Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

MINA

Mina Fungible Token Standard

Veridise Inc.
August 05, 2024

► **Prepared For:**

Mina Foundation
https://minaprotocol.com

► **Prepared By:**

Benjamin Sepanski
Sorawee Porncharoenwase

► **Contact Us:** contact@veridise.com

► **Version History:**

Aug. 05, 2024	V3 - Updated with fixes from o1js
Jul. 23, 2024	V2 - Updated with fixes from Mina Foundation
Jul. 07, 2024	V1
Jul. 05, 2024	Initial Draft

Contents

1 Executive Summary

From Jun. 24, 2024 to Jun. 28, 2024, the Mina Foundation engaged Veridise to review the security of their Mina Fungible Token Standard. The review covered the implementation of a standard fungible token contract for the Mina blockchain, as well as the components in the o1js library necessary to implement the contract. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week on commits `ebbc088a` (mina-fungible-token) and `1588ee56` (o1js). The auditing strategy consisted of extensive manual code review performed by Veridise engineers.

Project summary. The security assessment covered two primary components: the smart contracts implementing the token standard and the zero-knowledge circuit (ZK Circuit) implementing the constraints using the o1js library. The token standard itself implements a fungible token similar to popular standards such as the popular Ethereum ERC 20 standard.

To ensure compatibility across wallets and provers, the token contract itself is intended to be used without alteration. Certain functions call to an administrator contract which may provide custom logic for individual tokens. The token balances are maintained using Mina's native custom token feature, allowing for concurrent transfers. Other operations support concurrency via the actions and reducer model.

To ensure composability with other contracts, entire trees of `AccountUpdates` may be processed using utility functions implemented in the o1js library. These are implemented using a Merkle List, which was also in scope of the review.

Code assessment. The Mina Foundation developers provided the source code of the Mina Fungible Token Standard contracts and circuits for review. The source code appears to be original code written by the Mina Fungible Token Standard and o1js developers. It contains extensive documentation in the form of READMEs and other Markdown files*. Documentation comments on functions and storage variables explain the purposes of individual program constructs. To facilitate the Veridise auditors' understanding of the code, the Mina Fungible Token Standard developers also included several examples to illustrate the project's intended use.

The source code contains a test suite, which the Veridise auditors noted thoroughly tested each operation. The test suite also includes checks to ensure that operations fail in expected cases including unauthorized actions, insufficient balances, and incorrect transfer amounts.

Summary of issues detected. The audit uncovered 13 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, use of features still in beta leads to a denial-of-service attack (V-MIN-VUL-001), and attackers may flash-mint tokens without affecting the circulating supply (V-MIN-VUL-002). The Veridise auditors also identified 1 medium-severity

* `https://github.com/MinaFoundation/mina-fungible-token/tree/eaf5a6b322/documentation`

issue (V-MIN-VUL-003) which identifies a potential denial-of-service attack due to overflow in a lazily-accumulated sum. The Veridise auditors additionally identified 3 low-severity issues, 3 warnings, and 4 informational findings. These include centralization risks (V-MIN-VUL-004, V-MIN-VUL-006), improper handling of token ID permissions (V-MIN-VUL-005), and several minor concerns.

Of the 13 issues, the Mina Foundation has fixed 9 issues and provided partial fixes to 3 more. This includes fixes to all issues of low severity or higher, with the centralization issue (V-MIN-VUL-004) marked as partially fixed. Note that the centralization issue is marked as partially fixed since, by default, the token minting/burning/pausing is controlled by a single entity. Mina Foundation has mitigated the possibility of malicious upgrades and recommended secure practices for centralized parties in their documentation.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Mina Fungible Token Standard.

Improved Documentation. The o1js access-control system is highly expressive, which carries with it some complexity. In particular, understanding permissions surrounding the parent-child relationship can be difficult and lead to issues such as V-MIN-VUL-005 and V-MIN-VUL-006. The Veridise auditors recommend producing a detailed description of the following topics:

- The "access" permission: this permission field appears innocuous, but can have important impacts on which `AccountUpdates` may be parents of others (see V-MIN-VUL-006). This is mentioned in the documentation. However, a full reference document on the allowed actions associated to each permission would be highly beneficial to auditors and developers.
- Token IDs: each `AccountUpdate` is associated with a `PublicKey` and a `tokenId`. The allowed values for a `tokenId` depend on the `AccountUpdate`'s parent and the two boolean flags which make up its `mayUseToken` field. Any developer implementing a contract which produces `AccountUpdates` with children must understand the full permission structure implied by the restrictions on an `AccountUpdate`'s `tokenId`. Adding technical details to the existing documentation on terminology describing the precise semantics of `tokenIds` will help ensure they are used properly going forward.

Actions & Reducers. Actions and reducers, while powerful, come with heavy risks. Users may provably submit actions, which are later processed by the reducer in sequence. This opens up a generically applicable denial-of-service strategy in which an attacker submits an action which is designed to trigger an assertion failure during reduction. Since reductions process actions in the order which they were submitted, this may permanently shut down the contract. See V-MIN-VUL-003 for an example.

The Veridise auditors have a few recommendations regarding both the Mina Foundation team's Mina Fungible Token Standard and to the o1js team's design:

- Consider avoiding the actions and reducer model when possible. For example, leveraging account token balances for purposes other than tracking token balances can allow users to provide concurrent updates in some cases. See also the recommendation in V-MIN-VUL-003.

- Add additional documentation (and corresponding examples to the security best practices) indicating that an validating an action submission must *guarantee* a successful reduction computation.
- Consider whitelisting types allowed as action types. Attackers may leverage over-constrained operations in the reducer to trigger a revert. For example, non-native elliptic curve addition reverts when two points with the same x-coordinate are added together.
- One final option may be switching to a model in which actions may be processed in any order. This could be done by changing the Merkle List to a Merkle Tree, or by allowing callers to move actions into a separate Merkle Tree for out-of-order processing. The trade-offs here must be considered carefully, as a user-controlled processing order may create opportunities for front-running-style attacks such as sandwich attacks.

