2, https://github.com/Lumina-DEX/lumina-v1-core/pull/12		

The `Pool.checkLiquidityToken()` is meant to be invoked as part of the token withdrawal mechanism. For token-token pools, a liquidity provider can withdraw funds by calling `PoolTokenHandler.withdrawLiquidityToken()` on `token0` to receive their `token0` share. That function then invokes `PoolTokenHandler.subWithdrawLiquidity()` on `token1` to send the user their `token1` share. Finally, `PoolTokenHandler.subWithdrawLiquidity()` invokes `Pool.checkLiquidityToken()` to burn the requisite number of liquidity tokens.

In order to invoke `checkLiquidityToken()`, it is defined as a public `@method` on the `Pool` contract (shown in the below code snippet). This `@method` emits an event reporting the amount of tokens withdrawn, but does not validate the amounts (other than ensuring they are non-zero).

```
@method async checkLiquidityToken(liquidityAmount: UInt64,
                                                                    amountToken0:
    UInt64,
                                                                    amountToken1:
    UInt64,
                                                                    reserveMinaMin:
    UInt64,
                                                                    supplyMax: UInt64
    ) {
    liquidityAmount.assertGreaterThan(UInt64.zero, "Liquidity amount can’t be zero");
    reserveMinaMin.assertGreaterThan(UInt64.zero, "Reserve mina min can’t be zero");
    amountToken0.assertGreaterThan(UInt64.zero, "Amount token can’t be zero");
    supplyMax.assertGreaterThan(UInt64.zero, "Supply max can’t be zero");

    // token 0 need to be empty on mina pool
    const [token0, token1] = this.checkToken(false);

        // [VERIDISE]: Code elided....
    // [VERIDISE]: Compute supplyMax and get sender

    // burn liquidity from user and current supply
    liquidityAccount.balanceChange = Int64.fromUnsigned(liquidityAmount).negV2();
    await this.internal.burn({ address: sender, amount: liquidityAmount });

    this.emitEvent("withdrawLiquidity", new WithdrawLiquidityEvent({
            sender,
            amountToken0Out: amountToken0,
            amountToken1Out: amountToken1,
            amountLiquidityIn:
            liquidityAmount
     }));
}
```

Snippet 4.3: Definition of `checkLiquidityToken()`

By burning one liquidity token, a user may emit an event which claims the user drained the entire pool.

Impact Malicious users may use this event to disrupt off-chain listeners. For example, they might shut down a front-end tracking pool balances, or trick another user into thinking they had lost a large amount of funds as part of a social attack.

If third-party applications are using the amounts to authorize other financial transfers (e.g. for bridging), this could allow direct theft of funds.

Recommendation Emit the withdrawn token amounts at the site of withdrawal to prevent manipulation.

Developer Response The developers now emit the events in the withdrawal functions.

Updated Veridise Response This resolves the above issue, but reintroduces the problem described in V-LUM-VUL-009. Additionally, users can choose to burn without emitting an event by forfeiting their share of `token0`. While there is not a clear reason to do this, it still prevents frontends from being guaranteed to observe state changes by listening for events.

Developer Response The `sender` is now passed as an argument to functions, as described in the fix for V-LUM-VUL-009.

Additionally, an event is now emitted at each transfer/mint/burn site.

4.1.4 V-LUM-VUL-004: Signatures are not unique to tokens

Severity	Medium	**Commit**	ad04eba
Type	Authorization	**Status**	Acknowledged
File(s)	`contracts/src/pool/PoolFactory.ts`		
Location(s)	createAccounts()		
Confirmed Fix At	N/A		

The `createAccounts()` method verifies that the signer deploying the pool has signed the `newAccount` as seen below:

```
approvedSigner.or(signerToken0).or(signerToken1).assertTrue("Invalid signer");
signature.verify(signer, newAccount.toFields()).assertTrue("Invalid signature");
```

Snippet 4.4: Snippet from `contracts/src/pool/PoolFactory.ts:createAccounts()`

However, the tokens involved in the Pool have no bearing on the message being signed. Therefore, once someone obtains a valid signature on `newAccount`, one can deploy a pair for any two tokens they choose.

Impact The `Pool` deployed to `newAccount` may have tokens completely unrelated to the tokens the signer intended to deploy the `Pool` for.

