Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

LuminaDEX Multisig

Veridise Inc.

August 28, 2025

► **Prepared For:**

Lumina Labs
https://luminadex.com

► **Prepared By:**

Benjamin Sepanski
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Aug. 28, 2025	V2 - Incorporated issue fixes
Aug. 13, 2025	V1
Aug. 12, 2025	Initial Draft

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 4

3 Security Assessment Goals and Scope 5
- 3.1 Security Assessment Goals . . . 5
- 3.2 Security Assessment Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 6

4 Trust Model 7
- 4.1 Operational Assumptions. . . . 7
- 4.2 Privileged Roles. . . . 7

5 Vulnerability Report 9
- 5.1 Detailed Description of Issues . . . 10
  - 5.1.1 V-LMN-VUL-001: Signed data is not unique to a given action . . . 10
  - 5.1.2 V-LMN-VUL-002: The built in toFields method should be used . . . 13
  - 5.1.3 V-LMN-VUL-003: Maintainability Improvements . . . 14
  - 5.1.4 V-LMN-VUL-004: o1js best practices . . . 16
  - 5.1.5 V-LMN-VUL-005: Optimization Improvements . . . 17
  - 5.1.6 V-LMN-VUL-006: Dependencies contain Regex DoS vulnerability . . . 18

A Intended Behavior: Non-Issues of Note 20
- A.0.1 V-LMN-APP-INFO-001: Upgrade reorders fields . . . 20

B Verification Keys 22
- B.1 Latest Fix Commit . . . 22
- B.2 Audit Commit . . . 22
- B.3 Script. . . . 23

Glossary 26

1 Executive Summary

From Aug. 7, 2025 to Aug. 8, 2025, Lumina Labs engaged Veridise to conduct a security assessment of their LuminaDEX. The security assessment covered the addition of a multi-signature approval system into their AMM o1js smart contracts. Veridise conducted the assessment over 4 person-days, with 2 security analysts reviewing the project over 2 days on commit `0f92fd1d`*. 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. The security assessment covered an update to LuminaDEX, which the Veridise analysts previously audited†, most recently at commit `4ac8e2be`. LuminaDEX is an AMM, consisting of 3 smart contracts. The `Pool` and `PoolTokenHolder` contracts implement the details of swapping tokens, providing liquidity to the pool, and removing liquidity from the pool. The `PoolFactory` contract implements the logic for deploying the aforementioned contracts.

This audit covered an update to how privileged `PoolFactory` actions are authorized. In the previous audit, a signature from a designated account was required to apply `AccountUpdates` changing the verification key of any contracts or to change pool parameters. The updated logic implements a multi-signature. This requires signatures from two different signers to perform an update. The signers are stored on-chain in a Merkle tree which stores each signer's access rights at a designated leaf associated to the signer's public key. This allows different keys to be used for different operations which require varied security levels, and ensures multiple accounts must be compromised in order to change important protocol configurations.

In more detail, there are five operations which require privileged access. First, one may update the verification keys of the `PoolFactory` (which may arbitrarily change the logic of the contract, as well as provide new verification keys for the `Pool` and `PoolTokenHolders`). Second, one may change the `delegator`, i.e. the Mina account to which staked Mina is delegated. Third, one may change the `protocol`, i.e. the recipient of fees from the protocol. Fourth, one may have the right to deploy `Pools` and `PoolTokenHolders` from the `PoolFactory`. Finally, one may change the Merkle root of signers itself. Each of these different operations is identified with a distinct access right. Two different signers in the current signer tree with the necessary access right must sign off an update for it to be performed.

Code Assessment. The LuminaDEX developers provided the source code of the LuminaDEX contracts for the code review. The source code appears to be original code written by the Lumina Labs developers. It contains some documentation in the form of READMEs and documentation comments on contracts, functions, and storage variables. To facilitate the Veridise security analysts' understanding of the code, the LuminaDEX developers shared online documentation‡,

* `https://github.com/Lumina-DEX/lumina-v1-core/0f92fd1d`
† `https://veridise.com/audits-archive/company/lumina-labs/lumina-dex-2025-01-19/`
‡ `https://lumina-dex.github.io/lumina/`

information on their planned deployment configuration, and met with the analysts to describe the intention of the new features.