Redesigns. The Veridise team recommends reconsidering some of the design points regarding proper handling of account update forest structures (V-MIN-VUL-002, V-MIN-VUL-004, V-MIN-VUL-005, V-MIN-VUL-006, V-MIN-VUL-008) and handling actions and reducers (V-MIN-VUL-001, V-MIN-VUL-003). Validating entire forest structures is a complex task, and may be error-prone in general. Targeting a simpler input structure, such as limiting the input forest to "send-receive pairs", may make reasoning about the protocol simpler. As described above, using actions and reducers come with their own risks and should be treated with great care.

Centralization. As described in V-MIN-VUL-004 and V-MIN-VUL-006, the default token implementation is centralized and puts a large amount of trust in the deployer. The Veridise auditors recommend that the Mina Foundation developers include instructions for users on how to assess the trust assumptions of a token based on its permissions and administrator.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
mina-fungible-token[a]	`ebbc088a`	TypeScript	Mina
o1js[b]	`1588ee56`	TypeScript	Mina

[a] `https://github.com/MinaFoundation/mina-fungible-token/`
[b] `https://github.com/o1-labs/o1js/`

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jun. 24–Jun. 28, 2024	Manual	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	2	2	2
Medium-Severity Issues	1	1	1
Low-Severity Issues	3	3	2
Warning-Severity Issues	3	2	2
Informational-Severity Issues	4	4	2
TOTAL	13	12	9

Table 2.4: Category Breakdown.

Name	Number
Maintainability	3
Denial of Service	2
Transaction Ordering	2
Data Validation	2
Access Control	1
Authorization	1
Missing/Incorrect Events	1
Usability Issue	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Mina Fungible Token Standard's smart contracts. In our audit, we sought to answer questions such as:

- Does the token contract properly track balances and token supply?
- Is authorization tracked during token transfers?
- Can tokens be minted or burned without proper authorization?
- Can non-privileged entities perform administrative actions like pausing or upgrading the contract?
- Are there any common programming/smart contract risks such as front-running, unused variables, or unchecked returns?
- Can funds become locked?
- Are circuits properly constrained?
- Does the Merkle List implementation properly compute hashes? Is the encoding of its inputs injective?
- Does iteration over the `AccountUpdate` forest visit all relevant `AccountUpdates`?
- Can actions trigger reversion during reduction?
- Are any known issues from other token standards present in the Mina Fungible Token Standard?

3.2 Audit Methodology & Scope

Scope. The scope of this audit includes two repositories: the Mina Fungible Token Standard[*] and o1js[†]. The Mina Fungible Token Standard scope is limited to the root folder of the source code provided by the Mina Fungible Token Standard developers, which contains the smart contract implementation of the Mina Fungible Token Standard. In the o1js repository, the following files were in scope:

- `src/lib/mina/token/*.ts`
- `src/lib/provable/merkle-list.ts`

Methodology. Veridise auditors reviewed the reports of previous audits for Mina Fungible Token Standard, inspected the provided tests, and read the Mina Fungible Token Standard documentation. They then began a manual review of the code. During the audit, the Veridise auditors regularly communicated with the Mina Foundation developers to ask questions about the code.

[*] `https://github.com/MinaFoundation/mina-fungible-token/`
[†] `https://github.com/o1-labs/o1js/`

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

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

In this case, we judge the likelihood of a vulnerability as follows in 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

In addition, we judge the impact of a vulnerability as follows in 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

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-MIN-VUL-001	More than 500 pending mint/burns leads to DoS	High	Fixed
V-MIN-VUL-002	Tokens may be flash-minted without changing . . .	High	Fixed
V-MIN-VUL-003	Lazy sum accumulation can enable overflow DoS	Medium	Fixed
V-MIN-VUL-004	Centralization Risk	Low	Partially Fixed
V-MIN-VUL-005	ForestIterator checks for exact match in . . .	Low	Fixed
V-MIN-VUL-006	access permissions allows token deployer to . . .	Low	Fixed
V-MIN-VUL-007	Tokens default to unpaused	Warning	Fixed
V-MIN-VUL-008	Missing event on state update	Warning	Fixed
V-MIN-VUL-009	transfer() overwrites balanceChange	Warning	Open
V-MIN-VUL-010	Unused and duplicate program constructs	Info	Partially Fixed
V-MIN-VUL-011	Best practices and recommendations	Info	Partially Fixed
V-MIN-VUL-012	Use of deprecated o1js functions	Info	Fixed
V-MIN-VUL-013	Comments on token design	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MIN-VUL-001: More than 500 pending mint/burns leads to DoS

Severity	High	**Commit**	ebbc088
Type	Denial of Service	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	mint(), burn(), calculateCirculating()		
Confirmed Fix At	https://github.com/MinaFoundation/mina-fungible-token/pull/91,https://github.com/MinaFoundation/mina-fungible-token/pull/96		

In order for a reduction to succeed, it must process *all* actions which have been submitted but have not been processed. Otherwise, it will revert, with no possibility for recovery other than upgrading the contract.

`FungibleToken` dispatches actions during mints or burns to allow for concurrent supply changes. The `getCirculating()` and `updateCirculating()` methods compute the net supply change by `reduce()`-ing over the accrued actions. Both methods use the utility function, which configures the maximum number of actions per-reduction to be 500.

```
@method.returns(AccountUpdate)
async burn(from: PublicKey, amount: UInt64): Promise<AccountUpdate> {
  this.paused.getAndRequireEquals().assertFalse()
  const accountUpdate = this.internal.burn({ address: from, amount })
  this.emitEvent("Burn", new BurnEvent({ from, amount }))
  this.reducer.dispatch(Int64.fromUnsigned(amount).neg())
  return accountUpdate
}
```

Snippet 4.1: Definition of `burn()`

As can be seen in the above snippet, anyone may burn tokens (if they have them to burn). This allows any user to dispatch actions to the reducer.

Impact This attack is readily available to any user with an incentive to shut down the protocol. For example, a user with no balance could submit 501 `burn()`s of a 0-amount.

Once over 500 actions have been dispatched, the reducer will no longer function, and no further actions may be processed.

Recommendation Consider using a designated account's balance in the token ID to track the circulating balance. This has several benefits:

- Unlimited numbers of concurrent balance changes are supported.
- The circulating value will not need to be lazily computed.

Alternatively, provide functionality for a user to submit a recursive proof that they performed the reduction on all of the outstanding actions as a recovery mechanism.

Developer Response We no longer use the action/reducer model and instead rely on a designed account balance.

