OpenZeppelin | security

Fidelity Mintable Token Audit

Fidelity DIGITAL ASSETS™

March 14, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	Token
Timeline	From 2023-02-21 To 2023-02-24
Languages	Solidity

Total Issues	13 (12 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	7 (7 resolved)
Notes & Additional Information	6 (5 resolved)

Scope

We audited the fidelity/mintable-token-ethereum-contract repository at the 05fff03 commit.

In scope were the following contracts:

```
mintable-token-ethereum-contract/contracts/
├── MintableToken.sol
├── errorCoded.sol
├── mintAllocated.sol
├── restrictable.sol
├── roleManaged.sol
└── safeAccessControlEnumerableUpgradeable.sol
```

System Overview

The Fidelity Mintable Token contracts compile into a single contract, which is used as the implementation for an upgradeable proxy pattern. The Mintable Token is intended to be a stablecoin, representing US Dollars held in escrow by Fidelity.

The Mintable Token is an ERC-20-compliant token, with a few notable differences from a typical ERC-20 token.

One of these differences is that the system is upgradeable, allowing for feature changes by Fidelity in the future. Note that the ability to upgrade the system can be renounced, crystallizing the implementation and preventing any functionality changes from that point on.

The next difference is that the Mintable Token includes functions to increase and decrease the allowance, which mitigate a well-known attack vector in the ERC-20 `approve` function (described further here). The approach taken here by Fidelity is widely regarded as the solution to this attack vector.

The third difference is that the ability to transfer tokens can be restricted for specific users at Fidelity's discretion. This means any transfer of tokens to or from a restricted user is forbidden. Note that interactions with DeFi protocols that utilize the Mintable Token will also be forbidden if initiated by a restricted user. Fidelity may restrict or unrestrict users instantaneously through accounts possessing the `TOKEN_TRANSFER_CONTROLLER_ROLE`.

The fourth difference is that the Mintable Token will only be mintable and burnable by accounts controlled by Fidelity possessing the `MINTER_ROLE`. This allows Fidelity to control the circulating supply of tokens and ensure all existing tokens are accounted for.

The final key difference is that the Mintable Token is pausable. When the token is paused, users and smart contracts may not transfer tokens or change token allowances, tokens may not be minted nor burned, and the "mint allocations" of `MINTER_ROLE` holders may not be changed.

For permissioned actions, such as minting tokens or modifying the list of restricted accounts, all accounts holding the necessary roles will be initially controlled by Fidelity. Fidelity has informed us that there are secure procedures in place for managing these accounts.

Security Model & Privileged Roles

There are six privileged roles in the Mintable token system, and they are managed using OpenZeppelin's AccessControlUpgradeable library:

- `DEFAULT_ADMIN_ROLE`, which may modify role assignments for all roles in the system, including itself. This role may also unpause the system if it is paused. This role must always have at least one holder.
- `UPGRADER_ROLE`, which may trigger an upgrade to the Mintable Token contract.
- `MINT_ALLOCATOR_ROLE`, which may modify the amount of tokens any holder of the `MINTER_ROLE` may mint.
- `MINTER_ROLE`, which may mint or burn tokens. Note that minting tokens will consume the minter's assigned allowance.
- `TOKEN_TRANSFER_CONTROLLER_ROLE`, which may restrict or unrestrict accounts from transferring or interacting with the Mintable Token contract.
- `PAUSER_ROLE`, which may pause the system at any time.

Again note that all roles will initially be controlled by Fidelity. Since these privileged roles have the power to greatly affect most aspects of the Mintable Token system, there are implicit trust assumptions that Fidelity will manage the system in the best interest of their users.

- Fidelity is trusted to keep all private keys for any role-holding accounts safe and, ideally, unexposed to any internet-connected devices.
- Fidelity is trusted to monitor the system and ensure that malicious or prohibited users are immediately restricted by utilizing the `TOKEN_TRANSFER_CONTROLLER_ROLE`.
- Fidelity is trusted to manage all mint allocations and ensure that they, combined with already minted tokens, never exceed the total number of escrowed US Dollars.
- Fidelity is trusted to pause the system quickly if any malicious activity or bugs are detected.
- Fidelity is trusted to perform due diligence before upgrading the system via the `UPGRADER_ROLE`.
- Fidelity is trusted not to change the permissioned roles' structure, either by reassigning the admin roles for a given role or by creating new roles.
- Fidelity is trusted to only assign privileged roles to trusted accounts. This trust may be reinforced by the fact that the accounts belong to the Fidelity organization or are in some way incentivized to not act maliciously.

Low Severity

L-01 Inconsistent gap length

In all audited contracts with a `__gap` variable, the length of the `__gap` is 50.

The purpose of the gap is to provide unused space which future upgrades can utilize without disrupting the storage layouts of the inheriting contract. These gaps must be of a consistent, predictable length so they can be checked for accuracy.

