Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

MinaNFT

Proof of NFT

NFT Standard

Veridise Inc.
March 31, 2025

► Prepared For:

Mina Foundation
https://www.minafoundation.com/

► Prepared By:

Aayushman Thapa Magar
Benjamin Sepanski
Mark Anthony

► Contact Us:

contact@veridise.com

► Version History:

Mar. 31, 2025	V2
Mar. 21, 2025	V1
Mar. 19, 2025	Initial Draft

Contents

1 Executive Summary

From Mar. 3, 2025 to Mar. 14, 2025, Mina Foundation engaged Veridise to conduct a security assessment of their NFT Standard. The security assessment covered the o1js smart contract implementation of the core NFT contracts, as well as several default implementations of extension contracts. Veridise conducted the assessment over 6 person-weeks, with 3 security analysts reviewing the project over 2 weeks on commits `06506ba` - `e329d79`. The review strategy involved a tool-assisted analysis and thorough code review of the program source code performed by Veridise security analysts.

Project Summary. The security assessment focused on the core NFT contracts. In the Mina standard, each NFT is represented as a zkApp stored in a unique Mina `Account`. Each NFT's metadata and owner are stored in its `Account`'s `appState`. Individual NFTs are managed by another zkApp called a `Collection`.

Each `NFT` contract is bound to its `Collection` by its `tokenId`. More precisely, a `NFT` contract is a member of a `Collection` exactly when the `NFT` `Account`'s `tokenId` matches the derived `tokenId` of the `Collection` `Account`*. As a consequence, all `NFT` methods (minting, transferring, and updating state) must be performed through the `Collection` zkApp.

The NFT Standard enables a large level of customization. The `Collection` interacts with a fully customizable `Admin` contract, responsible for defining when `NFT`s can be minted, setting `NFT` transfer fees, permanently ending minting, handling contract upgrades, and configuring both `Collection` metadata and individual `NFT`s. The standard includes a default `Admin` implementation in which most configurations are immutable, and a centralized entity approves actions like pausing and resuming the contract.

Additional features allow arbitrary contract logic to own/transfer/approve upgrades for individual `NFT`s, as well as per-`NFT` custom logic to programmatically update `NFT` ownership. These features are highly configurable, allowing `NFT` creators to disable the usage of unwanted features through immutable, per-`NFT` feature-flags and optionally requiring `Admin`-contract approval on transfers.

Code Assessment. The Mina Foundation developers provided the source code of the NFT Standard contracts for the code review. The source code appears to be mostly original code written by the Mina Foundation developers. It contains documentation in the form of READMEs and thorough documentation comments on functions and storage variables. To facilitate the

* The terminology of `tokenId`s can be confusing. The Mina docs provide an overview of these concepts, but for completeness we briefly describe them here.

Mina `Account`s have two `tokenId`s of interest. Firstly, each Mina `Account` is uniquely identified by a public key and a `tokenId`. This `tokenId` is often called the `Account`'s *own* `tokenId`. Default Mina `Account`s (which hold MINA balances) have a `tokenId` of one. Secondly, each Mina `Account` has a *derived* `tokenId`. This value is a (cryptographically) unique value associated to the `Account`, and may be used as *different* `Account`s' own `tokenId`. Updates to `Account`s must always follow the `tokenId` derivation chain, enforcing a kind of child-parent relationship between the "child" `Account` whose own `tokenId` is equal to the derived `tokenId` of its "parent" `Account`.

Veridise security analysts' understanding of the code, the NFT Standard developers also shared a web interface for interacting with deployed `Collections`. Additionally, they provided several example use cases/extensions of the NFT standard such as auctions, marketplaces, and advanced admin contracts.

The source code contained a test suite, which Veridise security analysts noted was high-quality. They provided crucial insight into how contracts are configured and interact with each other. They provided a good understanding of the contract logic, function flows, and dependencies, which considerably aided the security review process, and helped with proof-of-concept development. However, mosts of the tests only use expected values, without checking for improper or unexpected input. Furthermore, no specific errors are explicitly tested. The current test suite only verifies whether a transaction succeeds when called with expected arguments.

The test coverage for `src/contract` is extensive, with 99.78% statement coverage and 98.57% function coverage, showing that most of the contract logic is tested. However, branch coverage is lower at 45.29%. For `src/interfaces`, the coverage is lower, with 91.1% of the statement and only 48.57% of the functions tested, while the branch coverage is at 54.54%.

Summary of Issues Detected. The security assessment uncovered 24 issues, 3 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, missing checks on flags allowed bypassing admin approval on transfers (V-MNFT-VUL-001) and changing the approved sender when disallowed (V-MNFT-VUL-002), and oracle checks required by `NFT` updates could be ignored (V-MNFT-VUL-003). The Veridise analysts also identified 2 medium-severity issues, including missing state updates (V-MNFT-VUL-004) and possible disallowed `NFT`-minting in case of a malicious upgrade (V-MNFT-VUL-005), as well as 6 low-severity issues, 11 warnings, and 2 informational findings. The Mina Foundation fixed 23 issues and provided a partial fix to V-MNFT-VUL-011, leaving only the inherent centralization risks of the protocol described in the issue.

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

Documentation. The documentation of the NFT Standard is extensive and thorough, with each important field and function documented. There are a few places, however, where additional specificity is important. The Veridise analysts believe it is especially crucial for future developers implementing the standard, and future users/auditors evaluating an instantiation of the standard, to clearly understand the exact purpose of each flag, the guarantees provided by the fee structure, and the responsibilities of both custom admin contracts and off-chain update logic. The analysts' recommended additions are listed out in detail in V-MNFT-VUL-023. While this is an informational issue, the Veridise team strongly advises all recommendations be taken to ensure the standard is properly used.

Checks for Users. Users should be especially careful when approving a contract as a sender, or transferring ownership of an NFT to a contract. As mentioned in V-MNFT-VUL-011, transfer via signature cannot be turned off in the standard. Instead, users must validate that the contract's access permissions are immutably set to proof-only, and accept the risk of NFT theft after a Mina hard fork.

Testing Configuration Flags. All of the high and critical-severity issues came from missing checks on flags. Given the highly-configurable nature of this protocol, a missing check on one of several flags is easy to miss. Consider adding a test for each flag ensuring that each mutability restriction causes a failure in the expected functions.

Publishing vkeys and permissions. All instantiations of the NFT Standard should use the same verification key for their `NFTs`, and the same verification key for their `Collections`. Additionally, the `Collection` permissions should be configured as specified in its `deploy()` function. To help users and developers easily validate that a project has used the constructor factory pattern correctly when interacting with and building new NFTs, consider publishing the expected verification key and permissions for the `Collection` and `NFT` contracts.

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	Audited Version	Final Version	Type	Platform
NFT Standard	`06506ba - e329d79`	`8e13c6a5`	o1js	Mina

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 3–Mar. 14, 2025	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	2	2	2
Medium-Severity Issues	2	2	2
Low-Severity Issues	6	6	5
Warning-Severity Issues	11	11	11
Informational-Severity Issues	2	2	2
TOTAL	24	24	23

Table 2.4: Category Breakdown.

Name	Number
Data Validation	9
Maintainability	4
Logic Error	3
Access Control	3
Usability Issue	2
Authorization	1
Missing/Incorrect Events	1
Under-constrained Circuit	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of the NFT Standard. During the assessment, the security analysts aimed to answer questions such as:

- Is there sufficient access control implemented for critical actions within the NFT Collection?
- Can the admin or oracle approval requirements for actions be bypassed?
- Can a non-authorized person perform privileged actions?
- Is NFT state updated correctly?
- Are the relevant NFT state flags and permissions verified properly?
- Can actions for an NFT be locked, causing denial of service?
- Are there any usability concerns with respect to integrating the NFT Standard?
- Are the transfer fees and royalty fees implemented correctly?
- Are NFTs unique and associated to a unique `Collection`?
- How can malicious action by the `Admin` harm NFT owners or creators?
- Is the project susceptible to centralization risks? And if so, is there sufficient documentation that informs a user regarding the same?
- How does use of the constructor factory pattern affect security of the protocol?