Recommendation Include `token0` and `token1` in the message being signed. Additionally, consider adding a network-specific identifier into the hash to avoid signature reuse across networks.

Developer Response The developers indicate that they do not plan to sign `newAccounts` provided by external parties. Instead, they will keep and store the private key associated to the `newAccount`, sign a transaction to create a pool, and then provide the transaction to the user requesting a new pool deployment.

4.1.5 V-LUM-VUL-005: Pool creation not as permissioned as intended

Severity	Low	**Commit**	ad04eba
Type	Logic Error	**Status**	Acknowledged
File(s)	`contracts/src/pool/PoolFactory.ts`		
Location(s)	createAccounts()		
Confirmed Fix At	N/A		

In `createAccounts()`, an entity must provide a signature authorizing the creation of a Pool for a token pair. The entities approved to do this are permissioned accounts stored in the `approveSigner` merkle tree, or either of the token accounts.

```
const approvedSigner = path.calculateRoot(signerHash).equals(approvedSignerRoot);
const signerToken0 = signer.equals(token0);
const signerToken1 = signer.equals(token1);
approvedSigner.or(signerToken0).or(signerToken1).assertTrue("Invalid signer");
```

Snippet 4.5: Snippet from `contracts/src/pool/PoolFactory.ts:createAccounts()`

Given that only one of the tokens is required to create a pair, a user could simply create a token (named `tokenTmp`) and create a pool for both `(token0, tokenTmp)` and `(token1, tokenTmp)`. Users would then be able to trade between `token0` and `token1`.

Impact The permissioned nature of pool creation is not strong enough to prevent an intermediary token from being used to enable swaps between two tokens.

Recommendation If permissioned deployment of pools is strictly required, then only allow `approvedSigners` to deploy, or require a signature from both tokens regarding the intention to create a pair between themselves.

Developer Response Initially, we wanted a permissionless creation, but we realized that in the event of a hard fork on mina, the owner of the pool's private key (which corresponds to the public key `newAccount`) could update the pool with the code he wanted, which poses quite a security problem. He may also have lost the key, so the funds could remain blocked in the pool.

After some consideration, we have opted for a hybrid model where users can deploy pools for tokens they deployed themselves, otherwise only we or our partners can deploy pools in order to keep the keys as a "trusted" entity.

Updated Veridise Response From our perspective, having just one of the pool token deployers sign off does not make them a trusted entity since anyone can take part in this process. Rather than relying on the above heuristic, it may be more effective to focus on aggressive documentation about the risks associated to hard forks and highlighting this risk in frontends. This is something every Mina project has to deal with, so users will likely be somewhat aware of the problem

In the long-term, having some sort of KYC or reputation/stake-based mechanism would be ideal.

Updated Developer Response In all case pool based on fungible token are on risk, because after a hard fork the problem will remain the same.

Lumina will provide only the smart contracts and the sdk, wo we are not responsible for the frontend implementation.

For the moment there is no ideal solution, but probably in the future o1js or some other entity will implement multisig on mina, and we will use this solution

4.1.6 V-LUM-VUL-006: Centralization Risk

Severity	Low	**Commit**	ad04eba
Type	Access Control	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/14`, `https://github.com/Lumina-DEX/lumina-v1-core/pull/18/`		

Similar to many projects, Lumina's Pool and Farm declare an administrator role that is given special permissions. In particular, these administrators are given the following abilities:

1. The owner is fully trusted to fairly compute the merkle root of `FarmRewards` and to fund the rewards contract.
2. The owner can call `withdrawDust()` at any time, returning all the rewards to themself.
3. For any upgradable contracts (which, in the second review, includes all contracts in this project), the owner can set the verification key of the contract.

See also V-LUM-VUL-008. Note further that the farm reward contracts allow at most one reward per user, so in addition to funds being present on the contract and their reward being present in the Merkle tree, users must also validate that no other rewards have been claimed on the address of their account.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious owner could distribute all farming rewards to a user.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Additionally, we recommend a delay in verification key upgrades to allow users to withdraw before the upgrade takes place.

Developer Response The developers added a delay of 2 weeks minimum before updating or withdrawing dust.

Updated Veridise Response This makes (2.) a much lower risk, as it provides a period of time in which users can withdraw their rewards before the owner has the ability to call `withdrawDust()`. However, this does not address (1.) or (3.).