To be consistent with OpenZeppelin's Upgradeable Contracts, the gap length should be `50 - SLOTS`, where `SLOTS` is the number of storage slots used in each contract. The `ERC20Upgradeable` contract uses five storage slots, so its `__gap` has a length of `45`.

- The `mintAllocated` contract consumes one storage slot, so its `__gap` should have a length of `49`.
- The `restrictable` contract consumes two storage slots, so its `__gap` should have a length of `48`.

Consider changing the lengths of these gap variables to be more consistent with the inherited OpenZeppelin Upgradeable contracts. Doing so will make validating the storage layout of upgrades easier and in turn make the upgrade process much less error-prone. Note that storage layouts can be checked via the `hardhat-storage-layout`.

***Update**: Resolved in pull request #4 at commit bbcf86f.*

L-02 Add `whenNotPaused` modifier to functions

In the `MintableToken` contract, the `whenNotPaused` modifier is used to restrict the calling of the internal `_beforeTokenTransfer` hook when the system is paused. This hook is called during the minting, burning, and transfer of tokens.

Within the codebase, restricting the `_beforeTokenTransfer` function call would in turn restrict calling the `transfer`, `transferFrom`, `mint`, and `burn` functions. Following the fail-fast principle, consider adding the `whenNotPaused` modifier to each of these functions' declarations explicitly.

***Update**: Resolved in pull request #13 at commit 03f9018.*

L-03 Incorrect contract name in error messages

In the `errorCoded` contract, the contract name for all the error messages is `MintableToken`. This could be misinterpreted as if the errors are coming from the `MintableToken` contract. Consider changing the contract names in the error messages to the respective contract that is triggering them.

***Update**: Resolved in pull request #5 at commit 2858550.*

L-04 Incomplete error message

In the `errorCoded` contract, the `ERR_4` error message states that the "*amount must be less than the current mint allocation*". However, in the `MintAllocated` contract, the `mint` function checks for the amount to be less than *or equal to* the current mint allocation. Consider changing the `ERR_4` error message to reflect this condition.

***Update**: Resolved in pull request #6 at commit 1f22cd5.*

L-05 Silent failures in `restrictTransfers` and `unrestrictTransfers`

The `restrictTransfers` function adds an address to the `restricted` set. Similarly, the `unrestrictTransfers` function removes an address from the `restricted` set.

However, the sets are designed to only have one instance of a specific value in them. As such, a specific address cannot be removed twice in a row, for example. The same is true for adding the same address more than once.

Attempting to `add` an address more than once to the `restricted` set will result in a `false` return value, which is ignored. Attempting to `remove` an address that is not in `restricted` will similarly return `false` and be ignored.

Following the "fail loudly" principle, consider implementing a `require` statement to verify the change to the `restricted` set has succeeded. Doing so will provide assurance and visibility for these sensitive operations. Alternatively, consider propagating the returns of the `add` and `remove` operations as return values of the `restrictTransfers` and `unrestrictTransfers` functions.

***Update**: Resolved in pull request #7 at commit 3147c5a.*

L-06 Uncalled initializers

Within the `initialize` function of `MintableToken.sol`, various inherited contracts are initialized.

However, two contracts in the inheritance tree are not being initialized here: `AccessControlUpgradeable` and `AccessControlEnumerableUpgradeable`.

Though neither of these contracts' `init` functions contains logic, for consistency with other empty `init` functions, consider calling them within the `initialize` function of `MintableToken.sol`. This will ensure that these inherited contracts are not forgotten in future development efforts.

***Update**: Resolved in pull request #8 at commit 25a8391.*

L-07 Redefined default admin role

The `RoleManaged` library redefines a `DEFAULT_ADMIN_ROLE`, which is already present in the `AccessControlUpgradeable` contract of the OpenZeppelin Upgradeable Contracts library.

While the definition of both roles is the same, the codebase uses the role inherited from the OpenZeppelin library. However, it is essential to add appropriate documentation to the `RoleManaged.DEFAULT_ADMIN_ROLE` to prevent potential issues caused by changes in the definition of either of the roles.

For improved readability, consider adding documentation to the `RoleManaged.DEFAULT_ADMIN_ROLE` stating its definition and use cases.

***Update**: Resolved in pull request #9 at commit f1fa493.*

Notes & Additional Information

N-01 Assign roles contractually

Currently, the Mintable Token system only assigns the `DEFAULT_ADMIN_ROLE` within its `initialize` function.

All other roles must be assigned through independent transactions, coming from the holder of the `DEFAULT_ADMIN_ROLE`.

Not only is assigning roles this way more expensive, but it is also more error-prone and has greater potential to expose the keys controlling the `DEFAULT_ADMIN_ROLE`. Doing so may result in unpredictable behavior, in case some transactions are not included in blocks. To decrease the likelihood of errors and increase predictability, consider including addresses for each role within the `initialize` function of `MintableToken`.