The source code contained a test suite, which the Veridise security analysts noted covered all happy paths. It checks each update path, and tests out the upgrade functionality to ensure that `Pool` contracts are able to pull their upgraded vkey from an upgraded `PoolFactory`.

The multi-sig access control is only enforced between upgrades of the Mina network. As hard forks invalidate verification keys, during a hard fork the verification-key upgrade permission degrades to signature[§]. The developers expressed their intent to deploy using a private key which is encrypted by two different keys so that any operations performed with this key require at least two parties to agree on the upgrade off-chain[¶]. This practice helps to improve the security of the project during network upgrades, although still requires users to trust the deployers. See Section 4 for more discussion.

Summary of Issues Detected. The security assessment uncovered 6 issues, 0 of which are assessed to be of high or critical severity by the Veridise analysts. All issues have been fixed, see Table 2.3. The Veridise analysts identified 1 low-severity issue, V-LMN-VUL-001, which shows that in some cases signatures can be reused for unintended actions, as well as 1 warning and 4 informational findings. The Veridise analysts have validated fixes provided by the LuminaDEX developers for all of these issues.

Additionally, section V-LMN-APP-INFO-001 notes that this upgrade from the previous version of the `PoolFactory` changes the order of state variables. However, the developers informed the Veridise team that the project is being redeployed rather than upgraded. The discussion was retained in Appendix A to ensure users correctly understand the upgrade process, which will require manual migration of funds.

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

Upgrades. Currently, `Pools` and `PoolTokenHolders` must be updated individually. This may mean that, for brief periods, the verification keys may become out of sync during an upgrade. Future versions should consider the following two improvements:

- When updating the verification key from a `Pool`, invoke the method to also update the verification key for any `PoolTokenHolders`. Only allow upgrading the `PoolTokenHolders` through this method to ensure the `Pool` and all its token holders always remain in sync.
- Consider polling the factory on each `Pool` method for a new verification key to ensure that all `Pools` are the latest version.

Without at least this first change, creating safe upgrades will be an error-prone process as intermediate, partially-upgraded states must be carefully reasoned about to prevent attacks during an upgrade.

[§] For more info, see https://docs.minaprotocol.com/zkapps/o1js-reference/variables/Permissions#verificationkeyproofduringcurrentversion

[¶] https://github.com/Lumina-DEX/lumina/blob/b4a0878dd43e25c3ac05bfcb1d4c552830219c99/packages/signer/src/helpers.ts#L92

Operational practices. Currently, signatures can be used until their deadline expires. While the system is designed so that signers must sign both the old value and the new value, signers should use short deadlines and never sign multiple updates within periods which overlap. Additionally, distinct keys should be used for different operations and should be rotated at a regular cadence. See Section 4 for a more complete discussion.

Verification key generation. Instead of hard-coding the `Pool` and `PoolTokenHolder` verification keys into the contract, consider storing them in a map from network name to verification key. This map should then be validated by the tests, so that the CI ensures the verification keys stay up to date. This will help users more easily identify commit-network pairs with contract verification keys. See Appendix B for concrete verification keys at the audit commit.

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
LuminaDEX	`0f92fd1d`[a]	TypeScript	o1js

[a] See Appendix B for the verification keys and hashes associated to the above commit.

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 7–Aug. 8, 2025	Manual & Tools	2	4 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	1	1	1
Informational-Severity Issues	4	4	4
TOTAL	6	6	6

Table 2.4: Category Breakdown.

Name	Number
Maintainability	3
Data Validation	2
Supply Chain Vulnerability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Are signatures correctly verified?
- Can signatures be reused after an update has been performed?
- Can signatures be used for actions other than the ones they are intended to permit?
- Are unused signatures eventually invalidated?
- Can a single signer perform an operation without a signature from a distinct signer?
- Are access permissions properly stored and enforced?
- Can the pool reach a denial-of-service state?
- Can any users escalate their access privileges?
- Are updates to the signer list properly access controlled and also performed via multi-signature?
- Do `Pools` pull up to date information from the `PoolFactory`?
- Are the upgrade mechanisms properly tested?
- Are common zkApp attack vectors present, such as insecure permissions, denial of service via actions/receiver, or missing state checks?