4.1.2 V-MIN-VUL-002: Tokens may be flash-minted without changing circulating

Severity	High	**Commit**	1588ee5, ebbc088
Type	Transaction Ordering	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	approveBase()		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/86`		

The `approveBase()` function allows users to process a large number of token transactions without needing to generate each one with a call to `transfer()`. As shown in the below snippet, `approveBase()` ensures the net balance change is 0. Additionally, Mina nodes ensure each account's token balance remains non-negative (not shown). These together imply conservation of the currency after execution of the `updates` approved by `approveBase` (ignoring the possibility of any `mint()`s or `burn()`s within the `updates`).

```
@method
async approveBase(updates: AccountUpdateForest): Promise<void> {
  this.paused.getAndRequireEquals().assertFalse()
  let totalBalance = Int64.from(0)
  this.forEachUpdate(updates, (update, usesToken) => {
    this.checkPermissionsUpdate(update)
    totalBalance = Provable.if(usesToken, totalBalance.add(update.balanceChange),
    totalBalance)
  })
  totalBalance.assertEquals(Int64.zero)
}
```

Snippet 4.2: Definition of `approveBase()`

Note that conservation of the currency is *not* guaranteed between the updates executed inside `approveBase()`. This allows users to effectively "flash-mint" tokens by

1. Including an `AccountUpdate` with a large positive balance change.
2. Using the funds to profit on some 3rd-party application.
3. Including an `AccountUpdate` with a large negative balance change to ensure currency conservation.

On initial review, some may considered this a feature rather than a bug. However, this flash-minting breaks one important invariant: no action is dispatched to update the `circulating` supply. In particular, during any transaction execution, `getCirculating()` may return a value entirely different from the total circulating supply.

Impact Misrepresentation of a token's total supply can have severe consequences. For instance, the below examples illustrate how this could be devastating for a voting protocol.

Note that, unlike with regular flash-minting which would have to go through a `mint()`-type function, this process occurs through a method without any calls to the `admin` contract. As a consequence, the `admin` contract *cannot turn off this behavior* with the current token implementation.

Example 1. Consider a voting protocol which is the `admin` of a token contract. Users may purchase `1 VoteTkn` for `1 Mina` by calling a deposit method in the `admin` contract, which authorizes the minting of `1 VoteTkn` to the depositor after some waiting period has passed.

Users may submit proposals to the `admin` contract on how to spend its accrued `Mina`. Token holders may send their tokens to a proposal to vote for it. Each proposal is executed once it reaches at least 2/3 of the circulating supply. After the proposal is executed, or at any time before the vote ends, voters may reclaim their vote tokens.

An attacker holding `1 VoteTkn` may break this system by the following sequence of actions:

1. Create a proposal to send them all of the `Mina`.
2. In a single transaction:
    a) Flash-mint themselves enough votes to pass the proposal.
    b) Vote on and execute the proposal.
    c) Reclaim the vote tokens.
    d) Pay back the flash-mint.

While this could be prevented by requiring a transaction to pass between a successful vote and proposal execution, developers may believe the time delay on purchasing `VoteTkns` is sufficient to protect against attacks and leave out the extra protection.

Example 2. The above example illustrates that flash-minting without `admin` authorization can lead to unforeseen consequences. In this example, we demonstrate how the mismatch between circulating tokens and `getCirculating()` can make this problem worse.

Note that one intended property of the above protocol is the following: No two proposals may be executed simultaneously. For a proposal to execute, it must have at least 2/3 of the supply. Once it starts executing, those funds are locked until it finishes, so no other proposal should also have 2/3 of the supply.

Flash-minting without changing the result of `getCirculating()` allows someone to avoid this problem. An attacker may flash-mint themselves enough votes to execute *every* proposal, since each flash-mint does not change the "denominator" of the computation.

This can be exploited if, for example, two proposals are designed explicitly to be mutually exclusive. Suppose multiple teams are bidding to purchase a set amount of protocol funds, and each proposal's `execute()` method checks:

1. The proposal contract has sufficient votes.
2. The current round is the expected proposal round.

An attacker may then create their own proposal which executes *all* of the bidder's proposals, despite each bidder requiring they have 2/3 of the vote in that round.

Recommendation Require that `totalBalance <= 0` after each update to `totalBalance` inside `forEachUpdate()`.

Developer Response `approveBase()` now checks that `totalBalance` is non-positive after each balance update.

4.1.3 V-MIN-VUL-003: Lazy sum accumulation can enable overflow DoS

Severity	Medium	**Commit**	ebbc088
Type	Denial of Service	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	mint(), burn(), calculateCirculating()		
Confirmed Fix At	https://github.com/MinaFoundation/mina-fungible-token/pull/91,https://github.com/MinaFoundation/mina-fungible-token/pull/96,https://github.com/MinaFoundation/mina-fungible-token/pull/101		

As described in V-MIN-VUL-001, `mint()`-ing or `burn()`-ing dispatches an action to the reducer describing the change in circulating supply. When the circulating supply is computed, these changes are aggregated into a single `UInt64`.

```
private calculateCirculating(
  oldCirculating: UInt64,
  pendingActions: MerkleList<MerkleList<Int64>>,
): UInt64 {
  let newCirculating: Int64 = this.reducer.reduce(
    pendingActions,
    Int64,
    (circulating: Int64, action: Int64) => {
      return circulating.add(action)
    },
    Int64.from(oldCirculating),
    { maxUpdatesWithActions: 500 },
  )
  newCirculating.isPositive().assertTrue()
  return newCirculating.magnitude
}
```

Snippet 4.3: Definition of `calculateCirculating()`

Importantly, the final sum of the unprocessed actions must:

1. Fit within an `Int64`, i.e. the (non-standard) range $[-2^{64}+1, 2^{64}-1]$, in *every intermediate computation*.
2. Leave the circulating supply non-negative (see V-MIN-VUL-012 for a description of `isPositive()`).

If either of these does not hold, then the circulating supply cannot be computed. While property 2 may be fixed by issuing mints, property 1 cannot be fixed by adding more actions.

Impact If property 1 is ever violated, there is no mechanism to recover from this state. The `calculateCirculating()` function (used by both `getCirculating()` and `updateCirculating()`) would be permanently DoS'ed.