In addition, during the assessment, the security analysts also aimed to verify if the code is vulnerable to any common o1js-specific vulnerabilities, such as:

- Under-constrained or over-constrained circuits
- Unsafe zkApp permissions
- Arithmetic overflows leading to denial of service
- Insufficient input parameter validation
- Inability to receive funds when needed
- Ability to attach `AccountUpdates` in unexpected portions of the update tree

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 tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged the open-source tools `npm audit`, Semgrep, and `eslint`. These tools are designed to find known issues in dependencies and common vulnerabilities in JavaScript programs.

Scope. The scope of this security assessment is limited to the following folders of the source code provided by the NFT Standard developers, which contains the smart contract implementation of the Mina NFT Standard:

- `silvana-lib/packages/nft/src/contracts/`
- `silvana-lib/packages/nft/src/interfaces/`
- `silvana-lib/packages/nft/src/util/div.ts`
- `silvana-lib/packages/storage/src/storage.ts`

Methodology. Veridise security analysts reviewed the reports of previous audits for NFT Standard, inspected the provided tests, and read the NFT Standard documentation. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise security analysts regularly met with the NFT Standard developers to ask questions about the code. The Veridise security analysts also perused the shared documentation for the NFT Standard which included references for testing.

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-MNFT-VUL-001	Admin approval for transfers can be bypassed	Critical	Fixed
V-MNFT-VUL-002	Approving a delegate address does not . . .	High	Fixed
V-MNFT-VUL-003	Oracle approval for updates can be bypassed	High	Fixed
V-MNFT-VUL-004	isPaused is not updated when updating . . .	Medium	Fixed
V-MNFT-VUL-005	Maliciously upgraded NFTs may mint new . . .	Medium	Fixed
V-MNFT-VUL-006	Oracle missing in equality check	Low	Fixed
V-MNFT-VUL-007	provedState in initialize method will . . .	Low	Fixed
V-MNFT-VUL-008	Permanently paused NFTs can be minted	Low	Fixed
V-MNFT-VUL-009	Admin may change minted NFT . . .	Low	Fixed
V-MNFT-VUL-010	Approved may be set when collection is . . .	Low	Fixed
V-MNFT-VUL-011	Centralization Risk	Low	Partially Fixed
V-MNFT-VUL-012	o1js best practices	Warning	Fixed
V-MNFT-VUL-013	Duplicate and unused program constructs	Warning	Fixed
V-MNFT-VUL-014	Incorrect URI/Symbol access control	Warning	Fixed
V-MNFT-VUL-015	Change of owner/admin does not use two . . .	Warning	Fixed
V-MNFT-VUL-016	MintParams fee/tokenId unused	Warning	Fixed
V-MNFT-VUL-017	Transfer event emitted twice	Warning	Fixed
V-MNFT-VUL-018	Missing checks in Admin.deploy()	Warning	Fixed
V-MNFT-VUL-019	Pausability of the collection and admin are . . .	Warning	Fixed
V-MNFT-VUL-020	Admin may deploy unusable NFT vkey	Warning	Fixed
V-MNFT-VUL-021	Unused Imports	Warning	Fixed
V-MNFT-VUL-022	from param unused in transfer functions	Warning	Fixed
V-MNFT-VUL-023	Typos and missing/incorrect comments	Info	Fixed
V-MNFT-VUL-024	Recommended contract factory validations	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MNFT-VUL-001: Admin approval for transfers can be bypassed

Severity	Critical	**Commit**	06506ba
Type	Authorization	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	Collection.transferByProof()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/18		

The function `transferByProof()` can be used to transfer ownership using a proof, in case the NFT `owner` or the `approved` address is a contract.

This method is missing an important validation. As highlighted in the snippet below, it does not verify that `CollectionData.requireTransferApproval` is false. Therefore, the admin approval requirement can be bypassed.

```
@method async transferByProof(params: TransferParams): Promise<void> {
  const { address, from, to, price, context } = params;
  const collectionData = CollectionData.unpack(
    this.packedData.getAndRequireEquals()
  );
  collectionData.isPaused.assertFalse(CollectionErrors.collectionPaused);
  // Veridise - Missing check which validates that admin approval is not required.
    //[...elided]

  const canTransfer = await approvalContract.canTransfer(transferEvent);
  canTransfer.assertTrue();
}
```

Snippet 4.1: Snippet from `transferByProof()`. Note that the `approvalContract` is either the `owner` or `approved` spender of the NFT, not the `admin`.

Impact NFTs can be transferred without approval from the admin, even if admin approval is required.

Without this check, an attacker may bypass all custom transfer logic enforced by the admin contract. For example, admins could not whitelist or blacklist accounts.

Recommendation Add an assert in the mentioned function which verifies `CollectionData.requireTransferApproval` is false.

Developer Response The developers assert that `requireTransferApproval` is false, as recommended.

4.1.2 V-MNFT-VUL-002: Approving a delegate address does not verify if changing approval is allowed

Severity	High	**Commit**	06506ba
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/nft.ts`		
Location(s)	NFT.approveAddress()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/19		

The function `approveAddess()` can be used to set or change the NFT's `approved` address for delegated actions. See snippet below for details.

```
  @method.returns(PublicKey)
  async approveAddress(approved: PublicKey): Promise<PublicKey> {
    const data = NFTData.unpack(this.packedData.getAndRequireEquals());
    data.isPaused.assertFalse(NftErrors.nftIsPaused);
    data.approved = approved;
    this.packedData.set(data.pack());
    this.emitEvent("approve", approved);
    return data.owner;
  }

  class NFTData extends Struct({
    //[elided]..
    /** Specifies if the NFT's approved address can be changed (readonly). */
  canApprove: Bool, // readonly
    //[elided]..
})
```

Snippet 4.2: Snippet from `approveAddress()`

However, this function is missing an important check. The `NFTData` contains a field `canApprove` which is used to specify if the NFT's `approved` address can be changed. This field is not validated within this function. So, the `approved` address can be set even when `canApprove` is false. See snippet below for context.

```
  class NFTData extends Struct({
        //[elided]..
      /** Specifies if the NFT's ownership can be transferred (readonly). */
      canTransfer: Bool, // readonly
        /** Specifies if the NFT's approved address can be changed (readonly). */
      canApprove: Bool, // readonly
        //[elided]..
})
```

Snippet 4.3: Snippet from `NFTData`

Impact A user can set the `approved` address to another account or smart contract, even if the collection creator specifically disallowed it.

For example, a collection which intends a marketplace to immutably be the `approved` address may rely on the `canApprove` flag. An attacker could potentially reset the `approved` public key, preventing the market from transferring the NFT when appropriate.

Recommendation Assert that `canApprove` is true before approving an address for delegation of actions.

Developer Response The developers assert that `canApprove` is true before approving an address, as recommended.

4.1.3 V-MNFT-VUL-003: Oracle approval for updates can be bypassed

Severity	High	**Commit**	e329d79
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	Collection.update()		
Confirmed Fix At	`https://github.com/SilvanaOne/silvana-lib/pull/26`		

In the `Collection` contract, the methods `updateWithOracle()` can be called to update a particular NFT with admin and oracle approval. The proof provided as input to the method contains an optional `oracleAddress` which can be used to link the NFT update with the network and accounts state.

Similarly, the method `update()` takes in a proof as input and updates the NFT without approval from the oracle. See snippet below for details.

```
@method async update(
  proof: NFTUpdateProof,
  vk: VerificationKey
): Promise<void> {
  await this._update(proof, vk);
}
```

Snippet 4.4: Snippet from `update()`

Here, the `update()` function does not validate that the proofs publicly input `oracleAddress` is empty. Therefore, it can be called with a proof containing an `oracleAddress`, to bypass the oracle approval and validations.

Impact An NFT can be upgraded with approval from the admin, but without the additional approval from the oracle. This would enable an attacker to bypass critical validations and checks performed by the oracle in relation to the network state.