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:

- *Semgrep.* Security analysts ran Semgrep rules to statically analyze if certain o1js pitfalls were encountered.
- *npm audit.* Security analysts ran `npm audit` to ensure no dependencies were out of date.

Scope. The scope of this security assessment is limited to the changes to files in the `contracts/src/pool` folder of the source code provided by the LuminaDEX developers since commit `4ac8e2be`. This directory contains the smart contract implementation of the LuminaDEX.

Methodology. Before the security assessment, the Veridise security analysts met with the LuminaDEX developers to ask questions about the code. Veridise security analysts reviewed the reports of previous audits for LuminaDEX, inspected the provided tests, and read the LuminaDEX documentation. They then began a review of the code assisted by both static analyzers.

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 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for LuminaDEX.

- The developers deploy using the provided (non-provable) `deploy()` function.
- All tokens used conform to the Mina fungible token standard.

4.2 Privileged Roles.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assume that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles:
  - Two multisig signers with the `updateFactory` right change the verification key of the `PoolFactory`.
  - Two multisig signers with the `updateSigner` right can change the list and permissions of all signers.
  - After a Mina protocol upgrade, the `PoolFactory` private key owner can set the verification key.
  - The private key of a deployed `Pool` can change the verification key.
- Highly-privileged, non-emergency roles:
  - The deployer of each contract is responsible for upgrading its vkey after a Mina network upgrade. The developers expressed their intent to deploy using a private key which is encrypted by two different keys so that any operations performed with this key require at least two parties to agree on the upgrade off-chain*. These practices can help improve the security of the project during network upgrades. The deployers must be sure to not share their secret keys with anyone, and must also trust each other not to store the keying material during generation or after it has

* https://github.com/Lumina-DEX/lumina/blob/b4a0878dd43e25c3ac05bfcb1d4c552830219c99/packages/signer/src/helpers.ts#L92

been unencrypted for signing. Auditing of this procedure, or use of a distributed signing scheme like FROST may be fruitful alternatives for future deployments.

- Limited-authority, non-emergency roles:
  - Two multisig signers with the `updateProtocol` right can change the collector of protocol fees.
  - Two multisig signers with the `updateDelegator` right can change the Mina stake delegate of all `Pools`.
  - A signer with the `deployPool` right can deploy `Pools`.

Readers should note that upgrades are currently listed as an emergency operation. This is because there is no ability to pause the contracts in case of an emergency. In this case, upgrading will be the only recourse. This action should be taken with extreme caution as it is an extremely sensitive operation. A future design might allow for an emergency upgrade to a pre-defined contract effectively pausing the protocol, while requiring a timelock for upgrades to other verification keys.

Operational Recommendations. Highly-privileged operations should be operated by separate key holders or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary. Note that the roles that can update the `PoolFactory` verification key are particularly sensitive, as they enable a complete protocol takeover that can be propagated to all previously-deployed `Pools`.

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:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- 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.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-LMN-VUL-001	Signed data is not unique to a given action	Low	Fixed
V-LMN-VUL-002	The built in toFields method should be used	Warning	Fixed
V-LMN-VUL-003	Maintainability Improvements	Info	Fixed
V-LMN-VUL-004	o1js best practices	Info	Fixed
V-LMN-VUL-005	Optimization Improvements	Info	Fixed
V-LMN-VUL-006	Dependencies contain Regex DoS . . .	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-LMN-VUL-001: Signed data is not unique to a given action

Severity	Low	**Commit**	0f92fd1
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/src/pool/Multisig.ts:319-331`		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/28, https://github.com/Lumina-DEX/lumina-v1-core/pull/26, 48f867f		

The hash of the data that is signed during upgrade operations does not contain information about the action that is being performed, it only contains information regarding the new state that is to be set. This can be seen in the methods which verify a transaction's right to set the `protocol` or the `delegator`.

```
verifyUpdateDelegator(updateInfo: UpdateAccountInfo) {
    const right = SignatureRight.canUpdateDelegator();
    verifySignature(this.signatures, updateInfo.deadlineSlot, this.info, this.info.
    approvedUpgrader, updateInfo.toFields(), right);
}

verifyUpdateProtocol(updateInfo: UpdateAccountInfo) {
    const right = SignatureRight.canUpdateProtocol();
    verifySignature(this.signatures, updateInfo.deadlineSlot, this.info, this.info.
    approvedUpgrader, updateInfo.toFields(), right);
}
```