Fortunately, `mint()`s are controlled by the admin. However, as described in V-MIN-VUL-002, users may currently "flash-mint" themselves large amounts of tokens. For tokens which represent wrapped underlying tokens, this may allow any user to DoS the protocol via a flash-mint, burn, deposit, withdraw sequence.

If the flash-minting issue is fixed, this may still become a problem, though it is less likely. 64 bits is just over 19 decimals. Assuming at most 500 pending actions at any given time (see V-MIN-VUL-001) and a token with 9 decimals, this leaves $3.6 \cdot 10^7$ tokens-per action which must be submitted. For many common applications, this is likely to be difficult. However, for tokens with the following properties, this will still be exploitable:

- Tokens with large numbers of decimals (e.g. 12).
- Tokens with extremely low economic value.
- Tokens which may be flash-minted.
- Tokens which wrap other vulnerable tokens as underlying assets.

Recommendation As recommended in V-MIN-VUL-001, consider aggregating the circulating supply into the balance of a designed account, rather than tracking in state.

Developer Response Instead of using actions and reducers to update the current circulation in the contract state, we now use an account where the balance corresponds to the current circulation.

- We use an account with the key of the token contract.
- The account balance is updated at every call of `mint()` and `burn()`.
- The `approveBase()` method checks that none of the involved `AccountUpdates` involve this special account.

Updated Veridise Response

- `mint()` and `burn()` should check that the `recipient` is not `this.address`. If not, the minting/burning will be double-count in the total supply.
- `transfer()` should check that `from` and `to` are not `this.address`. This is particularly important, since the account at `this.address` has sending permissions set to `none()`. This will also prevent the contract deployer from transferring funds.

Out of an abundant sense of caution, it may be worth making the circulating supply account's permissions more restrictive. Of special note is the `setPermissions` permission.

Updated Developer Response We added checks in `transfer()`, `mint()`, and `burn()` to prevent unexpected changes in the `circulatingSupply` address.

Updated Veridise Response Upon further review, we have some additional concerns regarding the circulating-supply account.

1. The permissions for the circulating-supply account are shown below

```
{
  editState: Permission.proof(),
  send: Permission.none(),         // Only difference from Permissions.default()
  receive: Permission.none(),
  setDelegate: Permission.signature(),
  setPermissions: Permission.signature(),
  setVerificationKey: Permission.VerificationKey.signature(),
  setZkappUri: Permission.signature(),
  editActionState: Permission.proof(),
  setTokenSymbol: Permission.signature(),
  incrementNonce: Permission.signature(),
  setVotingFor: Permission.signature(),
  setTiming: Permission.signature(),
  access: Permission.none(),
}
```

While the Veridise auditors cannot identify a means by which the owner may create a child `AccountUpdate` at the `circulatingSupply` address which changes the permissions, further restricting `setPermissions` to `Permission.impossible()` may provide additional insurance that a compromised or malicious token deployer cannot change the `circulatingSupply` permissions to DoS the protocol or steal tokens.

2. As described in https://github.com/o1-labs/o1js/issues/1439, `AccountUpdates` changing token balances of `Accounts` (including but not limited to the `circulatingSupply` account) may be performed *before* the smart contract is deployed. While the `isNew` check in the `deploy()` transaction protects the *deployer* from being front-run, a malicious or compromised deployer may intentionally change the deployment transaction to not assert `isNew`. This allows a malicious deployer to pre-initialize accounts with large amounts of funds. Consider documenting this possibility and recommending any wallet checks that `isNew` was set in a token's deployment before using that token.

Updated Developer Response We now initialize the `circulatingSupply` account's `setPermissions` permission to `Permission.impossible()`.

4.1.4 V-MIN-VUL-004: Centralization Risk

Severity	Low	**Commit**	ebbc088
Type	Access Control	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/MinaFoundation/mina-fungible-token/pull/99		

Similar to many projects, Mina's fungible token standard declares an administrator role that is given special permissions. In particular, these administrators are given the following abilities:

- Minting tokens.
- Pausing/unpausing the contract.

This administrator role is intended to be a smart contract. The default implementation is managed by a privileged entity who knows the secrete key associated to the admin contract's `adminPrivateKey` field.

Additionally, token contracts (by default) use o1js' default smart contract permissions, with `access` overridden to `Permission.proofOrSignature()`. As shown in the below snippet, this means that both contract upgradeability and contract permissions may be changed by the contract deployer, i.e. whoever knows the private key associated to the smart contract's address.

If that user is compromised, then the token balances will also become compromised. By upgrading the contract, the attacker may mint themselves an arbitrary number of tokens, or otherwise change the behavior of the token contract as they desire.

```
default: (): Permissions => ({
  editState: Permission.proof(),
  send: Permission.proof(),
  receive: Permission.none(),
  setDelegate: Permission.signature(),
  setPermissions: Permission.signature(),
  setVerificationKey: Permission.VerificationKey.signature(),
  setZkappUri: Permission.signature(),
  editActionState: Permission.proof(),
  setTokenSymbol: Permission.signature(),
  incrementNonce: Permission.signature(),
  setVotingFor: Permission.signature(),
  setTiming: Permission.signature(),
  access: Permission.none(),
}),
```

Snippet 4.4: Default permissions for o1js smart contracts.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the integrity of the token.

Recommendation As these are all particularly sensitive operations, anyone deploying a token should utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Additionally, any user purchasing a token should investigate the key management practices of the token deployer and validate the token contract permissions as one should with any o1js application.

Developer Response Documentation has been added describing the admin role's privileges and recommending strong security practices for centralized admin contracts.

4.1.5 V-MIN-VUL-005: ForestIterator checks for exact match in MayUseToken

Severity	Low	**Commit**	1588ee5
Type	Data Validation	**Status**	Fixed
File(s)	`src/lib/mina/token/forest-iterator.ts`		
Location(s)	next()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1741`		

The `TokenAccountUpdateIterator` iterates over an `AccountUpdate` forest using a pre-order depth-first traversal. It does this by maintaining a stack of Merkle lists, each representing the siblings of the current node's ancestors which have yet to be processed. Since ZK Circuits are limited in their size at compile time, callers can only support a fixed number of iterations.