Recommendation In the `update()` method, check that the proof's publicly input oracle address is empty.

Developer Response The developers validate the `proof.publicInput.oracleAddress` is empty when calling `update()` instead of `updateWithOracle()`.

4.1.4 V-MNFT-VUL-004: isPaused is not updated when updating the NFT state

Severity	Medium	**Commit**	06506ba
Type	Data Validation	**Status**	Fixed
File(s)	packages/nft/src/contracts/nft.ts		
Location(s)	NFT.update()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/27		

The `NFT.update()` method is used to update the NFT's state. The NFT contract's state includes `packedData`, which contains `isPaused`. This is used to verify whether the NFT is currently paused or not. The `output` parameter for `NFT.update()` method represents the desired new state of the NFT, after the updates have been made. However, `output.isPaused` is not saved to NFT data, causing the desired value for `isPaused` to not be saved to the NFT state.

```
@method.returns(Field)
async update(
  input: NFTState,
  output: NFTState,
  creator: PublicKey
): Promise<Field> {
      //[...elided]
      // Veridise - Missing mechanism to save new value for isPaused to data.
  data.owner = output.owner;
  data.approved = output.approved;
  data.version = output.version;

  this.packedData.set(data.pack());
      //[...elided]
}
```

Snippet 4.5: Snippet from `update()`

Impact No call to `update()` may change whether the NFT is paused or not.

Depending on the NFT implementation, this may prevent pausing in emergency scenarios.

Recommendation Add the new value for `isPaused` into NFT data.

Developer Response The developers now assign `output.isPaused` as `data.isPaused` similar to the other NFT states.

4.1.5 V-MNFT-VUL-005: Maliciously upgraded NFTs may mint new NFTs

Severity	Medium	**Commit**	e329d79
Type	Data Validation	**Status**	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	Collection		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/38		

Since an AccountUpdate's children may inherit its token ID, a malicious NFT implementation could approve arbitrary AccountUpdates to create new Accounts with the Collection's tokenId. This is impossible with the default NFT verification key. However, if a malicious NFT upgrade occurs, an attacker could use this ability to mint arbitrary NFTs to the Collection.

Impact If an admin fails to properly validate an upgrade, or owners are allowed to upgrade their owned NFTs arbitrarily, malicious actors may mint NFTs without permission from the admin or creator.

Recommendation Ideally, the Collection would validate that NFT updates have no children of their own, as in the expected implementation of the NFT. Doing this is not easy with the current o1js APIs. It would require iterating over the o1js AccountUpdateLayout to find the child update representing an NFTAccountUpdate as an entry in the MerkleList of children, and then validating that it itself has no children.

Instead, consider validating the NFT verification key hash matches the expected one on each @method call.

Additionally, consider setting the NFT receive permissions to Permissions.impossible() to ensure rogue NFT implementations cannot increase the balance on their account.

Developer Response I agree that validating AccountsUpdates is not easy, so the fix checks the verification key on the upgrade and sets Permission.impossible() for NFT. It should be noted that the balance is a symbolic value on the NFT account, as NFT is represented by the account state and not the balance. Still, having the wrong balance is a problem for off-chain services such as Explorers.

4.1.6 V-MNFT-VUL-006: Oracle missing in equality check

Severity	Low	**Commit**	06506ba
Type	Logic Error	**Status**	Fixed
File(s)	packages/nft/src/interfaces/types.ts		
Location(s)	NFTState.assertEqual()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/20		

The function `NFTState.assertEqual()` checks that two `NFTState` instances are exactly equal. It does this by asserting equality of each field of the two structs. However, the `oracleAddress` field is left out of this function. Consequently, if two `NFTStates` `a` and `b` are exactly equal except `a.oracleAddress != b.oracleAddress`, `NFTState.assertEqual(a,b)` will not cause an assertion failure.

Fortunately, this function is only called once in the in-scope portion of the codebase: inside `NFT.update()`. Immediately after it is invoked, the `oracleAddress` is checked to equal the desired value.

```
NFTState.assertEqual(
    input,
    new NFTState({
      // [VERIDISE] elided...
      oracleAddress: input.oracleAddress,
    })
  );

// assert that the read-only fields are not changed
input.creator.assertEquals(output.creator);
NFTTransactionContext.assertEqual(input.context, output.context);
input.oracleAddress.assertEquals(output.oracleAddress);
```

Snippet 4.6: Snippet from `NFT.update()`

Impact The singular in-scope call-site cannot be exploited due to the extra check. However, this may make the code more difficult to read and maintain.

Further, out-of-scope usage of this function may lead to errors in implementations of NFT update contracts. For example, consider the `merge()` function provided in the `NFTGameProgram` example:

```
merge: {
  privateInputs: [SelfProof, SelfProof],
  async method(
    initialState: NFTState,
    proof1: SelfProof<NFTState, NFTState>,
    proof2: SelfProof<NFTState, NFTState>
  ) {
    proof1.verify();
    proof2.verify();
    NFTState.assertEqual(initialState, proof1.publicInput);
    NFTState.assertEqual(proof1.publicOutput, proof2.publicInput);
    return {
      publicOutput: proof2.publicOutput,
    };
  },
},
```

Snippet 4.7: Definition of `merge()`

A malicious prover could use an arbitrary `oracleAddress` when creating `proof1`, then switch back to `initialAddress.oracleAddress` when creating `proof2` to pass the check in `NFT.update()`.

Recommendation Assert the two `oracleAddresses` are equal when asserting equality of `NFTStates`.

Developer Response The developers now assert that the two oracle addresses are equal, as recommended.

4.1.7 V-MNFT-VUL-007: provedState in initialize method will always be false

Severity	Low	**Commit**	06506ba
Type	Logic Error	**Status**	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	Collection.initialize()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/25		

The function this.account.provedState in Collection.initialize() will always be false, as only 7 app-state fields are being set by a proof:

- 1 for collectionName (Field)
- 2 for creator (PublicKey)
- 2 for admin (PublicKey)
- 1 for baseURL (Field)
- 1 for packedData (Field).

Consequently, the check in initialize() will not prevent an attacker from calling initialize() multiple times.

```
@method
async initialize(masterNFT: MintParams, collectionData: CollectionData) {
  this.account.provedState.requireEquals(Bool(false));
  // [elided]..
}
```

Snippet 4.8: Snippet from initialize()

Impact Fortunately, Collection.initialize() still cannot be called twice. initialize() internally calls the Collection._mint() method, which asserts that the newly minted master NFT is a new Account.

```
async _mint(params: MintParams): Promise<MintEvent> {
      const {name, address, data, metadata, storage, metadataVerificationKeyHash,
    expiry, } = params;
            // [elided]..
      update.account.isNew.getAndRequireEquals().assertTrue("Is new failed");
            // [elided]..
```

Snippet 4.9: Snippet from _mint()

Future iterations of this standard may become susceptible to repeated initialization.

Further, protocol users will not be able to rely on provedState to check if initialize() has been called and the project properly initialized via proof.

Recommendation To ensure that this.account.provedState will be true and initialize method can only be called once, either this.init() can be called to initialize all 8 app-state fields to zero, or the final 8th state can be set to zero manually in the initialize method.

Developer Response The developers added a new state `pendingCreatorX`, taking the 8th field in the state in the `Collection`. All the state fields have now been initialized via proof.

Updated Veridise Response This resolves the issue by setting all 8 fields. Additionally, the Veridise analysts recommend:

1. Adding a comment to the function indicating that changes must be made if the number of state fields available on the Mina blockchain changes.
2. Change the implementations of `CollectionData.isPaused`, `CollectionData.requireTransferApproval`, and `CollectionData.mintingIsLimited` to use the expression `4 + 32 + 64`.

Updated Developer Response The developers have updated `isPaused`, `requireTransferApproval` and `mintingIsLimited` to calculate the bits properly as per recommendation, accounting for 4 flag bits due to `pendingCreatorIsOdd` being added. The comments have also been added on `initialize` and `deploy`.