Snippet 5.1: Snippet from `Multisig.ts`

As can be seen, both methods utilize the `UpdateAccountInfo` struct, which contains the `oldUser`, `newUser` and `deadlineSlot`. The `UpdateAccountInfo.hash()` is what is signed by the signers in order to authorize the upgrades. Due to the hash preimage not containing data regarding the type of action that is being signed, a signature destined for an upgrade of the `protocol` can be reused for an upgrade of `delegator`, and vice versa.

Note that since the `UpdateAccountInfo` contains the `oldUser`, the signature can only be reused in the instance of `protocol` and `delegator` both previously being set to the same value. Additionally, signatures are private inputs into the generated circuits.

Impact If an attacker is able to obtain the signatures destined for `verifyUpdateDelegator()` or `verifyUpdateProtocol()`, he could replay the signature in order to force an unintended update to the `protocol` or `delegator`. This also may occur in other fields if other functionality is built using the `UpdateAccountInfo`.

Recommendation Include the `right` that is being exercised in the data that is to be signed.

Proof of Concept The below test script demonstrates the signature reuse issue:

```
import {
  Bool,
  Field,
  MerkleMap,
  Mina,
  Poseidon,
  PrivateKey,
  Signature,
  UInt32,
} from 'o1js';
import {
  Multisig,
  MultisigInfo,
  SignatureInfo,
  SignatureRight,
  UpdateAccountInfo,
} from '../../pool/Multisig';

/** Helper to create a SignatureInfo for a given signer */
function createSignatureInfo(
  key: PrivateKey,
  right: SignatureRight,
  merkle: MerkleMap,
  data: Field[]
): SignatureInfo {
  const user = key.toPublicKey();
  const witness = merkle.getWitness(Poseidon.hash(user.toFields()));
  const signature = Signature.create(key, data);
  return new SignatureInfo({ user, witness, signature, right });
}

/**
 * PoC test: signatures authorizing a delegator update can also be used to update
 * the protocol address. Passing indicates a vulnerability.
 */
describe('Signature reuse between delegator and protocol updates', () => {
  it('reuses delegator signatures to update protocol', async () => {
    // setup a local Mina instance for testing
    const Local = await Mina.LocalBlockchain({ proofsEnabled: false });
    Mina.setActiveInstance(Local);

    // signers and merkle map
    const signer1 = PrivateKey.random();
    const signer2 = PrivateKey.random();
    const merkle = new MerkleMap();

    // grant both delegator and protocol update rights
    const bothRights = new SignatureRight(
      Bool(false),
      Bool(false),
      Bool(false),
      Bool(true), // updateProtocol
```

```
      Bool(true), // updateDelegator
      Bool(false)
    );
    merkle.set(
      Poseidon.hash(signer1.toPublicKey().toFields()),
      bothRights.hash()
    );
    merkle.set(
      Poseidon.hash(signer2.toPublicKey().toFields()),
      bothRights.hash()
    );

    // data to be signed
    const deadline = UInt32.from(1);
    const update = new UpdateAccountInfo({
      oldUser: signer1.toPublicKey(),
      newUser: signer2.toPublicKey(),
      deadlineSlot: deadline,
    });
    const data = update.toFields();

    const info = new MultisigInfo({
      approvedUpgrader: merkle.getRoot(),
      messageHash: update.hash(),
      deadlineSlot: deadline,
    });
    const sig1 = createSignatureInfo(signer1, bothRights, merkle, data);
    const sig2 = createSignatureInfo(signer2, bothRights, merkle, data);
    const multisig = new Multisig({ info, signatures: [sig1, sig2] });

    // what signers intended: update the delegator
    expect(() => multisig.verifyUpdateDelegator(update)).not.toThrow();

    // vulnerability: same signatures also authorize protocol update
    expect(() => multisig.verifyUpdateProtocol(update)).not.toThrow();
  });
});
```

Developer Response The developers now add a prefix string to each multisig function.