To help mitigate this problem, the `TokenAccountUpdateIterator` attempts to skip sub-trees which may not update the derived token id of the token contract. To do so, the iterator skips the children of the current `accountUpdate` if the `accountUpdate` is either

1. An `AccountUpdate` for the token contract itself. This tree will be verified independently during transaction validation.
2. An `AccountUpdate` whose `tokenId` cannot be the token's derived token-id, nor can any of its children's.

```
// if we don't have to check the children, ignore the forest by jumping to its end
let skipSubtree = canAccessThisToken.not().or(isSelf);
childForest.jumpToStartIf(skipSubtree);
```

Snippet 4.5: Snippet from `next()` which skips the child subtree if either case (1.) (`isSelf`) or case (2.) (`canAccessThisToken.not()`) is true.

Identifying case (1.) is easy. Identifying case (2.) has two sub-cases:

1. Top-level: if `accountUpdate.mayUseToken.parentsOwnToken` is true, the account update's token id may be set to the token contract's derived token ID
2. Not top-level: Since (by (1.)) all `AccountUpdates` which could derive the token contract's derived token ID are skipped, we know that the derived ID of the *parent* of this inner node is not equal to the token contract's derived token ID. Therefore, the only way we can use the token contract's derived token ID is if we *inherit it* from our parent. In particular, `accountUpdate.mayUseToken.parentsOwnToken` must be true.

To implement the check for case (2.), an invariant is maintained on the iteration "`layers`" stored in the stack. `layer.mayUseToken` is `MayUseToken.ParentsOwnToken` iff its `layer` is the top-level. Otherwise, `layer.mayUseToken` is `MayUseToken.InheritFromParent`. So, the developers can determine whether this account update's children can access this token by comparing `accountUpdate.mayUseToken.parentsOwnToken` to the current layer's `mayUseToken` field.

```
// check if this account update / it's children can use the token
let canAccessThisToken = Provable.equal(
  MayUseToken.type,
  update.body.mayUseToken,
  this.currentLayer.mayUseToken
);
```

Snippet 4.6: Computation of `canAccessThisToken`.

Unfortunately, as described in the o1js audit report, the `mayUseToken` has four representations:

1. `{parentsOwnToken: false, inheritFromParent: false}`
2. `MayUseToken.InheritFromParent := {parentsOwnToken: false, inheritFromParent: true}`
3. `MayUseToken.ParentsOwnToken := {parentsOwnToken: true, inheritFromParent: false}`
4. `{parentsOwnToken: true, inheritFromParent: true}`

This fourth case is accepted by the VM, and treated as equivalent to case 2.

In particular, this causes the iteration to skip any account update with both flags set to `true`. This can be seen in the below code snippet, in which iteration skips the entire bottom level of a tree of account updates.

```
et updates: AccountUpdate[] = []
const length = 7;
for(let i = 0; i < length; ++i) {
  let update = AccountUpdate.defaultAccountUpdate(alexa.publicKey, token.
    deriveTokenId());
  update.label = ‘Update ${i}‘;
  updates.push(update);
}

// forest[0]
// Build a complete, 3-level tree
// t0
// t1    t2
// t3 t4 t5 t6
updates[3]!.body.callDepth = 2;
updates[4]!.body.callDepth = 2;
updates[5]!.body.callDepth = 2;
updates[6]!.body.callDepth = 2;
updates[1]!.body.callDepth = 1;
updates[2]!.body.callDepth = 1;
updates[0]!.body.callDepth = 0;

let trees: AccountUpdateTree[] = []
for(let i = 0; i < length; ++i) {
  let update = updates[i]!;
  update.body.mayUseToken = update.body.callDepth === 0
  ? { parentsOwnToken: new Bool(true), inheritFromParent: new Bool(false) }
  : { parentsOwnToken: new Bool(true), inheritFromParent: new Bool(true) };
  update.body.balanceChange = update.body.callDepth === 2 ? new Int64(new UInt64(100n
    )) : Int64.zero;
  trees.push(AccountUpdateTree.from(update))
}

trees[1]!.children.push(trees[3]!)
trees[1]!.children.push(trees[4]!)
trees[2]!.children.push(trees[5]!)
trees[2]!.children.push(trees[6]!)
trees[0]!.children.push(trees[1]!)
trees[0]!.children.push(trees[2]!)

```

```
// top-level tree to forest
let forest = AccountUpdateForest.empty();
forest.push(trees[0]!);

// Iterate over the forest so we can see what will be accessed!
let iterator = TokenAccountUpdateIterator.create(forest, token.deriveTokenId());
for(let i = 0; i < 7; ++i) {
  let {accountUpdate, usesThisToken} = iterator.next();
  Provable.log(i, usesThisToken, accountUpdate.toPretty())
}

console.log("Attack transaction.")
const attackTx = await Mina.transaction({
  sender: feepayer.publicKey,
  fee,
}, async () => {
  AccountUpdate.fundNewAccount(feepayer.publicKey, 1);
  await token.approveBase(forest);
})
console.log(attackTx.toPretty());
await attackTx.prove()
attackTx.sign([feepayer.privateKey, alexa.privateKey])
const attackTxResult = await attackTx.send()
console.log("Attack tx:", attackTxResult.hash)
await attackTxResult.wait()

await printTxn(attackTx, "Attack tx", legend)
await showTxn(attackTx, "Attack tx", legend)
```

Snippet 4.7: Malicious `AccountUpdate` tree construction. This PoC can be used in the setup provided by `examples/concurrent-transfer.eg.ts`.

This outputs

```
0 true {
  label: 'Update 0',
  ...
}
1 true {
  label: 'Update 2',
    ...
}
2 true {
  label: 'Update 1',
   ...
}
3 false {
  label: 'Dummy',
   ...
}
4 false {
  label: 'Dummy',
   ...
}
5 false {
  label: 'Dummy',
   ...
}
6 false {
  label: 'Dummy',
   ...
}
```

As a consequence, `AccountUpdates` which may not provably ignore token balances will be ignored by the iterator.

However, when a transaction is hashed by the network, it serialized and then de-serialized. During serialization, the `mayUseToken` field is validated to have at most one `true` flag, preventing this attack from successfully executing on the network.

Impact Currently, it seems this can attack cannot be executed due to the serialization and de-serialization routines. However, changes to the codebase could enable this attack in the future if it is not carefully documented for developers.