4.1.8 V-MNFT-VUL-008: Permanently paused NFTs can be minted

Severity	Low	**Commit**	06506ba
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	_mint()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/21		

`Collection._mint()` takes in an authorized `MintParams` request and mints a new `NFT`. However, it does not perform any checks that the minted `NFT` is consistent with its own configuration.

While most state is trivially consistent with the `NFT`'s configuration, there is one notable exception. Some `NFT`s are unpausable, but nonetheless have an `isPaused` flag.

```
/** Specifies if the NFT contract can be paused, preventing certain operations (
    readonly). */
canPause: Bool, // readonly
/** Indicates whether the NFT contract is currently paused. */
isPaused: Bool,
```

Snippet 4.10: Snippet fron `NFTData` in `packages/nft/src/interfaces/types.ts`

As seen below, if an unpausable `NFT` is minted, it cannot be resumed.

```
  @method.returns(PublicKey)
  async resume(): Promise<PublicKey> {
    const data = NFTData.unpack(this.packedData.getAndRequireEquals());
    data.canPause.assertTrue(NftErrors.noPermissionToPause);
```

Snippet 4.11: Snippet from `NFT.resume()`

Impact A malicious or buggy `admin` may intentionally mint users unusable NFTs.

For example, suppose a `NFT` gives rights to vesting funds which can be redeemed after a certain time period, and expires if not eventually claimed. A scammer could mint paused, unpausable `NFT`s. These `NFT`s would be unusable, preventing users from claiming their funds before the `NFT` expires.

Recommendation Enforce the property that unpausable `NFT`s are not paused when minted.

Developer Response The developers now validate that unpausable `NFT`s are not set to paused during the minting process.

4.1.9 V-MNFT-VUL-009: Admin may change minted NFT address/owner

Severity	Low	**Commit**	06506ba
Type	Data Validation	**Status**	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	mint()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/22		

Minting an NFT within a `Collection` starts by specifying a `MintRequest`. This request consists of an NFT `address`, NFT `owner`, and some arbitrary `context` data provided to the `admin`.

However, as shown in the below snippet, the actual `_mint()` operation is performed based on the `mintParams` returned by the `admin`. Consequently, the minted NFT's `address` and `owner` may be unrelated to the `mintRequest`.

```
@method async mint(mintRequest: MintRequest): Promise<void> {
  // [VERIDISE] extra checks elided....
  const mintParams = (await adminContract.canMint(mintRequest)).assertSome(
    CollectionErrors.cannotMint
  );

  // [VERIDISE] extra checks elided....
  await this._mint(mintParams);
}
```

Snippet 4.12: Snippet from `Collection.mint()`

Impact A malicious or buggy `adminContract` may mint an NFT unrelated to an owner request.

For example, suppose some market mints an NFT in return for a user deposit, giving rights to withdraw the funds in the future. A malicious prover network may be able to replace the `mintParams.owner` with an address controlled by the prover network. The depositor will see a `mintRequest` for an NFT owned by the depositor, but may unknowingly submit a transaction minting an NFT to the malicious prover network.

Recommendation Either do not allow the `adminContract` to specify the NFT `address` and `owner`, or ensure the `mintParams` match the `mintRequest`.

Developer Response The developers now verify that the `address` and the `owner` in the `mintParams` match the `mintRequest`.

4.1.10 V-MNFT-VUL-010: Approved may be set when collection is paused

Severity	Low	Commit	06506ba
Type	Logic Error	Status	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	Collection.approveAddressByProof()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/28		

The `approveAddressByProof()` method is used to approve an address to transfer the NFT. This method is missing a validation to check if `CollectionData.isPaused` is `false`, which enables approving addresses even when the collection is paused.

```
  @method async approveAddressByProof(
    nftAddress: PublicKey,
    approved: PublicKey
  ): Promise<void> {
    // Veridise - Missing check which validates that collection is not paused.
          //[...elided]
    this.emitEvent("approve", new ApproveEvent({ nftAddress, approved }));
  }
```

Snippet 4.13: Snippet from `approveAddressByProof()`

Additionally, there are several other actions within the collection which can be performed while the collection is paused. These include `mintByCreator()`, `mint()`, `approveAddressByproof()`, `upgradeNFTVerificationKeyBySignature()`, `upgradeNFTVerificationKeyByProof()`, `upgradeVerificationKey()`, `pauseNFTBySignature()`, `pauseNFTByProof()`, `resumeNFT()` and `resumeNFTByProof()`. And `NFT.upgradeVerificationKey()` can be performed when an NFT is paused.

Out of these, the `mintByCreator()` is intended to work even with a paused collection to be able to mint the Master NFT which holds the collection metadata. But, there is no documentation which refers to the allowance or disallowance of the other mentioned actions, within a paused collection.

Impact Addresses may be set as approvers even when the contract is currently paused.

Transfers will still not be possible while the contract is paused. However, depending on the implementation of the owner contract, it may not be possible to reverse the unintended `approver` change.

The concerns mentioned for `approveAddressByProof()` also extend to the other actions mentioned in the description. Moreover, not documenting the behaviour of the mentioned actions can lead to misplaced assumptions and usability concerns for collection creators.

Recommendation Add validation to check if the contract is currently paused using `CollectionData.isPaused`.

Also, add documentation which lists the allowed and disallowed actions for a paused collection and/or NFT, and add validations to the aforementioned functions accordingly.

Developer Response The developers now check that the `Collection` is not paused when calling `approveAddressByProof()`, `upgradeNFTVerificationKeyBySignature()`, `upgradeNFTVerificationKeyByProof()`, `pauseNFTBySignature()`, `pauseNFTByProof()`, `resumeNFT()`, `resumeNFTByProof()`, `setRoyaltyFee()`, and `setTransferFee()`.

The refactoring allowed the developers to remove the `mintingIsLimited()` function, see V-MNFT-VUL-023.

4.1.11 V-MNFT-VUL-011: Centralization Risk

Severity	Low	**Commit**	06506ba
Type	Access Control	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/41		

Similar to many projects, Mina's NFT Standard defines several roles/contracts which are given special permissions or perform important validations for critical operations. The abilities of these entities and their trust assumptions are outlined below.

This issue starts by outlining the roles for the core contracts, `Collection` and `NFTs`, then the `NFTAdmin`. Then, it describes several items users should be careful of when implementing or using instances of the standard.

The core contracts rely on various contracts which may depend on the particular application. In particular, `NFT` owners, `NFT` approved spenders, `NFT` admins, and `NFT` oracle contracts may vary from `Collection` to `Collection` and are not specified here.

Importantly, after discussions with the developers, the Veridise analysts understand that ***the NFTStandardApproval, NFTStandardOwner, NFTStandardUpdate are templates that are not intended to be used as-is***, and are to be changed according to the case of the user. Veridise analysts did review these contracts and found no flaws, but they are highly centralized wrappers around a standard user accounts.

Protocol Contract Roles.

1. `Collection`:
   a) deployer: This is any party who may produce signatures for the `Collection.address`. The deployer has a highly privileged role, but only during deployment, initialization, and network upgrades. The deployer may perform any of the following actions:
      i) Set the permissions as specified during deployment.
      ii) Upgrade the `Collection` during a hard fork.
      iii) Initialize the `Collection` without permission of the `admin/creator`, allowing them to determine the entire `CollectionData` initial state and set the "master NFT".
   b) `creator`: The `Collection.creator` receives fees based on the `Collection`'s configured royalty and transfer fees, and may mint tokens. More specifically, the `creator`:
      i) Receives fees determined by the `Collection transferFee`, `NFT` transfer price, and `Collection royaltyFee`.
      ii) Prevent users from transferring funds by setting their `receive` permissions to `impossible`, causing fees to fail.
      iii) Mint `NFTs` when the contract is not paused, and minting for the `Collection` has not been limited (see the `admin`'s role below).
      iv) Upon permission from the `admin` (see below), transfer the `creator` role.

c) `admin`: The `Collection.admin` configures all of the `Collection` settings, including metadata, fees, and the paused status. The `admin` is intended to be a smart contract, whose implementation depends on the specific `Collection` instance. This smart contract may:

   i) Upgrade the `Collection`'s verification key to implement arbitrary logic.
   ii) Configure the collection's fees, name, and base URL.
   iii) Pause and un-pause the `Collection`, and individual `NFTs`.
   iv) Transfer `admin` rights to another account.
   v) Transfer the `creator` role, upon approval by the `creator`.
   vi) "Limit" `NFT` minting, i.e. permanently prevent future minting on this `Collection`.
   vii) Mint `NFTs` when the contract is not paused, and minting for the `Collection` has not been limited.
   viii) Restrict updates to `NFT`-data.
   ix) Restrict `NFT` transfers when the `Collection` is configured with `requireTransferApproval == true`.
   x) Upgrade `NFT` verification keys, with owner approval if required based on the `NFT`'s data.