5.1.2 V-LMN-VUL-002: The built in toFields method should be used

Severity	Warning	**Commit**	0f92fd1
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/src/pool/Multisig.ts`		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/27`, 7f04bde		

The classes which extend the o1js `Struct` type should use the built-in `toFields()` method instead of reimplementing their own. This improves the maintainability of the codebase in case class fields change and prevents issues like the one found in the `MultisigInfo` implementation described below.

The following code snippet shows the implementation of `toFields` for th`MultisigInfo` struct:

```
toFields(): Field[] {
    return this.messageHash.toFields();
}
```

Snippet 5.2: Snippet from `Multisig.ts`

This does not include the `approvedUpgrader` and `deadlineSlot` fields, therefore the `toFields()` encoding is incomplete.

Instead, the implementation could be written using the o1js built-in methods:

```
toFields(): Field[] {
    return MultisigInfo.toFields(this);
}
```

Impact The codebase requires additional maintenance. Additionally, manually written encodings may be incorrect, as shown above.

Fortunately, `MultisigInfo.toFields()` is not used, so there is no direct impact to the incorrect encoding.

Recommendation Remove the `toFields()` implementations and rely on the built-in implementation `<Struct Type>.toFields(obj)`.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.3 V-LMN-VUL-003: Maintainability Improvements

Severity	Info	**Commit**	0f92fd1
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/src/pool/` ▶ `Multisig.ts` ▶ `PoolFactory.ts`		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/31`, 4e53d8f		

The maintainability of the code may be improved by changes in the following locations:

1. `Multisig.ts:hasRight()`: Explicitly specify the return type.
2. `Multisig.ts:verifyUpdateSigner()`: This function does not verify the `upgradeInfo.oldRoot` and `this.info.approvedUpgrader` are equal. Although this is done in `PoolFactory.updateApprovedSigner()`, it would be best to verify this directly in case future uses of this function fail to check.
3. `Multisig.ts`: Specify the signing scheme size by a variable, instead of hardcoding at various locations.
4. `Multisig.ts`: The `UpgradeInfo` struct is unused.
5. `Multisig.ts` and `PoolFactory.ts`: The intended usage of Multisig is to have operations verified by 2 signatures from separate entities. There are multiple locations where an array of signatures should be typed as a static-sized array `[SignatureInfo, SignatureInfo]` instead of a variable sized array `SignatureInfo[]`. This includes:
   a) `MultisigSigner.signatures`
   b) `MultiSig.constructor()`
   c) `verifySignatures()`
   d) `PoolDeployProps.signatures`
6. `PoolFactory.ts.PoolDeployProps`: This struct contains the `signatures` and `multisigInfo` fields, which could instead be replaced by a `Multisig`
7. `IPoolState.ts:checkToken()`: Fix the documentation comment regarding the return value, as it is incorrect.
8. The `uppdatePool` right (typo specified below) is never utilized.

The following typos were found in the codebase:

1. `Multisig.ts`
   a) `SignatureRight.uppdatePool` should be `updatePool`.
   b) The comments on the various `verifyUpdateX` functions contain `subbit` instead of `submit`.

Impact Issues may arise in the future from inconsistencies in the codebase. Additionally, the recommended changes will improve the ability to understand the code.

Recommendation Implement the recommendations.

Developer Response (1.), (2.), (4.), (6.), (7.) are implemented. (8.) no longer applies.

(3.) and (5.) are not implemented due to difficulties with the type system.

Veridise Response We would still recommend updating the constructor to perform type checks, and adding a length check where the type system can't enforce the length 2 invariant.

Updated Developer Response (3.) and (5.) were implemented by adding dynamic checks.

5.1.4 V-LMN-VUL-004: o1js best practices

Severity	Info	**Commit**	0f92fd1
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/src/pool/PoolFactory.ts`		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/tree/fix/best-pratice`, ae8c454		

The following locations diverge from o1js best practices:

1. `PoolFactory:deploy()`: As mentioned in the previous Veridise report, for non-provable deployments, requiring `this.account.isNew.requireEquals(new Bool(true))` will ensure people validating the deployment transaction do not have to look at any prior transactions.

Impact The codebase may be more difficult to maintain, and unforeseen consequences may arise.