In the case of (1.), this is a risk that users must validate themselves when using the protocol, as there are no guarantees around the structure of the Merkle tree.

Regarding (3.), the implemented delay is only enforced as a delay after contract deployment, and therefore 2 weeks after the deployment, admins can upgrade contracts instantly. It would be more secure for both (2.) and (3.) to take a two-step approach in which admins make a transaction informing their intent to upgrade a contract (or withdraw dust), and *then* the clock for the delay starts which gives users the time to make an informed decision before the admin action takes place. Lastly, this delay was only implemented for `FarmReward` and no other contracts.

Additionally, (3.) will always be a partial risk due to hard forks.

Updated Developer Response For the farm, we are implementing an upgrade in 2 steps with 24 hours delay. Also, we removed the withdraw dust method.

Given that tokens are upgradable, it is possible that in the event of an upgrade of a token used by the farm contract, the latter may need to be updated to remain compatible. A delay that is too long would run counter to the interests of users.

Updated Veridise Response The above changes have served to mitigate the centralization risks. In the current state, the protocol owner is trusted to perform the following actions:

1. Fund the farm appropriately.
2. Upgrade all contracts in the case of a Mina hard fork.
3. Upgrade the farm contract, but only after proposing the upgrade publicly at least 24 hours before the upgrade.
4. Upgrade the pool at any time.

(1) is a standard assumption of rewards contracts, and (2) is a trust assumption common to all applications on the Mina network. (3) is a very small risk, since users can withdraw funds in the 24-hour waiting period if the new vkey is malicious.

The developers indicated that (4) was acceptable in order to ensure pools can be upgraded when tokens are upgraded.

4.1.7 V-LUM-VUL-007: Ownership transferred in one step

Severity	Warning	**Commit**	ad04eba
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/src/pool/PoolFactory.ts`		
Location(s)	setNewOwner()		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/4		

`PoolFactory.setNewOwner()` transfers ownership of the pool factory with authorization from the previous owner. If any error is introduced into this transaction, ownership of the pool factory may be permanently lost.

Impact Transferring ownership is a highly sensitive operation. A one-step transfer may be more error-prone than necessary, potentially leading to loss of control of the pool factory.

Recommendation Use a two-step ownership transfer procedure, forcing the new owner to accept ownership. If the project needs to support renunciation of ownership, handle this as a special case.

Developer Response The developers now request a signature from the new owner during the `setNewOwner()` call to prevent an incorrect new owner address.

4.1.8 V-LUM-VUL-008: No provable initialization

Severity	Warning	**Commit**	ad04eba
Type	Authorization	**Status**	Fixed
File(s)	`contracts/src/pool/Pool.ts`, `contracts/src/pool/PoolTokenHolder.ts`		
Location(s)	deploy()		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/15		

Neither `Pool` nor `PoolTokenHolder` use a provable initialization method. Rather, both contracts override their `deploy()` method to prevent an honest user from directly deploying a pool or pool token holder. However, since this assertion is not performed in a provable context, this does not prevent a malicious user from deploying the `Pool` or `PoolTokenHandler` anyways.

```
async deploy() {
    await super.deploy();

    Bool(false).assertTrue("You can't directly deploy a pool");
}
```

Snippet 4.6: Snippet from `Pool`

Consequently, users must validate the deployment transaction of each `Pool` in order to trust the implementation.

In the second review, the Veridise analysts note the same non-provable deployment holds for the `Farm`. While this has a similar implications as for the pool, the `Farm` owner is fully trusted not only to decide who receives rewards, but to also fund the reward contract. Consequently, the non-provable deployment can likely be absorbed into the trust assumptions already existing on the owner. See also V-LUM-VUL-006.

Impact Malicious actors may deploy pools with maliciously configured permissions or liquidity accounts. This could allow them to deploy a ruggable pool with the same vkey used by trusted pools.

Additionally, some checks must be repeated (such as `checkToken()`) and some extra functions must be called to complete initialization (such as `Pool.setProtocol()`).

Recommendation Use a provable initialization method to

1. Set pool permissions.
2. Initialize the liquidity account.
3. Validate `token1` is not an empty PublicKey.
4. Initialize the `Pool`'s `protocol` and `delegator`.