Recommendation Since the checks preventing this attack are far from the in-scope code and not familiar to the auditors, we recommend the Mina team take extra care to validate that there is no path by which a transaction can be submitted without being de-serialized.

At the very least, add documentation to the `AccountUpdate` struct and the relevant portion of the `mina` repository indicating the importance of checking that `mayUseToken` has at most one `true` boolean flag.

A more robust fix would ensure this situation cannot occur within any valid proof. To do this, consider updating the `check()` function of `AccountUpdate` to rule out the `{true, true}` case.

Developer Response We rate this as more severe than a warning. We want correct code execution to be proven and be universally verifiable, and not rely on each node operator to check wellformedness on deserialization.

Updated Veridise Response We have updated the severity and recommendation.

Updated Developer Response We updated `check()` inside of o1js-bindings to validate that not both `inheritFromParent` and `parentsOwnToken` are set. Additionally, the Mina `account_update` constructor now parses the `MayUseToken` into one of the three allowed variants, erroring on the `{true, true}` case.

4.1.6 V-MIN-VUL-006: access permissions allows token deployer to mint tokens

Severity	Low	**Commit**	ebbc088
Type	Authorization	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	deploy()		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/99/files`		

Since `access` permissions are (by default) set to `proofOrSignature`, an owner may attach arbitrary account balance updates beneath an `AccountUpdate` to the deployed smart contract which does not require proof authorization, e.g. a nop `AccountUpdate` or a receive `AcccountUpdate`.

```
console.log("Attack transaction.")
const attackTx = await Mina.transaction({
  sender: feepayer.publicKey,
  fee,
}, async () => {
  AccountUpdate.fundNewAccount(feepayer.publicKey, 1)
  let nopUpdate = AccountUpdate.defaultAccountUpdate(token.address, token.tokenId);

  let maliciousUpdate = AccountUpdate.defaultAccountUpdate(alexa.publicKey, token.
    deriveTokenId());
  maliciousUpdate.balanceChange = new Int64(new UInt64(100n));
  maliciousUpdate.body.mayUseToken = {parentsOwnToken: new Bool(true),
    inheritFromParent: new Bool(false)}
  AccountUpdate.attachToTransaction(nopUpdate);

  nopUpdate.approve(maliciousUpdate);

  nopUpdate.requireSignature();
  maliciousUpdate.requireSignature();
})

await attackTx.prove()
attackTx.sign([feepayer.privateKey, contract.privateKey, alexa.privateKey])
const attackTxResult = await attackTx.send()
console.log("Attack tx:", attackTxResult.hash)
await attackTxResult.wait()

console.log(attackTx.toPretty())
```

Snippet 4.8: Snippet showing how a token deployer can generate an unlicensed mint by attaching it to a nop `AccountUpdate`.

Impact

1. The amount of trust placed in the token deployer is (by default) total. Signature `access` permissions are required for other signature-authorized permissions like upgrading, setting permissions, or setting the URI in the default smart contract permissions. While

the owner is already a privileged actor by default, this is an important action they can perform which may be unexpected to users of the token.
2. Whenever someone upgrades a token contract to be "non-upgradeable," users **must** check that the owner set `access` permissions to proof only.

Recommendation Document this possibility clearly in the token contract, on the permissions documentation, and on the documentation of the "approve" function.

Consider changing the default behavior to upgrading/changing permissions via a check with the admin contract, and defaulting smart contract `access` permissions to proof-only.

Developer Response Permissions for `access` are now set to `proof`-only. We've also added a `@method` to upgrade the contract when the o1js version changes.

4.1.7 V-MIN-VUL-007: Tokens default to unpaused

Severity	Warning	**Commit**	ebbc088
Type	Data Validation	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	deploy()		
Confirmed Fix At	https://github.com/MinaFoundation/mina-fungible-token/pull/85		

The `FungibleToken.deploy()` function initializes the contract state. As seen in the below snippet, token deployments initialize `paused` to `False`. Consequently, tokens will be active as soon as they are deployed.

```
this.admin.set(props.admin)
this.circulating.set(UInt64.from(0))
this.decimals.set(props.decimals)
this.paused.set(Bool(false))
```

Snippet 4.9: Snippet from `deploy()`

Impact Launching tokens as unpaused by default may open users to bugs in deployment scripts. Effectively, it requires that the token `admin` is properly configured when the token itself is deployed. This may not be the case for all deployments, and a careless user might leave themselves open to exploit for a brief period of time.

In the below example, we describe a situation in which a non-atomic deployment may lead to direct theft of funds. While this does require a user mistake, improving defaults may reduce the number of incidents like the one below.

For example, suppose that a user is deploying a token contract which is intended to represent shares in a vault (depositing `m` mina should mint `m` tokens if there are no tokens, or `m * vault.balance / circulating` otherwise). The admin contract should authorize mints when users deposit funds into the vault. Suppose further that the user sets up their protocol in multiple transactions, executed in this order:

1. Deploy token admin contract, with `admin.vault` state variable set to a `dummy` address.
2. Deploy token contract.
3. Deploy vault contract.
4. Set admin contract `admin.vault` state variable to vault contract address.

Between steps 2 and 3, an attacker may send funds to the `dummy` address, minting themselves tokens with no underlying funds in the actual vault. This now changes the ratio of the token's total supply to the vault's funds. After the next unsuspecting depositor deposits in the vault, they will now receive `m * vault.balance / circulating = 0` tokens instead of `m`. The attacker can now withdraw those funds from the vault.

Recommendation Add `paused` to the `FungibleTokenDeployProps` passed into the `deploy()` function, and recommend initializing to `paused` for non-atomic multi-contract deployments.

Developer Response A new `startUnpaused` flag passed to the `deploy()` function determines whether or not the contract starts as paused. It defaults to `false`.

4.1.8 V-MIN-VUL-008: Missing event on state update

Severity	Warning	**Commit**	ebbc088
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	pause(), unpause()		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/70,https://github.com/MinaFoundation/mina-fungible-token/pull/88`		

Upon a state update, it is strongly recommended that developers emit an event to indicate that a change was made. Doing so allows both external users and protocol administrators to monitor the protocol for a variety of reasons, including for potentially suspicious activity. It is therefore critical for significant changes to the protocol to be accompanied with events to enable this monitoring. The `pause()` and `unpause()` functions, however, do not emit events.