Recommendation Implement the specified recommendations.

Developer Response The developers implemented the recommendation.

5.1.5 V-LMN-VUL-005: Optimization Improvements

Severity	Info	**Commit**	0f92fd1
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/src/pool/Multisig.ts`		
Confirmed Fix At	https://github.com/Lumina-DEX/lumina-v1-core/pull/29, 08e3a63		

The security analysts found the following opportunities for optimization improvements:

1. `Multisig.ts`: The implementation of `hash()` for `SignatureRight` only requires that it produces a unique value for each possible setting of the booleans. Compared to the current concatenation of fields, this can be done more efficiently by taking a weighted sum of the values.

Impacts More constraints are created than necessary, elongating the time it takes to prove the execution of a given contract.

Recommendation Implement the recommended optimization improvements.

Developer Response The developers now represent each right as a unique power-of-2, and the `hasRight()` function will simply ensure the result of a bitwise AND of a user's right and the requested right is equal to the requested right.

5.1.6 V-LMN-VUL-006: Dependencies contain Regex DoS vulnerability

Severity	Info	**Commit**	0f92fd1
Type	Supply Chain Vulnerability	**Status**	Fixed
Location(s)	`contracts/package.json`		
Confirmed Fix At	`https://github.com/Lumina-DEX/lumina-v1-core/pull/30`		

`npm audit` identifies a few regex-based denial-of-service vulnerabilities present in the project dependencies.

```
@babel/helpers  <7.26.10
Severity: moderate
Babel has inefficient RegExp complexity in generated code with .replace when
    transpiling named capturing groups - https://github.com/advisories/GHSA-968p-4wvh
    -cqc8
fix available via ‘npm audit fix‘
node_modules/@babel/helpers

@babel/runtime  <7.26.10
Severity: moderate
Babel has inefficient RegExp complexity in generated code with .replace when
    transpiling named capturing groups - https://github.com/advisories/GHSA-968p-4wvh
    -cqc8
fix available via ‘npm audit fix‘
node_modules/@babel/runtime

brace-expansion  1.0.0 - 1.1.11 || 2.0.0 - 2.0.1
brace-expansion Regular Expression Denial of Service vulnerability - https://github.
    com/advisories/GHSA-v6h2-p8h4-qcjw
brace-expansion Regular Expression Denial of Service vulnerability - https://github.
    com/advisories/GHSA-v6h2-p8h4-qcjw
fix available via ‘npm audit fix‘
node_modules/brace-expansion
node_modules/typedoc/node_modules/brace-expansion
```

Impact Webapps built using this library may be vulnerable to regex-based denial of service attacks. This does not currently affect any in-scope code.

Recommendation Running

```
npm audit fix
```

resolves this issue.

Consider also running `npm audit` within GitHub CI at a regular cadence to ensure that no new vulnerability disclosures are missed.

Developer Response The developers have updated the `package-lock.json` to fix the outdated dependencies, and run a CI job that runs `npm audit fix`.

Updated Veridise Response Running `npm audit fix` in the CI only fixes the dependencies in the CI job, and does not push those changes to the repository. Instead, you should run `npm audit` so that any vulnerabilities in dependencies will cause the CI job to fail.

Updated Developer Response The developers now run `npm audit` in the CI job.

A Intended Behavior: Non-Issues of Note

A.0.1 V-LMN-APP-INFO-001: Upgrade reorders fields

Severity	Warning	**Commit**	0f92fd1
Type	Logic Error	**Status**	Intended Behavior
Location(s)	`contracts/src/pool/PoolFactory.ts:107-123`		

The new `PoolFactory` contract has a different set of state fields than in the previously audited version (at commit `4ac8e2bee4806c1573f600cae3e55f824ef6c486`). Noting that a `PublicKey` is represented by two `Fields`, the below two code snippets show that

1. The field at index `0` remains the same.
2. The fields at indices `1` and `2` become the `protocol`.
3. The fields at indices `3` and `4` become the `delegator`.
4. The fields at indices `6` and `7` become unused.

```
export class PoolFactory extends TokenContract {

    @state(Field) approvedSigner = State<Field>();
    @state(PublicKey) owner = State<PublicKey>();
    @state(PublicKey) protocol = State<PublicKey>();
    @state(PublicKey) delegator = State<PublicKey>();
```

Snippet A.1: Definition of `PoolFactory` state at previous commit.

```
export class PoolFactory extends TokenContract implements PoolFactoryBase {

    @state(Field)
    approvedSigner = State<Field>()

    @state(PublicKey)
    protocol = State<PublicKey>()

    @state(PublicKey)
    delegator = State<PublicKey>()
```

Snippet A.2: Definition of `PoolFactory` state at current commit.