Developer Response The developers added provable `init()` functions.

Updated Veridise response The implementation does not force the user to call the `init()` functions before using other functions. It also does not force the caller to initialize the state correctly, for example the permissions could be misconfigured.

The developers should either explicitly check that `init()` has been called, or revert to the original implementation and document the centralization risk as additional verification required by the user.

Previously, Veridise recommended the developers to follow the examples like this one. Upon further review, however, this example may be misleading.

In the linked example, a single transaction deploys the vkey via signature with an `isNew` requirement, then calls the `init()` `@method` to perform all account initialization. It also checks in all `@methods` that `provedState` is `true`. In thoery, this will enable developers to remove the checks performed at initialization from other functions in the contract.

However, `provedState` is true if and only if all state has been set via proof. So, one way `provedState` could be true without having called the provable `init()` function is if a malicious deployer deployed a malicious, upgradeable contract, provably edited the state to a malicious state or maliciously misconfigured the permissions, then upgraded to the `Pool`'s `vkey`.

Updated Developer Response Only approved signer can now deploy a pool. A token owner can no longer deploy a pool.

Updated Veridise Response Since users can deploy directly without using the factory, they may still produce a pool with a valid verification key, but improperly configured permissions or incorrectly initialized liquidity tokens. We would recommend documenting that users should check the deployment transaction of a pool to validate that the pool was created by the expected factory.

Updated Developer Response Ok perfect the goal is to reference only pool created by the `PoolFactory`, we just fetch the event from this contract, so we can ignore `Pool` deployed by an other method.

Updated Veridise Response We have updated the issue to fixed, and will add some text to the report highlighting that users must only use pools deployed by the factory.

4.1.9 V-LUM-VUL-009: Events should separate receiver/sender

Severity	Warning	**Commit**	ad04eba
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	`contracts/src/pool/Pool.ts`		
Location(s)	SwapEvent, WithdrawLiquidityEvent		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/10`		

In `o1js`, `this.self.sender` refers to an (otherwise-unconstrained) `PublicKey` provable variable cached on `this.self`. This has two important implications:

1. `this.self.sender` may be any `(Field, Bool)` pair.
2. `this.self.sender` may change whenever `this.self` changes (i.e. in each `@method` invocation).

Lumina emits events after swaps, deposits, and liquidity withdrawals. However, some of these workflows split up the swap across several `@method`s. For example, the `swapFromMinaToToken()` workflow (shown in the below snippet) sends output tokens to the user using `this.swap()`, then receives MINA as input from the user via `pool.swapFromMinaToToken()`.

```
@method async swapFromMinaToToken(frontend: PublicKey, taxFeeFrontend: UInt64,
    amountMinaIn: UInt64, amountTokenOutMin: UInt64, balanceInMax: UInt64,
    balanceOutMin: UInt64
) {
    const pool = new Pool(this.address);
    // we check the protocol in the pool
    const protocol = pool.protocol.get();
    await this.swap(protocol, frontend, taxFeeFrontend, amountMinaIn,
    amountTokenOutMin, balanceInMax, balanceOutMin, true);
    await pool.swapFromMinaToToken(protocol, amountMinaIn, balanceInMax);
}
```

Snippet 4.7: Definition of `PoolTokenHolder.swapFromMinaToToken()`

Note that the `sender` used by `this.swap` is not passed to `poolswapFromMinaToToken()`. However, `swap` emits the `SwapEvent` as shown below. The `sender` used in `swap()` may be a different account than the one used in `swapFromMinaToToken()`, since `this.self` will refer to a different `AccountUpdate`, but only one account is emitted (the account receiving funds).

```
this.emitEvent("swap", new SwapEvent({ sender, amountIn: amountTokenIn, amountOut }))
    ;
```

Snippet 4.8: Snippet from `this.swap()`

An analogous situation exists for the `WithdrawLiquidityEvent`.

Impact Event listeners cannot accurately track swap senders/receivers. This may cause service issues for off-chain listeners, or increase the attack surface for social attacks based on errors induced in frontends through incorrect events.

Recommendation Include the `sender` and `receiver` accounts separately in the `SwapEvent` and `WithdrawLiquidityEvent`.

Developer Response The developers now pass the `sender` directly to each function which must use the same `sender` for cross-chain interactions.

