Bartercard

Qoin Token

Smart Contract Security Assessment

Version: 2.0

Contents

Introduction **2**
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary **4**

Detailed Findings **6**

Summary of Findings **7**
- Permissive Access Control Model . . . 8
- `Qoin` Token Minting and Burning Possible While Smart Contract is Paused . . . 9
- `Logger` Contract Comments . . . 10
- Unncessary Use of Gas Station Network Framework . . . 11
- ERC20 Implementation Vulnerable to Front-Running . . . 12
- Miscellaneous General Statements . . . 14

A Test Suite **16**

B Vulnerability Severity Classification **17**

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the `Qoin` token smart contract. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the `Qoin` token contract contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the `Qoin` token smart contract.

Overview

The `Qoin` smart contract represents a utility token that will serve as a payment mechanism for merchants within the Bartercard ecosystem. `Qoin` tokens will be used as an internal accounting metric to track transactions between merchants, in the context of a private chain (permissioned Ethereum-based Blockchain). Bartercard will act as an issuer, and will ***mint*** tokens to the relevant parties as required.

The `Qoin` tokens have the following properties:

- Privileged users (i.e. Administrators) cannot modify account balances
- The total `Qoin` supply is capped at 10 billion tokens
- The token contract leverages the OpenZeppelin framework (*Version 2.4.0*) by inheriting the following smart contracts:
  - `Context`
  - `ERC20`
  - `ERC20Detailed`
  - `ERC20Mintable`
  - `ERC20Capped`

  - `ERC20Pausable`
  - `ERC20Burnable`
- The deployer of the `Qoin` token contract can mint tokens and assign them to arbitrary recipients. Additionally, the deployer is also able to pause the smart contract supporting the `Qoin` tokens.
- The deployer can add new accounts (i.e. Ethereum addresses) as *minters* and/or *pausers*.
- A dedicated smart contract, `Logger`, is used to keep track of all addresses who have received any `Qoin` tokens. Indeed, this smart contract logs all transfer and minting recipients in an array of addresses (`loggedAddresses`). This feature will be leveraged to reconstruct the state of the `Qoin` account balances in case of a token upgrade.

Security Assessment Summary

This review was conducted on commit `04f3f64be531b0a1c2c29b638b2ae26b79acfeb9` (repository shared via email, for which the SHA256 signature is `11cfea3be8d5981da7a891f5550b627d276def3a0b70079f0731b58907ce355c`), which contains the file `Qoin_flattened.sol`.

The complete list of contracts contained in `Qoin_flattened.sol` is as follows:

```
└── Qoin_flattened.sol
    ├── Qoin
    ├── Context (OpenZeppelin)
    ├── SafeMath (OpenZeppelin)
    ├── ERC20 (OpenZeppelin)
    ├── ERC20Detailed (OpenZeppelin)
    ├── Roles (OpenZeppelin)
    ├── MinterRole (OpenZeppelin)
    ├── PauserRole (OpenZeppelin)
    ├── ERC20Capped (OpenZeppelin)
    ├── ERC20Mintable (OpenZeppelin)
    ├── ERC20Pausable (OpenZeppelin)
    ├── ERC20Burnable (OpenZeppelin)
    └── Logger
```

This security assessment targeted exclusively the following contracts:

- `Qoin`
- `Logger`

All other contracts were considered out of scope.

The manual code review section of the reports, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focuses on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

The testing team identified a total of six (6) issues during this assessment, of which:

- One (1) is classified as low risk,
- Five (5) are classified as informational.

These issues have all been acknowledged by the development team.