Impact If `pause()` or `unpause()` is called, protocol users and monitors may want to investigate. `pause()`-ing the protocol may indicate to users that their funds are at risk. Unexpected `pause()`-ing or `unpause()`-ing may indicate key compromise to admins.

Recommendation Add an event that indicates when `paused` has changed.

Updated Developer Response We have added functionality to emit events when the paused flag changes. We also added a `BalanceChangeEvent` which is emitted on each balance-change inside of `approveBase()`.

4.1.9 V-MIN-VUL-009: transfer() overwrites balanceChange

Severity	Warning	**Commit**	1588ee5
Type	Transaction Ordering	**Status**	Open
File(s)	`src/lib/mina/token/token-contract.ts`		
Location(s)	transfer()		
Confirmed Fix At	N/A		

The `TokenContract.transfer()` function produces two account updates: one for the sender of the funds and one for the receiver.

Rather than creating the `AccountUpdates` from scratch, it can accept a `from` and `to` `AccountUpdate`. In this case, it overwrites the `balanceChange` field to the `amount` being transferred.

```
async transfer(
  from: PublicKey | AccountUpdate,
  to: PublicKey | AccountUpdate,
  amount: UInt64 | number | bigint
) {
  // [VERIDISE] ... elided

  from.balanceChange = Int64.from(amount).neg();
  to.balanceChange = Int64.from(amount);

  let forest = toForest([from, to]);
  await this.approveBase(forest);
}
```

Snippet 4.10: Snippet from `transfer()`

However, `from` and `to` may have non-zero `balanceChanges` when passed to `transfer()`. Setting `balanceChange` overrides whatever the previous value was.

Impact If users pass an `AccountUpdate` with an existing balance change to this function generated by another application, they may experience difficult-to-debug prover errors due to the mutation.

Recommendation Use `AccountUpdate.balance.addInPlace()` and `AccountUpdate.balance.subInPlace()` instead of overwriting balances. Note that, if `from` and `to` have existing balances, this will require they are negatives of each other.

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

4.1.10 V-MIN-VUL-010: Unused and duplicate program constructs

Severity	Info	**Commit**	1588ee5, ebbc088
Type	Maintainability	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/84,https://github.com/MinaFoundation/mina-fungible-token/pull/91`		

Description The following program constructs are unused or have repeated functionality which could be abstracted into a common function.

1. `src/lib/mina/token/token-methods.ts`:
   a) `tokenMethods()`: `mint()`, `burn()`, and `send()` all recompute the derived token id before calling `getApprovedUpdate()`. Consider computing the derived token ID within the `getApprovedUpdate()` function. To avoid recomputing the derived token ID in `send()`, the developers could have `getApprovedUpdate()` take the token ID as an optional parameter, and return the computed value.

---

2. `src/lib/mina/token/token-contract.ts`:
   a) `TokenContract.approveAccountUpdate()`: This function duplicates most of the body of `approveAccountUpdates()`.

---

3. `o1js/src/lib/provable/merkle-list.ts`:
   a) `from()`, `fromReverse()`: Note that `from(array)` is simply `fromReverse([...array].reverse())`, but both functions instead implement the same logic.
   b) `MerkleListIterator`, `MerkleListIterator.Unsafe`: Both objects implement `previous()` and `next()` methods which share the majority of their logic.

---

4. `mina-fungible-token/FungibleToken.ts`:
   a) `checkPermissionEquals()`: This function manually reimplements `Provable.equal()`.
   b) `getCirculating()`, `updateCirculating()`: These functions share a large amount of functionality. While most of it is abstracted into `computeCirculating()`, they still share non-trivial amounts of logic.

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 `getCirculating()` and `updateCirculating()` functions will be removed in `https://github.com/MinaFoundation/mina-fungible-token/issues/66`.

Updated Developer Response

1. `src/lib/mina/token/token-methods.ts`:
   a) We have decided not to take the recommendation.
2. `src/lib/mina/token/token-contract.ts`:
   a) We have decided not to take the recommendation.
3. `o1js/src/lib/provable/merkle-list.ts`:
   a) We have decided not to take the recommendation.
   b) We have decided not to take the recommendation.
4. `mina-fungible-token/FungibleToken.ts`:
   a) `checkPermissionEquals()` is removed and replaced with `Provable.equal()`.
   b) The `updateCirculating` and `getCirculating` methods have been removed.

4.1.11 V-MIN-VUL-011: Best practices and recommendations

Severity	Info	**Commit**	1588ee5, ebbc088
Type	Maintainability	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/70,https://github.com/MinaFoundation/mina-fungible-token/pull/92`		

Description In the following locations, the auditors identified opportunities to improve code quality and increase adherence to TypeScript best practices:

1. `o1js/src/lib/provable/merkle-list.ts`:

   a) `MerkleList.forEach()`: Consider putting the `isDummy` argument first in the `callback`. In TypeScript, `(element: T) => void` will type-check as an instance of `(element: T, isDummy: Bool, i: number) => void`, but this almost certainly signals an error by the user.

   b) `MerkleList.popIf()`: `this.data` is supposed to restore the state from before `pop()` if the `condition` evaluates to `false`. However, if the list was empty, and the `condition` was false, then the `data` beforehand was `[]`. Restoring the state should keep it as `[]`, rather than changing it to `[{previousHash: emptyHash, element: provable.empty()}]`. Luckily, this new list is functionally the same for the purposes of a `MerkleList`. Nonetheless, restoring the original `data` may prevent prover errors in the future.

   c) `MerkleListIterator.previous()`: An inline comment mentions that `"it returns a dummy element if we're at the start of the array"`. Note that a `MerkleListIterator` has two array-like structures:

      - The underlying JavaScript array, which is in the reverse order of the list.
      - The list, represented by the sequence of hashes.

      We recommend clarifying in this comment that `previous()` returns a dummy element if at the start of the *list*, which corresponds to the end of the *array*.

   d) `MerkleList.next()`: Unlike `MerkleList.previous()`, `MerkleList.next()` does *not* prove membership in the list. Instead, it only computes the "next hash" if this element were appended to the current hash.

   This adds a very important caller requirement: users *must remember to assert the iterator is at the end of the list!* Otherwise, the circuit will be under-constrained.
   We recommend adding documentation to `next()` warning of this fact, and moving it into the `Unsafe` module. If a "safe" implementation is required, we recommend requiring users to provide a function implementing their desired invocation of `next()`, which can be executed within a context that is guaranteed to call `assertAtEnd()`.