4.1.10 V-LUM-VUL-010: Pool actions may be DOSed for gas cost of deposit

Severity	Warning	**Commit**	ad04eba
Type	Denial of Service	**Status**	Acknowledged
File(s)	`contracts/src/pool/Pool.ts, contracts/src/pool/PoolTokenHandler.ts`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Unlike in blockchains such as Ethereum, in which transactions must be protected against slippage by the application at execution time, o1js transactions consist of precomputed `AccountUpdates` which describe the state changes made by the transaction.

For the purposes of the Lumina DEX, this means that users commit to the amount of liquidity tokens minted or received, the amount of MINA spent or received, and the amount of underlying tokens spent or received at transaction proof/signing time. In particular, these transactions are immune to slippage.

However, in order to maintain solvency of the pool, bounds on the pool state are checked at transaction runtime. For example, when withdrawing a share of the liquidity, the user is entitled to `numLiquidityTokens * numReserveTokens / totalNumLiquidityTokens`. The number of reserve tokens or liquidity token supply may change between transaction creation and execution. To handle this, Lumina checks that the values used when the transaction is created, e.g., `reserveToken0Max` and `supplyMin`, err in favor of the pool. Since the network node will validate the preconditions that `numReserveTokens <= reserveTokenMax` and `supplyMin <= totalNumLiquidityTokens`, the amount of tokens received for the withdrawal is less than or equal to the number of tokens which would be received if the transaction were created and executed synchronously. Essentially, the system checks for slippage against the *pool*, rather than checking for slippage against the pool user.

Unfortunately, this gives rise to a potential griefing attack. Unlike traditional AMMs, which require price manipulation to trigger slippage bounds and deny a transaction, the bounds checked in Lumina may be violated using only benign deposits and withdrawals. For example, if Alice intends to withdraw `100` liquidity tokens from a pool with `1000` total liquidity tokens, and `1000` each of token 0 and token 1, she may naively set `reserveToken0Max` and `reserveToken1Max` to `1000`, and set `supplyMin` to `1000`. An attacker, Bob, may then deposit a single token 0 and token 1, bringing the reserves to `1001`, and causing Alice's proof to fail.

Note that no fees are taken for deposits/withdrawals, so the cost to Bob is only the cost of gas for depositing/withdrawing the single liquidity token.

Impact The barrier to preventing someone else's swap, withdrawal, or deposit from taking place is

1. Paying the gas costs to frontrun them and then withdraw tokens as necessary.
2. Maintaining enough capital to violate the pool's slippage protections.

This may allow attackers to lock a users' funds in the pool, or prevent them from taking advantage of time-sensitive opportunities.

Recommendation Clearly document this possibility. Recommend usage of private transaction pools for frequently denied transactions.

Developer Response The developers have indicated an intent to provide the requested documentation surrounding these methods.

4.1.11 V-LUM-VUL-011: o1js/Token Best Practices

Severity	Info	**Commit**	ad04eba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	contracts/src/pool/MathLibrary.ts		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/3, https://github.com/Lumina-DEX/lumina-v1-core/pull/5, https://github.com/Lumina-DEX/lumina-v1-core/pull/13, https://github.com/Lumina-DEX/lumina-v1-core/pull/16, https://github.com/Lumina-DEX/lumina-v1-core/pull/17		

The following locations diverge from `o1js` best practices.

1. Throughout the project, consider incorporating linting/automated checks into CI to remove unused imports.
2. `contracts/src/pool/MathLibrary.ts`:
   a) `mulDivMod()`:
      i. This function computes the quotient of a division from a prover-hint (shown in the below snippet). Instead of supplying the intended type of `q` directly to the `Provable.witness` function, the `Field` type is supplied, and the `UInt64.check()` method is manually inlined and invoked on the witness result. Instead of manually range checking, consider calling `Provable.witness(UInt64, ...)`, which will be more robust to changes in the `o1js` implementation moving forward and benefit from any future performance improvements. This also decreases the number of unsafe `UInt64` constructions (via `new UInt64`), which is an error-prone operation.

```
let q = Provable.witness(Field, () => new Field(x.toBigInt() / y_.
    toBigInt()));
RangeCheck.rangeCheckN(UInt64.NUM_BITS, q);
// [VERIDISE] Code elided...
let q_ = new UInt64(q.value);
```