To support this review, the testing team used the following automated testing tools:

- Rattle: https://github.com/trailofbits/rattle
- Mythril: https://github.com/ConsenSys/mythril

- Slither: https://github.com/trailofbits/slither
- Surya: https://github.com/ConsenSys/surya

Output for these automated tools is available upon request.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the `Qoin` token smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including comments not directly related to the security posture of the token contract, are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
QOIN-01	Permissive Access Control Model	Low	**Closed**
QOIN-02	`Qoin` Token Minting and Burning Possible While Smart Contract is Paused	Informational	**Closed**
QOIN-03	`Logger` Contract Comments	Informational	**Closed**
QOIN-04	Unncessary Use of Gas Station Network Framework	Informational	**Closed**
QOIN-05	ERC20 Implementation Vulnerable to Front-Running	Informational	**Closed**
QOIN-06	Miscellaneous General Statements	Informational	**Closed**

QOIN-01	Permissive Access Control Model		
Asset	`Qoin`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `Qoin` smart contract defines two distinct types of privileged accounts:

- **Minters**: Ethereum accounts allowed to mint Qoin tokens to arbitrary addresses by calling the related `mint()` function;
- **Pausers**: Ethereum accounts allowed to pause the Qoin token contract by calling the related `pause()` function.

The account that deployed the Qoin token contract (by calling the `constructor()` function) is granted both privileges and is allowed to add additional **minters** and **pausers**.

After granting arbitrary Ethereum accounts these privileges, the deployer can no longer revoke these permissions. Indeed, an account can renounce the **minter** and **pauser** roles (by calling `renounceMinter()` and `renouncePauser`), but there is no way for the contract deployer to *downgrade* the permissions of an account.

As a result, if a minter/pauser account is compromised (e.g. associated private key leaked), the contract would be in an insecure state as the malicious actor/attacker could perform privileged actions, with no way for the contract deployer to stop them. A new `Qoin` token contract would therefore have to be deployed.

Recommendations

Consider implementing additional roles to manage the access control on the `Qoin` token contract. Specifically, `MinterAdmin()` and `PauserAdmin()` roles could allow Bartercard to manage and revoke privileged accounts.

Resolution

The development team acknowledged the issue and provided the below response:

"Minter Roles should not be granted lightly, and only two accounts will be configured with the minting function.

- *(The initial Contact Creator), If it is established the Private Key for this user is compromised then that user would revoke their access and the secondary Key would resume minting*
- *(A Secondary trusted Account) the keys and password for this user would be kept in a safe and not used unless the Initial Contract Creator has revoked the first key*

By restricting the minting function to trusted entities, we mitigate the risk of the key holder not revoking their access when requested."

QOIN-02	`Qoin` Token Minting and Burning Possible While Smart Contract is Paused
Asset	`Qoin`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `mint()` and `burn()` functions are not protected by the `whenNotPaused` modifier. As a result, it is possible for an Ethereum account with the **minter** role to mint and burn tokens while the contract is in a `paused` state.

Refer to our test suite (`test_pausing.py`) for a proof-of-concept.

Recommendations

Ensure that this behaviour is intended. If not, add the `whenNotPaused` modifier to the `mint()` and `burn()` functions.

Resolution

The development team acknowledged the issue and provided the below response:

"The ability to Mint coins when the contract is paused is not seen as an issue. The minter accounts need to be notified when the contract is paused."

QOIN-03	`Logger` Contract Comments
Asset	`Logger`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Logger` contract allows keeping track of all addresses who have received any `Qoin` tokens. Indeed, this smart contract logs all transfer and minting recipients in an array of addresses (`loggedAddresses`). This feature is intended to be used in the event of a token upgrade, whereby Bartercard can have access to all Ethereum addresses who received `Qoin` tokens.

The testing team identified the following points during this assessment:

1. The `Logger` contract logs addresses that have received 0 `Qoin` tokens. Indeed, the logAddress modifier is triggered for 0 value transfers. This means that any Ethereum account can add any address to the `loggedAddresses` array (including accounts who have never received any `Qoin` tokens);
2. The `Logged` event emitted by the `Logger` contract does not define a parameter name;
3. The `Logger` contract does not keep track of allowances and spenders.

Refer to our test suite (`test_logger.py`) for more details.

Recommendations

1. Ensure that this behaviour is acceptable. If required, consider preventing 0 value transfers;
2. Consider providing the `Logged` event with a variable name for clarity purposes;
3. Confirm that allowances and spenders will be kept off-chain.