***Update**: Acknowledged. The Fidelity team stated:*

> *Fidelity Digital Assets elected to simplify the initial smart contract deployment process, which is purposely designed to be highly orchestrated to ensure security, integrity, and easy auditing of the Fidelity Mintable Token smart contract. Additional roles will be added via a formal process with approval controls and automated integrity checks within the Hardware Security Module (HSM)-based solution that manages all privileged roles and cryptographic signatures for the Fidelity Mintable Token administrator functions.*

N-02 Inconsistencies in contract filenames

The following inconsistencies were identified in the contract filenames:

- Apart from the `MintableToken` contract, the contracts and their respective filenames are in different cases. Consider naming all the contract files in "CapWords" style.
- Similarly to the `safeAccessControlEnumerableUpgradeable` contract, consider adding the "Upgradeable" suffix to all upgradeable contracts.

Consider following these conventions to comply with the Solidity Style Guide.

***Update**: Resolved in pull request #12 at commit aeb9ab5.*

N-03 `_modifyAllowance` is redundant

The `_modifyAllowance` function wraps `ERC20Upgradeable._approve` but does not apply any additional checks.

In its current form, it can be replaced with direct calls to `ERC20Upgradeable._approve` without affecting the system's functionality. Doing so will reduce redirection in the codebase and make it clearer.

Consider replacing the instance within `increaseAllowance` and the instance within `decreaseAllowance` with direct calls to `ERC20Upgradeable._approve`, as well as deleting the `_modifyAllowance` function.

***Update**: Resolved in pull request #11 at commit d93dde1.*

N-04 Declare `int`/`uint` as `int256`/`uint256`

There are two instances where `int` or `uint` are used instead of `int256` or `uint256`:

- On line 120 of `MintableToken.sol`
- On line 75 of `mintAllocated.sol`

To favor explicitness, all instances of `int` or `uint` should be declared as `int256` or `uint256`.

***Update**: Resolved in pull request #3 at commit d241b47.*

N-05 Unused error messages

To improve readability, consider removing the unused `ERR_8` and `ERR_9` error messages from the `ErrorCoded` library.

***Update**: Resolved in pull request #2 at commit ee46d83.*

N-06 Unused imports

Throughout the codebase, imports on the following lines are unused:

- Import `PausableUpgradeable` of `MintableToken.sol`
- Import `mintAllocated` of `MintableToken.sol`

- Import `errorCoded` of `MintableToken.sol`
- Import `roleManaged` of `MintableToken.sol`
- Import `EnumerableMapUpgradeable` of `restrictable.sol`
- Import `EnumerableMapUpgradeable` of `safeAccessControlEnumerableUpgradeable.sol`
- Import `errorCoded` of `safeAccessControlEnumerableUpgradeable.sol`
- Import `roleManaged` of `safeAccessControlEnumerableUpgradeable.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

***Update**: Resolved in pull request #10 at commit 4e77e26.*

Conclusions

The auditors found the codebase to be clean and easily understandable. Several low-severity issues and notes were identified.

Some high-level considerations for secure management of the codebase are provided below:

- Consider utilizing private mempool and transaction ordering services when broadcasting transactions to Ethereum. This may lower the potential for malicious actors to front-run sensitive operations to their advantage (e.g., transferring funds from an address that is about to be restricted).
- Consider simulating sensitive changes to role holder mappings before broadcasting them, to ensure the correct addresses receive roles.
- Advise projects seeking to integrate with this codebase, and users seeking to use Mintable Tokens, that both `tx.origin` and `msg.sender` may be restricted. The use of `tx.origin` in this way is non-standard.
- Before any upgrades, manually check that the storage layout of the contracts creates no overwrites. `hardhat-storage-layout` can be used to assist in this.
- Before any upgrades, simulate the entire upgrade process offline and run all tests on the upgraded contract. Utilize OpenZeppelin Upgrades Plugins to ensure the upgrades are safe.
- Once deployed for public use, publicly announce any planned changes to the role structure or the total token supply.

Appendix

Monitoring Recommendations

To guarantee the secure functioning of the protocol and prompt responses from privileged roles, it is advisable to monitor the deployed contracts for the following:

- Monitor the total supply of Mintable Tokens at all times, checking for the `Mint` and `Burn` events in each block.
- Periodically ensure access to the `PAUSER_ROLE` key, checking that it can be accessed quickly if needed. Ensure there is a plan in place for activating the pause functionality if it is needed, and all involved members of the Fidelity organization are aware of the plan.
- Monitor the mempool for transactions calling `grantRole`, or emissions of the `RoleGranted` and `RoleAdminChanged` events. Ensure that every emission coincides with a planned change to the role structure.
- Monitor for transaction failures related to restricted accounts (reverts returning `ERR_1`) to detect malicious use of the protocol. Monitoring for this may also help detect incompatibilities with certain DeFi protocols. If an incompatibility is detected, consider publicly announcing this to all users.