Snippet 4.9: Snippet from `mulDivMod()`

      ii. Usage of `new UInt64()` may lead a reader to believe a range-check is occurring automatically. Instead, use `UInt64.Unsafe.fromField()` to communicate to the reader that a dangerous operation is taking place, and additional scrutiny is required.
      iii. Declaring the return type makes this function more robust to changes in the future.
3. `contracts/src/pool/Pool.ts`:
   a) `Pool.events`:
      i. Consider renaming `events.BalanceChange` to `events.balanceChange`.
      ii. Avoid using std library types like `PublicKey` as event types to increase maintainability in the case of future changes to events or standard library types.
4. `contracts/src/pool/PoolFactory.ts`:

   a) `createAccounts()`: Consider allowing different pools to use different token symbols.
   b) `createState()`: Declaring the return type makes this function more robust to changes in the future.

The following locations in `contracts/src/farming` diverge from `o1js` best practices:

1. `contracts/src/farming/Farm.ts`:
   a) `deploy()`:
      i. Ensure that `startTimestamp` is before `endTimestamp`.
      ii. Ensure the `endTimestamp` is in the future, this can be added as a precondition to the deployment `AccountUpdate`.
      iii. If the application desires it, also ensure that `startTimestamp` is in the future.
      iv. It would be best to use multiples of the slot times for the timestamps, so the block number is easy to compute.
2. `contracts/src/farming/FarmTokenHolder.ts`:
   a) `events.upgrade`: Although this will be stable, it is better to wrap with a type when possible.
   b) `deploy()`:
      i. For non-provable deployments, requiring `this.account.isNew.requireEquals(new Bool(true))` will ensure people validating the deployment transaction don't have to look at any prior transactions.
3. `contracts/src/farming/FarmReward.ts`:
   a) `events.upgrade`: Although this will be stable, it is better to wrap with a type when possible.

Impact Over time, the code may become less readable and more difficult to maintain.

Recommendation Implement the above recommendations.

Developer Response The developers implemented fixes for all of the above except (1), (2)(a)(ii), and (4)(a).

Updated Veridise Response The Veridise team added best practices which should be implemented for the farming contracts.

Updated Developer Response Regarding the pool, the developers added a linter to the CI, resolving point (1) for the pool contracts. Additionally, they decided not to implement customizable token names, because it will be difficult to deal with strings in prover code, and want to avoid the possibility of the user putting names that can be shocking.

The developers implemented fixes for all of the above recommendations regarding the farming contracts, except for 1(a)(iii) and 1(a)(iv) for the farming contract.

The developers also pointed out that 1(a)(ii) is already implemented via

```
this.network.timestamp.requireBetween(UInt64.zero, args.endTimestamp);
```

Updated Veridise Response The Veridise analysts acknowledge that 1(a)(ii) was invalid.

The remaining points are 2(a)(ii) for the pool, and 1(a)(iii)-1(a)(iv) of the farming contracts.

Updated Developer Response The developers resolved 2(a)(ii) for the pool and 1(a)(iv) for the farming contracts.

Updated Veridise Response The Veridise analysts have reviewed the fixes, and marked the issue as resolved. To summarize, 1(a)(ii) is invalid, and all other recommendations were implemented.

4.1.12 V-LUM-VUL-012: Typos and incorrect comments

Severity	Info	**Commit**	ad04eba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/8`		

Description In the following locations, the auditors identified minor typos and potentially misleading comments:

1. `contracts/src/pool/Pool.ts`:
   a) `minimunLiquidity` should be rewritten as `minimumLiquidity`.
2. `contracts/src/pool/PoolTokenHolder.ts`:
   a) `swap()`: `Inccorect` should be rewritten as `Incorrect`.
3. `contracts/src/pool/PoolFactory.ts`:
   a) The comment on `SignerMerkleWitness` incorrectly states there are 32 possible signers, whereas the height is set to 32 and therefore 2^{32} signers are possible.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have implemented the recommendation.