Note that using an archive node also allows for the reconstruction of a smart contract state at any given block height. By leveraging the `Transfer` events, Bartercard can perform a token upgrade based on the existing balances.

Resolution

The development team acknowledged the issue and provided the below response:

QOIN-04	Unncessary Use of Gas Station Network Framework
Asset	`Qoin`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Qoin` token contract inherits OpenZeppelin's `Context` smart contract. This particular module allows smart contracts to support *Meta-transactions*, i.e. transactions for which the gas cost is paid by a different user than the transaction initiator.

Specifically, the `Context` smart contract defines two functions that provide information about the current execution context, including the sender of the transaction and its data. This module is part of the Gas Station Network framework proposed by OpenZeppelin to standardise the use of *Meta-transactions*.

The `Qoin` token contract does not support meta transactions, and as such, does not need to leverage modules of this framework.

This issue does not pose a direct security risk to the `Qoin` contract.

Recommendations

The testing team understands that the OpenZeppelin version used by the `Qoin` token contract incorporates the `Context` module by default. Consider using a previous version of the OpenZeppelin libraries that do not make use of the Gas Station Network framework.

Resolution

The development team acknowledged the issue and provided the below response:

"This issue is not a security concern and does not apply to the QOIN smart contract as the GSN framework is not used."

QOIN-05	ERC20 Implementation Vulnerable to Front-Running
Asset	`Qoin`
Status	**Closed:** See Resolution
Rating	Informational

Description

Front-running attacks [3, 4] involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction.

The ERC20 implementation is known to be affected by a front-running vulnerability, in its `approve()` function.

Consider the following scenario:

1. Alice approves Malory to spend 1000 QOIN, by calling the `approve()` function on the Qoin token contract;
2. Alice wants to reduce the allowance, from 1000 QOIN to 500 QOIN, and sends a new transaction to call the `approve()` function, this time with `500` as a second parameter.
3. Malory watches the blockchain transaction pool and notices that Alice wants to decrease the allowance. Malory proceeds with sending a transaction to spend the 1000 QOIN, with a higher gas price, which is then mined before Alice's second transaction;
4. Alice's second `approve()` happens, effectively providing Malory with an additional allowance of 500 QOIN;
5. Malory spends the additional 500 QOIN. As a result, Malory spent 1500 QOIN, when Alice wanted her to only spend 500 QOIN.

We acknowledge that the `increaseAllowance()` and `decreaseAllowance()` functions have been added to this contract to mitigate this attack vector.

Recommendations

Be aware of the front-running issues in `approve()` and potentially add extended approve functions which are not vulnerable to the front-running vulnerability.

A further method typically used to address this attack vector is to force users to change the allowance to 0 first, before updating it to the desired value (requires two separate transactions). See the `safeApprove()` function in OpenZeppelin's `SafeERC20` contract.

Resolution

The development team acknowledged the issue and provided the below response:

"The possibility of this attack vector is noted, and Approved spenders is not currently a function in the wallet. The functions of `increaseAllowance()` *and* `decreaseAllowance()` *provide the same functionality as Approve but without the front running vulnerability."*

QOIN-06	Miscellaneous General Statements
Asset	`Qoin & Logger`
Status	**Closed:** See Recommendations
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment:

1. The version of the OpenZeppelin contracts in use has not been security reviewed. The latest security audit conducted against the OpenZeppelin contracts targeted version `2.0.0` (in October 2018);
2. The ERC20 standard does not prevent users from inadvertently sending tokens to smart contracts addresses that don't have a way to withdraw/transfer those tokens. This can lead to significant value loss as described here;
3. The `Qoin` token contracts do not use the latest version of the Solidity compiler (version in use: `0.5.11`, latest version available: `0.5.12`);
4. The function `loggedAddressesLength()` in the `Logger` contract could be declared as an `external` function instead of a `public` function to save gas.