When the verification key for the `PoolFactory` changes, the interpretation of its state will change. However, the values stored *in* that state will not change. This means that, after the upgrade, the `protocol` and `delegator` will be set incorrectly.

Impact If this goes unnoticed, then pool fees may go to the former owner and the pools may delegate their stake to the former `protocol` address. This may cause the *actual* `protocol` and `delegator` to miss out on funds they are due.

Fortunately, this can be mitigated by immediately having the required signers update the protocol and delegator to their correct values.

Recommendation First, consider adding an `initialize()` method to `PoolFactory` which can only be called once per verification key change, and is called after the verification key changes inside of `PoolFactory.updateVerificationKey()`. For future updates, this will ensure that any custom upgrade logic can be executed atomically with the upgrade.

When performing the current upgrade, first obtain valid signatures from the approved signers to ensure that the protocol will be able to set its `protocol` and `delegator` addresses to the correct values. Perform these operations in a single transaction.

In the future, pay special attention to changes in the layout of the state variables and update those state values accordingly.

Developer Response This is a new deployment and there are no future plans for changes to the state order. I'm not sure that implementing complex mechanics for this data is necessary.

Updated Veridise Response We have downgraded this issue to a warning as the project is being redeployed rather than upgraded. Additionally, we note that users should migrate funds from the old protocol to the new one. Once all funds are moved, we recommend the Lumina developers deactivate the old version of the protocol to prevent accidental deposits into the wrong pools.

If there are no plans for updating state in the future, no initialization function is needed.

B Verification Keys

B.1 Latest Fix Commit

Verification Keys. The Veridise analysts compiled the in-scope contracts using at commit `4e53d8ff`. See the script in Section B.3 to understand how the contracts were compiled.

```
VKey Hashes for network local:
PoolFactory             :
    2716789211430794631122080148122680839978646990806151225230774417479638575632 9
Pool                    :
    7480901441026468595703278519003423509807923435933557767337375565636818933806
PoolTokenHolder         :
    2277113866768662823113103428963904572155442178688260510714583501827399824955 0

VKey Hashes for network testnet:
PoolFactory             :
    2716789211430794631122080148122680839978646990806151225230774417479638575632 9
Pool                    :
    7480901441026468595703278519003423509807923435933557767337375565636818933806
PoolTokenHolder         :
    2277113866768662823113103428963904572155442178688260510714583501827399824955 0

VKey Hashes for network mainnet:
PoolFactory             :
    9721332103283856545562845124352659001038504157365708280445560991150188191608
Pool                    :
    3299583463397083593714557042003388888155095734726973619427877840682642099496
PoolTokenHolder         :
    1211292691381238393535320728386915167881304537469373761555894384207073131340 0

VKey Hashes for network devnet:
PoolFactory             :
    2716789211430794631122080148122680839978646990806151225230774417479638575632 9
Pool                    :
    7480901441026468595703278519003423509807923435933557767337375565636818933806
PoolTokenHolder         :
    2277113866768662823113103428963904572155442178688260510714583501827399824955 0
```