4.1.13 V-LUM-VUL-013: [Upd.] Unused and duplicate program constructs

Severity	Info	**Commit**	ad04eba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/9`		

Description The following program constructs are unused or re-implement existing functionality:

1. `contracts/src/pool/Pool.ts`:
   a) `supply()`: In the `isFirstSupply` case, the `balanceLiquidity.equals(UInt64.zero)` check is asserted to be true in two different locations.
   b) `swapFromTokenToMina()`: This function reimplements much of the functionality defined in `Pool.sendTokenAccount()`.
   c) `checkLiquidityToken()`: `reserveMinaMin` is unused other than checking that it is non-zero. It is also improperly named, as this route is intended for usage by token-token pools. The minimum token amount is checked on the pool token holder during withdrawal, so this argument may be safely removed.
   d) `checkLiquidityToken()` and `withdrawLiquidity()`: Both of these functions contain the same logic for burning a user's LP shares. This functionality could be abstracted away, or the two functions could be combined into one with a conditional on `isMinaPool`.
2. `contracts/src/pool/PoolTokenHolder.ts`:
   a) `withdrawLiquidity()`: `amountToken1Min` is unused.
   b) `checkToken()`: This function is implemented in both the `Pool` class and the `PoolTokenHolder` class. Consider creating a function to abstract away the shared functionality.

The following locations in `contracts/src/farming` have unused or duplicate constructs:

1. `contracts/src/farming/Farm.ts`:
   a) The `FarmingInfo` struct is defined, but unused.
2. `contracts/src/farming/FarmTokenHolder.ts`:
   a) `approveBase()`: This method is unnecessary as this contract extends from `SmartContract` and not `TokenContract`.
3. `contracts/src/farming/FarmReward.ts`:
   a) `this.token`: This variable is unused.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers have removed the unused and duplicate code.

4.1.14 V-LUM-VUL-014: Optimizations

Severity	Info	**Commit**	ad04eba
Type	Gas Optimization	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/7`		

In the following locations, the auditors identified missed opportunities for potential gas savings.

1. `contracts/src/pool/Pool.ts`:
   a) `setDelegator()`, `setProtocol()`: These functions contain assertions preventing someone from calling them repeatedly. However, the functions are idempotent. Calling them more than once (if the factory's protocol/delegator have not changed) does nothing. Consider moving these assertions within a `Provable.asProver()` block to avoid adding unnecessary constraints.
2. `contracts/src/pool/PoolTokenHolder.ts`:
   a) `swap()`: Asserting `balanceOutMin > 0` is unnecessary. It is asserted that `amountTokenOutMin > 0` and `amountTokenOutMin < balanceOutMin`, which together imply that `balanceOutMin > 0`.

Impact Gas may be wasted, costing users extra funds.

Recommendation Perform the optimizations.

Developer Response The developers have implemented the optimizations.

4.1.15 V-LUM-VUL-015: New pool event should emit signer

Severity	Info	**Commit**	ad04eba
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/src/pool/PoolFactory.ts`		
Location(s)	createAccounts()		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/3		

The `createAccounts()` function will emit a `PoolCreationEvent` once the pool and corresponding `PoolTokenHolder` accounts have been created. However, as can be seen below, the unconstrained sender is used as the `sender` field of the event:

```
const sender = this.sender.getUnconstrainedV2();
this.emitEvent("poolAdded", new PoolCreationEvent({ sender, poolAddress: newAccount,
    token0Address: token0, token1Address: token1 }));
```

Snippet 4.10: Snippet from `contracts/src/pool/PoolFactory.ts:createAccounts()`

Given that the `sender` is unconstrained, and also may not be the entity approving this pool creation, listeners of this event may not receive accurate information.

Impact An unrelated `PublicKey` may be emitted for this event.

Recommendation Emit the signer of the pool creation instead of `sender`.

Developer Response The developers add the signer to the event, and now retrieve the sender with a constrained `getAndRequireSignatureV2()` method call.

Glossary

AMM Automated Market Maker. 1

DEX Decentralized Exchange. 1

Mina Mina Protocol is a succinct 22KB blockchain utilizing zero-knowledge proofs. See `https://minaprotocol.com` for more details. 1, 32

o1js A zero-knowledge TypeScript library which allows users to write zero-knowledge circuits without writing constraints themselves. It is also used to write zkApps for the Mina blockchain. For more information, see `https://docs.minaprotocol.com/zkapps/o1js`. 1

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. 1, 2, 32

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 32

zkApp A smart contract written for the Mina blockchain. See `https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks` for more. 32