Recommendations

1. Consider using an audited version of the OpenZeppelin contracts (version `2.0.0` does not incorporate the Gas Station Network framework, see QOIN-04);
2. Consider implementing ERC777 (or ERC223) which both prevent users from sending tokens to contracts that fail to implement a `tokensReceived()` function;
3. Consider upgrading the version of the Solidity compiler in use to the latest version available (*Note: version 0.5.12 does not fix any security issues exploitable in the context of the `Qoin` token contract. However, it is recommended to use the latest available version of the Solidity compiler*);
4. Replace the `loggedAddressesLength()` function visibility to `external` for gas saving purposes.

Recommendations

The development team acknowledged these informational comments and provided the below response:

1. *"The latest OpenZeppelin is currently on a General Release code level 2.4.0 this code has not been through a security Audit and is out of the scope of this audit. If we consider this an unacceptable risk then we should rebuild using OpenZeppelin 2.0. James Zaki(AlphaWallet) was happy to proceed with version 2.4, and we are happy to proceed with James' recommendation.*
2. *This is a feature of ERC20 token and would require a redesign of the Blockchain.*

3. *This is not relevant to our Smart Contract*
4. *This is an accepted recommendation and will be implemented if the QOIN contract is updated.*"

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `pytest` framework was used to perform these tests and the output is given below.

```
tests/test_approvals.py::test_transfer_approval                                PASSED  [8%]
tests/test_approvals.py::test_transfer_approval_insufficient_allowance         PASSED  [16%]
tests/test_deploy.py::test_deploy                                              PASSED  [25%]
tests/test_events.py::test_transfer_event                                      PASSED  [33%]
tests/test_events.py::test_approval_event                                      PASSED  [41%]
tests/test_events.py::test_access_control_events                               PASSED  [50%]
tests/test_events.py::test_pausing_events                                      PASSED  [58%]
tests/test_logger.py::test_log_address                                         PASSED  [66%]
tests/test_pausing.py::test_pausing_mechanism                                  PASSED  [75%]
tests/test_transfers.py::test_valid_transfers                                  PASSED  [83%]
tests/test_transfers.py::test_transfer_to_zero_address                         PASSED  [91%]
tests/test_transfers.py::test_transfer_insufficient_balance                    PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

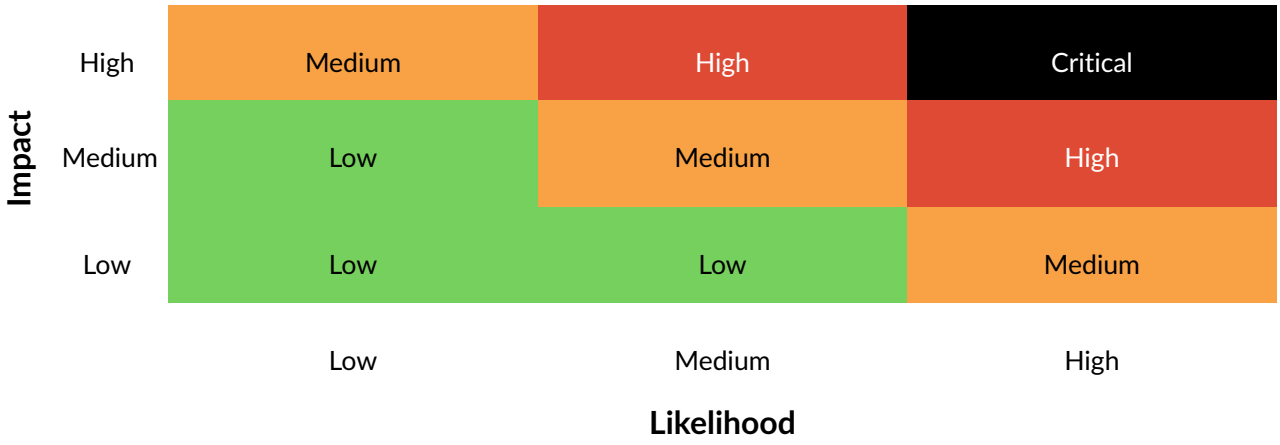

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[3] Sigma Prime. Solidity Security - Front Running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[4] NCC Group. DASP - Front Running. Website, 2018, Available: http://www.dasp.co/#item-7. [Accessed 2018].