Snippet B.1: verification key hashes

B.2 Audit Commit

Verification Keys. The Veridise analysts compiled the in-scope contracts using at commit `0f92fd1d`. See the script in Section B.3 to understand how the contracts were compiled.

```
VKey Hashes for network local:
```

```
PoolFactory             :
    22716212087851435454450612291355903703567402262897361027871637961450907310944
Pool                    :
    74809014410264685957032785190034235098079234359335577673373755656368189338 06
PoolTokenHolder         :
    22771138667686628231131034289639045721554421786882605107145835018273998249550

VKey Hashes for network testnet:
PoolFactory             :
    22716212087851435454450612291355903703567402262897361027871637961450907310944
Pool                    :
    74809014410264685957032785190034235098079234359335577673373755656368189338 06
PoolTokenHolder         :
    22771138667686628231131034289639045721554421786882605107145835018273998249550

VKey Hashes for network mainnet:
PoolFactory             :
    43661296992681061905470348574546346491047845126231706313511235045847803793 51
Pool                    :
    32995834633970835937145570420033888881550957347269736194278778406826420994 96
PoolTokenHolder         :
    12112926913812383935353207283869151678813045374693737615558943842070731313400

VKey Hashes for network devnet:
PoolFactory             :
    22716212087851435454450612291355903703567402262897361027871637961450907310944
Pool                    :
    74809014410264685957032785190034235098079234359335577673373755656368189338 06
PoolTokenHolder         :
    22771138667686628231131034289639045721554421786882605107145835018273998249550
```

Snippet B.2: verification key hashes

B.3 Script.

The Veridise analysts compiled the in-scope contracts using the below script.

```
import { Mina, NetworkId, Provable, SmartContract } from 'o1js';
import { Pool } from '../../pool/Pool.js';
import { PoolFactory } from '../../pool/PoolFactory.js';
import { PoolTokenHolder } from '../../pool/PoolTokenHolder.js';

async function compileAndPrintVKey<T extends typeof SmartContract>(contract: T) {
    const {verificationKey} = await contract.compile();
    const maxNameLength = 24; // 'FarmRewardTokenHolder'.length
    const paddedName = contract.name.padEnd(maxNameLength, ' ');
    Provable.log(`${paddedName}:`, verificationKey.hash);
}

async function compileAndPrintAllVKeys() {
    const allContracts = [
        PoolFactory,
```

```
        Pool,
        PoolTokenHolder,
    ];
    for(const contract of allContracts) {
        await compileAndPrintVKey(contract);
    }
}


const networkUrls: Record<string, string> = {
    // Mina networks
    // https://docs.minaprotocol.com/mina-protocol#the-mina-protocol
    // https://docs.minaexplorer.com/minaexplorer/graphql-getting-started
    mainnet: 'https://graphql.minaexplorer.com',
    // https://docs.minaexplorer.com/minaexplorer/berkeley-testnet
    testnet: 'https://berkeley.graphql.minaexplorer.com/',
    // https://docs.minaexplorer.com/minaexplorer/testnet
    devnet: 'https://devnet.graphql.minaexplorer.com/',

    // Zeko networks
    // https://docs.zeko.io/for_end_users#adding-to-auro-wallet
    'zeko:devnet': 'https://devnet.zeko.io/graphql',
};

async function compileAndPrintAllVKeysForAllNetworks() {
    const networkIds: NetworkId[] = [
        { custom: 'local' },
        'testnet',
        'mainnet',
        { custom: 'devnet' },
        { custom: 'zeko:devnet' },
    ];

    for (const networkId of networkIds) {
        let Network;

        const idString = typeof networkId === 'string' ? networkId : networkId.custom
    ;
        if (idString === 'local') {
            Network = await Mina.LocalBlockchain({ proofsEnabled: true });
        } else {
            const graphqlUrl = networkUrls[idString];
            if (!graphqlUrl) {
                console.error(`No GraphQL URL defined for network ${idString}`);
                continue;
            }
            Network = await Mina.Network({
                networkId,
                mina: graphqlUrl,
            });
        }

        Mina.setActiveInstance(Network);
```

```
        console.log(‘VKey Hashes for network ${idString}:‘);
        await compileAndPrintAllVKeys(); // make sure this is defined
        console.log("");
    }
}

compileAndPrintAllVKeysForAllNetworks();
```

Snippet B.3: Script used to generate verification keys

Glossary

AMM Automated Market Maker. 1

Mina Mina Protocol is a succinct 22KB blockchain utilizing zero-knowledge proofs. See `https://minaprotocol.com` for more details. 2, 26

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

Semgrep Semgrep is an open-source, static analysis tool. See `https://semgrep.dev` to learn more. 5

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, 26

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. 26

zkApp A smart contract written for the Mina blockchain. See `https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks` for more. 5, 26