---

2. `mina-fungible-token/FungibleToken.ts`:

   a) `FungibleToken.events.SetAdmin`: A raw `PublicKey` is used as the event type for setting an admin. This follows a different pattern than the other events, which each define a custom event type. Further, if the admin event is changed in the future, this will make future changes more invasive.

b) `FungibleToken.adminContract`: Either suffixing the name of a constructor function with "`Constructor`" or using a capital first letter makes its intended usage more clear.

c) `FungibleToken.deploy()`: Note that the initial deployment is signature-authorized, not proof-authorized. Consequently, none of the type invariants may be trusted in the initial deployment. As such, `admin` may be an invalid `PublicKey` (aka `isOdd` may be non-boolean), and `decimals` may be larger than a `UInt8`. Luckily, whenever the states are used, they are `witnessed` (either explicitly or inside of `state.get()`), which adds constraints for the type-invariants. It still may aid users and app developers to document that the deployed token contract may have invalid `decimals` to ensure no off-chain code makes assumptions about the `decimal`-value based on the verification-key alone.

d) `FungibleToken.approveBase()`: Consider documenting that forests are limited in size by `MAX_ACCOUNT_UPDATES` defined in `token-contracts.ts`.

e) General Documentation: Consider noting the following facts for developers:

   - `Mint`, `Burn`, and `Transfer` events may be triggered with amounts equal to `0`. Off-chain listeners should be sure to take this into account. For example, maintaining a list of depositors into by tracking `Mint` events requires filtering to non-zero amounts.

Impact These minor errors may lead to future developer confusion.

Developer Response

1. `o1js/src/lib/provable/merkle-list.ts`:

   a) We have decided not to take this recommendation.
   b) We have decided not to take this recommendation.
   c) We have decided not to take this recommendation.
   d) We have decided not to take this recommendation.

---

2. `mina-fungible-token/FungibleToken.ts`:

   a) An admin event type now wraps the `PublicKey`.
   b) The name of the field is now `AdminContract`.
   c) `decimals` and `admin` are now set in the `@initialize()` method.
   d) We added documentation to `approveBase()`.
   e) We have decided not to take this recommendation.

4.1.12 V-MIN-VUL-012: Use of deprecated o1js functions

Severity	Info	**Commit**	1588ee5, ebbc088
Type	Maintainability	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`, `o1js/src/lib/mina/token/token-contract.ts`		
Location(s)	calculateCirculating(), transfer()		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/91,https://github.com/o1-labs/o1js/pull/1776/files`		

1. `mina-fungible-token/FungibleToken.ts`:
   a) `calculateCirculating()` uses the `isPositive()` function to assert that the new computed total supply is non-negative. However, `isPositive()` has been deprecated in favor of `isPositiveV2()`, which does not suffer from multiple 0-representations.

---

2. `o1js/src/lib/mina/token/token-contract.ts`:
   a) `TokenContract.transfer()` uses the `Int64.neg()` function, which is deprecated in favor of `negV2()`.

Impact Deprecated functions may not be supported in future versions of o1js, requiring re-definition of the standard token implementation.

Recommendation Change the implementations to use non-deprecated functionality.

Developer Response

1. `mina-fungible-token/FungibleToken.ts`
   a) `calculateCirculating()` has been removed.
2. `o1js/src/lib/mina/token/token-contract.ts`:
   a) The new version of `neg()` is now used (`negV2()`), which will be replaced in the V2 version of o1js.

4.1.13 V-MIN-VUL-013: Comments on token design

Severity	Info	**Commit**	ebbc088
Type	Usability Issue	**Status**	Fixed
File(s)	`mina-fungible-token/FungibleToken.ts`		
Location(s)	FungibleToken		
Confirmed Fix At	`https://github.com/MinaFoundation/mina-fungible-token/pull/98/`		

This issue describes some limitations of the design point chosen for the fungible token standard.

1. Custom transfer logic is not supported, since no calls are made to the admin contract. Thus, token implementations will struggle to implement the following features:
   a) Fee on transfer. For examples, see here.
   b) Token blacklists or whitelists. For examples, see here.
   c) Circuit-breaking or transfer amount limits. For examples, see here.
2. Custom burn logic is not supported.
   Many applications may wish to maintain some invariant related to the total supply. For instance, a `wMina` token contract's admin would have a mechanism to lock or release `Mina` in return for minting or burning `wMina`. This would currently be implemented by the `wMina` admin contract having a method which calls burn on behalf of the user. However, this would only maintain the invariant `wMina supply >= locked Mina`, rather than strict equality.
   This type of invariant is generally of interest to any token representing a share of some wrapped assets.
3. Custom balanceOf logic is not supported:
   a) Rebasable (like stEth) tokens may be difficult to implement. For more examples, see here.

Impact Certain token applications may be limited, or lead to more complicated user implementations.

Recommendation Consider enabling additional admin hooks. To allow most tokens to operate without as many hooks, a flag could be added which turns on or off individual admin hooks.

If constraint or usability limitations prevent this, consider documenting some of the limitations clearly in the standard. Providing example workarounds for some of the above applications may also help developers avoid creating their own overly-complicated solutions.

Developer Response We added additional documentation.

Glossary

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

Ethereum One of the world's largest blockchains, known especially for its novel adoption of Turing-complete smart contracts to support decentralized applications. See https://en.wikipedia.org/wiki/Ethereum to learn more. 1

Merkle List Similar to a Merkle Tree, a Merkle List is a cryptogrpahic commitment to a set of values. However, openings can only occur at the end of the list, rather than at any index. See https://en.wikipedia.org/wiki/Hash_list to learn more. 1, 3, 7

Merkle Tree A cryptographic commitment to a list of values which can be opened at individual entries in the list. See https://en.wikipedia.org/wiki/Merkle_tree to learn more. 3, 35

Mina Mina Protocol is a succinct 22KB blockchain utilizing zero-knowledge proofs. See https://minaprotocol.com for more details. 1, 35

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, 2, 7

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, 7, 35

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. 1, 35

ZK Circuit zero-knowledge circuit. 1

zkApp A smart contract written for the Mina blockchain. See https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks for more. 35