2. `NFT`: `NFTs` (when used properly) are deployed directly by the `Collection`. Depending on their configuration when minted, there may still be some special roles with extra authority over the particular `NFT`:

   a) deployer: Whoever knows the private key may upgrade the `NFT` on hard forks.
   b) `owner`: The owner may
      i) transfer the `NFT` ownership based on signature or verification key (for `NFTs` with `canTransfer`)
      ii) set the `approved` address based on signature or verification key (for `NFTs` with `canApprove`)
      iii) prevent upgrading the `NFT`'s verification key for `Collections` with `isOwnerApprovalRequired`
   c) `approved`: An approved account may transfer the `NFT` ownership (for `NFTs` with `canTransfer`).
   d) `metadataVerificationKeyHash`: Anyone who can create a proof which verifies against the `metadataVerificationKeyHash` may update the `NFT` itself (contingent upon approval by the `Collection admin`). More precisely, the may:
      i) Edit `owner` or `approved` (for `NFTs` with `canChangeOwnerByProof`, regardless of `canTransfer` or `canApprove`)
      ii) Edit the `name`, `metadata`, `storage`, `isPaused`, or `metadataVerificationKeyHash` (for `NFTs` with `canChangeName`, `canChangeMetadata`, `canChangeStorage`, `canPause`, and `canChangeMetadataVerificationKeyHash`, respectively).
      iii) Set the `NFT` version arbitrarily high, causing denial-of-service.

Default Implementations.

1. `NFTAdmin`. This contract extends the class `NFTAdminBase` and serves as the foundational administrative layer for the NFT collection. The address of the `NFTAdmin` contract corresponds to the `Collection.admin`. It provides approval for critical functionalities within the collection such as NFT upgrades, pausing and resuming operations and ownership management. Note that this contract is upgradable, and therefore a malicious

admin can pose a signifiant threat to the collection. The contract has its own `admin`, which is required to sign off on various (but not all) approvals in the default implementation.

a) `admin`: This account may perform any of the following actions
   i) Upgrades the NFTAdmin's verification key.
   ii) Pause or resume the `NFTAdmin` contract.
   iii) Transfers ownership of the contract to a new admin.
   iv) Upgrade specific NFT verification keys (possibly with consent of the owner, if required).
   v) `canChangeRoyalty()` - Determines if the royalty fee can be changed for a Collection.
   vi) `canChangeTransferFee()` - Determines if the transfer fee can be changed for a Collection.
   vii) `canPause()` - Determines if the collection can be paused.
   viii) `canResume()` - Determines if the collection can be resumed.

b) deployer: The deployer is the public key used to deploy the NFT collection contract. It is responsible for
   i) Correctly configuring the verification key and permissions for the zkApp.
   ii) Upgrading the zkApp during hard forks.

Contracts providing approval for critical actions related to the NFT collection. The following contracts are provided as templates in the project and are not meant to be used as is. Instead a user deploying a collection should tailor them as per the requirements. But, these templates provide a good estimate of trust assumptions on the part of the collection. For the default implementations, the admin of the contract signs off on each permitted action, but the deployer can change the `VerificationKey` unprompted, and therefore it remains fully in control.

1. `NFTStandardApproval` - This contract provides approval for transfers by proof, if the owner of the NFT is a contract.
2. `NFTStandardOwner` - This contract is the default implementation of an NFT owner contract. It provides approval for critical NFT actions like pause, resume, approve, transfer and upgrade.
3. `NFTStandardUpdate` - This contract is a default implementation of the `oracle`. The `oracle` optionally provides approval for an NFT update.

Impact As a standard intended for broad use across several implementations, the precise impact of these centralization risks may be difficult to asses. Given this setting, the Veridise team wishes to highlight some specific risks based on the above centralization issues:

1. **Signature-based transfers**: Transfers via signature cannot be prevented for an `NFT`. This means that, for a third-party smart contract to truly own the `NFT`, their `access` permissions must be set to `proof`-only. Otherwise, whoever knows the private key may bypass the smart contract logic and transfer the `NFT` to themselves.
2. **Creator dependence on admin-set fees**: The `Collection` `admins` may set fees arbitrarily, including to zero.
3. **NFT owner dependence on admin-set fees**: The `Collection` `admin` may set fees arbitrarily high, preventing transfers.

4. **Use of "standard" contracts**: Implementers may use the standard owner, updater, or approver contracts.
5. **NFT update risks**: The `metadataVerificationKey` encodes logic which may arbitrarily update the `NFT` (up to mutability flags), even when paused. This may fully DoS the `NFT` by setting the version to `UInt32.MAXINT()`, preventing further transfers.
6. **Rogue NFT updates on hard-forks**: During a Mina hard-fork, the owner of an `NFT`'s private key may upgrade the verification key. If `Collection creators/admins` do not control these keys, it may lead to serious issues (see V-MNFT-VUL-005). Conversely, if `Collection creators/admins` lose control of these keys, upgrades may be prevented.
7. **Key loss / malicious action**: As always, centralized roles may offer promising targets for attackers, or be abused by role holders. Depending on the `admin` contract, this could include a full contract upgrade, targeted denial of service to `NFT` holders, or theft of `NFT`s.

Recommendation Some of these issues should be mitigated through both user- and developer-facing documentation.

1. **Signature-based transfers**: Users should validate contract permissions before trusting it with ownership of their `NFT`.
2. **Creator dependence on admin-set fees**: `NFT` creators should validate the `admin` contract has sufficient protections, or is operated by a trusted party, to prevent loss of fees.
3. **NFT owner dependence on admin-set fees**: `NFT` owners should validate the `admin` contract has sufficient protections, or is operated by a trusted party, to prevent prohibitively exorbitant of fees.
4. **Use of "standard" contracts**: `NFT` users should not use the standard contracts.

A few of the above issues may be mitigated by concrete action.

1. **NFT update risks**: Consider setting a maximum version increase for updates. Given the current Mina block time of several minutes, this will ensure the version limit is not reached before the next hard fork.

Finally, some problems are best mitigated through extensive care in the operational security practices taken when operating the specified roles.

1. **Rogue NFT updates on hard-forks**: `Collection admin/creators` should own and operate the keys of all `NFT`s, and carefully store them in a persistent manner (see operational-security guidance below).
2. **Key loss / malicious action**: All deployer, administrative, and `creator` roles should take care to follow security best practices (see below).

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

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

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

Developer Response The developers added a best practices section in the readme, along with `BEST_PRACTICES.md` . This documents best practices mentioned in the issue writeup.

Given that the collection creator is implementing his creative ideas by creating a collection, some centralization, reflecting the creator's role, should remain in the protocol. There have been attempts in MinaNFT V2 to create a decentralized collection where everyone can add NFT, but creators were very unhappy with it, and the developers have stopped this practice.

Additionally, the developers added the same doc on the documentation site: `https://docs.minanft.io/Documentation/v3/best_practices`

4.1.12 V-MNFT-VUL-012: o1js best practices

Severity	Warning	**Commit**	06506ba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/25, https://github.com/SilvanaOne/silvana-lib/pull/40, https://github.com/SilvanaOne/silvana-lib/pull/41, https://github.com/SilvanaOne/silvana-lib/pull/42,, e4744af, a1b55b5, 6356836		

Consider implementing the following o1js best practices:

1. **Avoid use of `Unsafe` APIs**:

   a) `packages/nft/src/util/div.ts`:

      i) `mulDiv()`: Invoking `Provable.witness(T, ...)` returns an arbitrary prover-supplied value. The only constraints on this value are imposed by calling `T.check()` to ensure the value satisfies the type invariants of `T`. In `mulDiv()`, when implementing the division algorithm, the quotient and remainder are provided by `Provable.witness(...)`, and then separately range-checked. As shown below, this requires using the `UInt64.Unsafe` API. Developers could avoid this by replacing `MulDivResultInternal` with a `Struct` which specifies `result` and `remainder` as `UInt64`s rather than `Field`s. This will ensure the `Struct.check()` function (which, by default, invokes `check()` on each of its fields) will perform the range checks automatically.

```
const fields = Provable.witness(MulDivResultInternal, () => {
   // Arbitrary prover code may be executed here
 });
 Gadgets.rangeCheck64(fields.result);
 Gadgets.rangeCheck64(fields.remainder);
 // other checks required for division correctness ...
 return {
   result: UInt64.Unsafe.fromField(fields.result),
   remainder: UInt64.Unsafe.fromField(fields.remainder),
 };
```

Snippet 4.14: Snippet from `mulDiv()`.

2. **Avoid using native o1js types in events.** Like `Field`, `UInt32`, `UInt64`, `PublicKey`, etc, so that the semantics of each event are clear.

   a) In the events defined in `Collection.events`, several make use of native o1js types, as can be seen in the snippet below.

```
    events = {
      update: PublicKey,
            //[...] elided
      upgradeVerificationKey: Field,
            //[...] elided
      ownershipChange: OwnershipChangeEvent,
      setName: Field,
      setBaseURL: Field,
      setRoyaltyFee: UInt32,
      setTransferFee: UInt64,
      setAdmin: PublicKey,
    };
```

Snippet 4.15: Snippet from `Collection.events`.

Impact Not following best practices may lead to projects with reduced "by default" security/usability, allowing simple errors to magnify into large mistakes.

Recommendation Follow the above o1js best practices.

Developer Response

1. The Unsafe usage has been documented clearly, and was kept to avoid reliance on an undocumented feature of `Provable.witness()`. Additionally, an optimization (using `UInt64.assertLessThan()`) was introduced.
2. The developer now uses custom event types for each event.

Additionally, the developers added some optimizations suggested by Veridise, including packing and unpacking optimizations. Further, the developers increased the version size to 64 bits.

Finally, the developers added documentation for the flags and metadata to the github repository and documentation site.

4.1.13 V-MNFT-VUL-013: Duplicate and unused program constructs

Severity	Warning	**Commit**	06506ba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/SilvanaOne/silvana-lib/pull/30`		

Description The following program constructs are unused or duplicate constructs:

1. `packages/storage/src/storage/storage.ts`:
   a) `Storage.isEmpty()`: This function effectively inlines both `Storage.equals()` and `Storage.empty()`.

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 implemented the recommendation, removing duplicate code.

4.1.14 V-MNFT-VUL-014: Incorrect URI/Symbol access control

Severity	Warning	**Commit**	06506ba, e329d79
Type	Access Control	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`, `packages/nft/src/contracts/admin.ts`		
Location(s)	Collection.deploy(), Admin.deploy()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/23, https://github.com/SilvanaOne/silvana-lib/pull/25		

`Collection.deploy()` defines the `AccountUpdate` which deployers are expected to use when deploying a `Collection`. The account permissions configured by `deploy()` are shown below.

The `setZkappUri` permission, which allows users to set the `Collection`'s `zkappUri`, and the `setTokenSymbol` permission, which allows users to change the `Collection`'s token symbol, are both set to `Permissions.none()`. This means any account update may change the `zkappUri` or the token symbol.

Fortunately, the `access` permission is set to `proof`, preventing any non-proof authorized `AccountUpdates` to the `Collection` account.

```
 this.account.permissions.set({
  ...Permissions.default(),
  setVerificationKey:
    Permissions.VerificationKey.proofDuringCurrentVersion(),
  setPermissions: Permissions.impossible(),
  access: Permissions.proof(),
  send: Permissions.proof(),
  setZkappUri: Permissions.none(),
  setTokenSymbol: Permissions.none(),
});
```

Snippet 4.16: Snippet from `Collection.deploy()`

The same issues exists in the `deploy()` function of the default `Admin` contract.

Impact Future versions of the protocol may introduce ways for users to submit their own, custom `AccountUpdates`, opening up the door to scams or introducing offensive content into a creator's `Collection`.

Recommendation Set the `setZkappUri` and `setTokenSymbol` permissions to be either impossible or controlled via signature/proof in both the `Collection` and `Admin` contracts.

Developer Response The recommended changes have been implemented in the `Collection` and the `Admin`.

4.1.15 V-MNFT-VUL-015: Change of owner/admin does not use two step pattern

Severity	Warning	**Commit**	06506ba
Type	Access Control	**Status**	Fixed
File(s)	`packages/nft/src/contracts/admin.ts, packages/nft/src/contracts/collection.ts`		
Location(s)	Admin.transferOwnership() , Collection.setAdmin()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/25		

When electing a new owner in the `Admin` contract, the current owner passes on the PublicKey of the new owner to the `transferOwnership()` method.

```
@method.returns(PublicKey)
async transferOwnership(to: PublicKey): Promise<PublicKey> {
    // [elided]..
  this.admin.set(to);
  // [elided]..
}
```

Snippet 4.17: Snippet from `transferOwnership()`

Similarly, the `setAdmin()` method is used when changing admin in `Collection` contract.

```
  @method
  async setAdmin(admin: PublicKey): Promise<void> {
  // [elided]..
    this.admin.set(admin);
    this.emitEvent("setAdmin", admin);
  }
```

Snippet 4.18: Snippet from `setAdmin()`

The ownership/adminship is immediately revoked and the new owner/admin has all the administrative privileges. Making such critical changes in a single step can be error prone and lead to irrecoverable mistakes.

Impact If an incorrect PublicKey is accidentally set as the owner, all the administrative privileges will be lost. In this case, actions such as pausing or resuming can no longer be performed.

Recommendation It is recommended to implement a two-step ownership transfer process, where the new owner must confirm the acceptance of ownership before ownership from the previous owner is revoked.

Developer Response The fix for `Admin` is as recommended, whereas the for `Collection`, a different approach is taken. Only the x coordinate of the public key is stored as a state (as `pendingCreatorX`) and a bool indicating if it is negative or not is stored in `collectionData.pendingCreatorIsOdd`.

Updated Veridise Response The `acceptOwnership()` function in the `Admin` contract should check that the pending admin is not the empty key. Otherwise, the Mina runtime will ignore the `AccountUpdate` and allow anyone to transfer ownership of the `Admin` contract to the empty public key.

This can be seen with the following test, which passes

```
it("Should accept ownership", async() => {
  const tx = await Mina.transaction(whitelistedUsers[2],
    async () => {
      await (<NFTAdmin>adminContract).acceptOwnership()
    }
  );
  await tx.prove();
  assert.strictEqual(
    (
      await sendTx({
        tx: tx.sign([whitelistedUsers[2].key]),
        description: "mint",
      })
    )?.status,
    expectedTxStatus
  );
})
```

The analysts recommend the following changes:

1. Instead of performing a 2-step transfer using two functions, consider transferring control in a single step by requiring both the current admin and the pending admin to sign the `transferOwnership()` transaction.
2. Make a similar change in the `Collection` contract.
3. Add in both positive and negatives tests for the `Admin` and `Collection` ownership transfers.

Updated Developer Response Unfortunately, it is not always possible to make the transfer in one step. There is no way for two wallet users to sign the same TX without exposing the private keys, as the wallet always signs with the tx sender key. Also, there will be significant issues with nonce and keeping the tx. Therefore, I would prefer to keep the process in two separate transactions.

Updated Veridise Response This resolves the issue for the `Collection`, but does not fix the issue in the `Admin` contract.

Updated Developer Response The developers now verify that the pending admin is not an empty public key in the `Admin` contract as well.

4.1.16 V-MNFT-VUL-016: MintParams fee/tokenId unused

Severity	Warning	**Commit**	06506ba
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	mint(), initialize(), mintByCreator()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/24		

`mint()` and `mintByCreator()` can be used to mint new NFTs within a `Collection`. Additionally, `initialize()` mints a "master NFT" when initializing the `Collection`. All rely on `MintParams` specified by either the `creator` or the `admin`.

In all three functions, the `MintParams.tokenId` and `MintParams.fee` values are unchecked.

```
/**
 * Represents the parameters required for minting a new NFT.
 */
class MintParams extends Struct({
  // [VERIDISE] elided other fields...

  /** The token ID of the NFT. */
  tokenId: Field,
  // [VERIDISE] elided other fields...

  /** The fee associated with minting the NFT. */
  fee: UInt64,
  // [VERIDISE] elided other fields...
}) {
```

Snippet 4.19: Definition of `MintParams`.

Impact Admin contracts will not be able to manage multiple `Collection`.

Fees may be missed or go unpaid.

Recommendation Assert that the returned `tokenId` matches the `Collection`'s derived token ID.

Remove `fee` from the `MintParams` struct.

Developer Response The developers made the following changes:

- They added a check for the tokenId.
- They added a fee and tokenId to MintEvent as this is required for indexing on minascan explore that keeps track of the NFT prices.

Updated Veridise Response After discussions with the developers, the Veridise understand that the fee is intended to be a part of the `MintParams` to link it to the mint request and have it be part of the `MintEvent`. This fee value is an arbitrary amount and an admin can choose to charge whatsoever they wish to put in the event.

The Veridise analysts recommend additionally documenting that this fee is fully admin-controlled to make this clear to implementers.

Updated Developer Response The developers added the requested docs.

4.1.17 V-MNFT-VUL-017: Transfer event emitted twice

Severity	Warning	**Commit**	06506ba
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	Collection.approvedTransferBySignature()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/31		

The `approvedTransferBySignature()` event emits a transfer event. However, `_transfer` emits the same event.

```
@method async approvedTransferBySignature(
      params: TransferParams
    ): Promise<void> {
      // [VERIDISE] elided...
      const transferEvent = await this._transfer({
        transferEventDraft,
        transferFee: collectionData.transferFee,
        royaltyFee: collectionData.royaltyFee,
      });
      // [VERIDISE] elided...
      this.emitEvent("transfer", transferEvent);
    }
```

Snippet 4.20: Snippet from `approvedTransferBySignature()`

Impact Off-chain listeners may incorrectly think multiple transfers occurred.

Recommendation Remove the second event emission.

Developer Response The developers removed the second event emission in `approvedTransferBySignature()`.

4.1.18 V-MNFT-VUL-018: Missing checks in Admin.deploy()

Severity	Warning	**Commit**	e329d79
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/admin.ts`		
Location(s)	Admin.deploy()		
Confirmed Fix At	`https://github.com/SilvanaOne/silvana-lib/pull/32`		

The `Admin.deploy()` function creates the `AccountUpdate` which admins should use to deploy the `Admin` contract. However, there are no internal consistency checks to ensure that if the contract is deployed with `canBePaused = false`, then `isPaused = false`.

If this does happen, then the contract cannot be resumed.

```
@method
async resume(): Promise<void> {
  await this.ensureOwnerSignature();
  this.canBePaused.getAndRequireEquals().assertTrue();
  this.isPaused.set(Bool(false));
```

Snippet 4.21: Snippet from `Admin.resume()`

Impact Admins may waste gas or accidentally deploy an unresumable contract.

Since `isPaused` is only checked when `Admin.admin` is changed (or when pausing/resuming), this may not be noticed immediately.

Recommendation Validate that `canBePaused` and `isPaused` are not both false before creating the deployment `AccountUpdate`.

Developer Response The developers now validate `isPaused` is `false` when `canBePaused` is `false` before creating the deployment `AccountUpdate`.

4.1.19 V-MNFT-VUL-019: Pausability of the collection and admin are connected

Severity	Warning	Commit	e329d79
Type	Usability Issue	Status	Fixed
File(s)	packages/nft/src/contracts/admin.ts		
Location(s)	NFTAdmin.pause()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/33		

The method `pause()` is used to pause certain administrative actions in the `NFTAdmin` contract. This method can only be called if the state field `canBePaused` is true. However, this same state field is also used to indicate whether the NFT collection can be paused. Therefore, the pausability of the collection and the admin are intertwined.

```
@method
async pause(): Promise<void> {
  await this.ensureOwnerSignature();
  this.canBePaused.getAndRequireEquals().assertTrue();
  this.isPaused.set(Bool(true));
  this.emitEvent("pause", new PauseEvent({ isPaused: Bool(true) }));
}
```

Snippet 4.22: Snippet from `pause()`

Having the same field denote the pausability of both these contracts may be surprising to NFT implementers.

Impact If the admin cannot be paused, then the collection cannot be paused as well and vice-versa. This can be problematic in cases where these two operations are performed independently of each other.

Recommendation Add a separate state field to the admin to track whether it can be paused.

If the implementation is intended, then add documentation to ensure that users are made aware of it.

Developer Response The developers added a separate boolean flag, `allowPauseCollection`, to control pausability of the `Collection`. They also added several clarifying comments to `Admin` fields.

4.1.20 V-MNFT-VUL-020: Admin may deploy unusable NFT vkey

Severity	Warning	**Commit**	e329d79
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	Collection._mint()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/34		

The `_mint()` function deploys an `NFT` contract at a new `Account` after `admin` or `creator` approval has been validated. The `verificationKey` is constrained using results from `Provable.witness()` (arbitrary instances of the provable types provided at runtime by whoever generates the proof) and two constants: the verification keys of the `NFT` contract compiled for the mainnet and devnet proving systems.

Since `isMainnet` is an arbitrary `Bool`, a prover may choose to set `verificationKey` to *either* the mainnet or devnet keys.

```
const verificationKey: VerificationKey = Provable.witness(
  VerificationKey,
  () => // [VERIDISE] arbitrary prover code
);

const mainnetVerificationKeyHash = Field(
  nftVerificationKeys.mainnet.vk.NFT.hash
);
const devnetVerificationKeyHash = Field(
  nftVerificationKeys.devnet.vk.NFT.hash
);
const isMainnet = Provable.witness(Bool, () => // [VERIDISE] arbitrary prover code
);
// We check that the verification key hash is the same as the one
// that was compiled at the time of the deployment
verificationKey.hash.assertEquals(
  Provable.if(
    isMainnet,
    mainnetVerificationKeyHash,
    devnetVerificationKeyHash
  )
);
```

Snippet 4.23: Snippet from `_mint()`

Impact If the verification key comes from the wrong proving system, none of the `NFT` functionality will work correctly. Consequently, a malicious admin or prover may mint a user an unusable `NFT`.

Recommendation Instead of using `Provable.if()`, use a regular JavaScript `if/else` block. Whichever path is taken at circuit-compilation time will be hard-coded into the circuit.

For example, in the below code snippet, `Contract0` and `Contract2` are identical when `config` is `false` at compilation time, while `Contract0` and `Contract1` are identical when `config` is `true` at compilation time.

```
let config: boolean = false;

class Contract0 extends SmartContract {
    @state(Bool) dummy = State<Bool>();
    @method async noop(): Promise<void> {
        if(config) {
            const dummy = this.dummy.getAndRequireEquals();
            dummy.assertTrue();
        }
    }
}

class Contract1 extends SmartContract {
    @state(Bool) dummy = State<Bool>();
    @method async noop(): Promise<void> {
        const dummy = this.dummy.getAndRequireEquals();
        dummy.assertTrue();
    }
}

class Contract2 extends SmartContract {
    @state(Bool) dummy = State<Bool>();
    @method async noop(): Promise<void> {
    }
}
```

Developer Response The developers now use a regular JavaScript `if`/ `else` block instead of `Provable.witness()` as per the recommendation.

4.1.21 V-MNFT-VUL-021: Unused Imports

Severity	Warning	**Commit**	06506ba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/SilvanaOne/silvana-lib/pull/35`		

The following are imported from `o1js` but never used within context of the files.

- `SmartContract` in `contracts/collection.ts`
- `Field` and `Bool` in `interfaces/ownable.ts`
- `Field` in `interfaces/pausable.ts`

Recommendation We recommend removing the unused imports.

Developer Response The developers removed the unused imports in `contracts/collection.ts` and `interfaces/pausable.ts`.

Updated Veridise Response An earlier typo in the issue wrote `interfaces/owner.ts` instead of `interfaces/ownable.ts`. This has been fixed in the issue text.

Updated Developer Response The additional import was removed.

4.1.22 V-MNFT-VUL-022: from param unused in transfer functions

Severity	Warning	**Commit**	e329d79
Type	Usability Issue	**Status**	Fixed
File(s)	packages/nft/src/contracts/collection.ts		
Location(s)	Collection.transferBySignature(), Collection.approvedTransferBySignature()		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/36		

The function transferBySignature() can be used to transfer an NFT without admin approval. Correspondingly, its counterpart approvedTransferBySignature() requires approval from the admin to transfer an NFT. If the NFT owner is a contract then transferByProof() and approvedTransferByProof() can be used respectively.

The signature-based transfers rely on creating an AccountUpdate from the (unconstrained) sender, and then checking that the sender is either the owner or the approved contract. Consequently, as shown in the below snippet, the "params.from" address is ignored.

```
    @method async approvedTransferBySignature(
     params: TransferParams
    ): Promise<void> {
     const { address, to, price, context } = params;
        //[Veridise]
        //...elided....
        //[Veridise]
     const transferEventDraft = new TransferExtendedParams({
      from: PublicKey.empty(), // will be added later
      //[Veridise]
         //...elided....
     });
     await this._transfer({
      transferEventDraft,
      //[Veridise]
         //...elided....
     });
    }
```

Snippet 4.24: Snippet from transferBySignaturet()

Impact Intentional misuse of this argument could affect audibility of traces, especially since the emitted event always sets from to owner.

Further, there is a chance for the functions approvedTransferByProof() and approvedTransferBySignature() to be associated with transferring an NFT using the approved address due to the ambiguous naming.

Recommendation Consider removing the from parameter from the signature-based transfer arguments.

Consider renaming the approved* transfer methods to adminApproved*.

Developer Response The developers removed the `from` parameter from signature-based transfer arguments.

4.1.23 V-MNFT-VUL-023: Typos and missing/incorrect comments

Severity	Info	**Commit**	06506ba
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/SilvanaOne/silvana-lib/pull/37`, `https://github.com/SilvanaOne/silvana-lib/pull/28`		

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

1. `packages/nft/src/`
   a) `contracts/collection.ts`:
      i) `Collection.approveAddressByProof()`: The nat-spec comment on this function is incorrect.
      ii) `Collection._transfer()`:
         1. The documentation for the function parameters in the nat-spec comment is out of date.
         2. The `// TODO` comment in `_transfer()` appears to be out of date.
         3. The fee structure is not documented. Consider documenting this to ensure admins set the `transferFee` is set correctly.
      iii) `Collection.transferOwnership()`: Consider changing the documentation (and possibly function name) to reference the `creator`, rather than referring to the `creator` as the "owner." This could reduce possible confusion between interpretations of both the `admin` and the `creator` as an "owner" of the `Collection` contract.
   b) `interfaces/`
      i) `events.ts`:
         1. `UpgradeVerificationKeyEvent.tokenId`: The documentation comment for this variable describes the version number instead of the `tokenId`.
      ii) `types.ts`:
         1. `NFTImmutableState.id`: This field is described as "The unique identifier of the NFT within the collection". However, it may be set arbitrarily by the collection administrator and defaults to always zero. The documentation should mention that non-admin users should not rely on this `id` for their operations, and instead use the NFT's public key and token ID to identify it.
         2. `NFTData`: Add documentation which carefully lists the intended behavior and uses for the various approval flags represented in `NFTData`.
            a) `.canChangeOwnerByProof`: The document should indicate that this flag is intended to be used only by the `update()` method, and that it overrides both the `canApprove` and `canTransfer` flags.
            b) `.canApprove`: The current documentation indicates that `approved` cannot be changed when `canApprove` is `false`. However, `approved` is reset to empty upon the first transfer. This edge case should be noted in the

documentation of the configuration flag. Additionally, the documentation does not note that `canApprove` may be bypassed when `canChangeOwnerByProof` is `true`.

   c) `.canTransfer`: The documentation does not note that `canTransfer` may be bypassed when `canChangeOwnerByProof` is `true`.

3. `CollectionData.mintingIsLimited()`: Add documentation to mention that this particular method is not a getter for `mintingIsLimited`, or rename to `mintingIsLimitedOrPaused()`, to avoid misusing it in the future.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers implemented the recommendation.

4.1.24 V-MNFT-VUL-024: Recommended contract factory validations

Severity	Info	**Commit**	e329d79
Type	Data Validation	**Status**	Fixed
File(s)	`packages/nft/src/contracts/collection.ts`		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/SilvanaOne/silvana-lib/pull/41		

The Mina NFT standard uses a new contract factory pattern for development. For example, suppose a contract `Foo` is intended to call a contract `Bar`. Using the contract factory pattern, `Foo` would access `Bar` by calling a function which returns a constructor for `Bar`, instead of just calling `Bar` directly. An example can be seen in the below code snippet.

```
function FooFactory(barFactory: () => BarConstructor) {
    class Foo extends SmartContract {
      @method async foo(address: PublicKey) {
        const barInstance = new BarConstructor()(address);
        barInstance.bar();
      }
    }
    return Foo;
}
```

Since the logic of `Foo` and `Bar` are compiled separately, taking this approach (instead of just calling `new Bar()` directly) should not change the verification key of `Foo`.

This pattern allows users to more easily swap out different implementations of `Bar`, so long as each implementation has a `@method` with the same signature as `Bar.bar()`. This is especially helpful for the NFT standard, which expects users to have custom admin, owner, update, and approver contracts.

Impact When compiling a class created with the factory pattern, users must call the factory to get a concrete instance of the class, then compile that instance. To ensure that all the usual checks performed when calling another smart contract are in place, this instance must be instantiated with constructors of actual o1js smart contracts.

For example, a malicious compiler could use an overriden o1js smart contract whose constructor sets its `tokenId` to an unconstrained variable, instead of a constant 1. This would create an attack vector which may allow an attacker to maliciously deploy contracts with the `Collection`'s `tokenId`.

Recommendation When using the factory pattern,

1. Compile the factory-created contract with concrete instantiations of the contracts it may call.
2. Compile the factory-created contract with multiple different concrete instantiations of the contracts it may call, and validate the `vkey` is unchanged.
3. Consider using `Provable.isConstant()` to check that the `AccountUpdate` produced by method calls has a constant token ID of 1.

```
const OwnerContract = ownerContract();
const owner = new OwnerContract(address);
assert(Provable.isConstant(Field, owner.self.tokenId))
Provable.assertEqual(Field, owner.self.tokenId, TokenId.default);
return owner;
```

Developer Response The developers included a best practices section in the readme, along with `BEST_PRACTICES.md` . This contains sections **Recommended contract factory validations for developers** and **Best practices of contract factories** which outline what was mentioned in the issue writeup.

Glossary

Mina Mina Protocol is a succinct 22KB blockchain utilizing zero-knowledge proofs. See `https://minaprotocol.com` for more details. 1, 47

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, 47

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. 47

zkApp A smart contract written for the Mina blockchain. See `https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks` for more. 47