PUBLIC

Security Audit

of MONART TOKEN Smart Contracts

August 28, 2019

Produced for

by

CHAINSECURITY

Table Of Contents

Foreword

We would first and foremost like to thank MONART for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations and results.

– ChainSecurity

Executive Summary

MONART engaged CHAINSECURITY to perform a security audit of MONART TOKEN, an Ethereum-based smart contract. The MONART TOKEN smart contract is the base layer of an tokenized art ecosystem that MONART is establishing.

CHAINSECURITY audited the smart contracts which are going to be deployed on the public Ethereum chain. Audits of CHAINSECURITY use state-of-the-art tools for detection of generic vulnerabilities and checks of custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure the highest security standards. The MONART TOKEN is completely based on the OpenZeppelin library. During the audit, CHAINSECURITY only found minor issues.

All reported issues have been addressed by MONART. In particular, all security and design issues have been eliminated with appropriate code fixes.

Audit Overview

Methodology of the Audit

CHAINSECURITY's methodology in performing the security audit consisted of four chronologically executed phases:

1. Understanding the existing documentation, purpose, and specifications of the smart contracts.
2. Executing automated tools to scan for common security vulnerabilities.
3. Manual analysis by one of our CHAINSECURITY experts covering both security and functional correctness (based on the provided documentation) of the smart contracts.
4. Formalizing security and correctness properties that capture the intended behavior of the smart contracts and checking these using analysis tools for Ethereum smart contracts.
5. Preparing the report with checked properties, vulnerability findings, and potential exploits.

Scope

Source code files received	August 8, 2019
Git commit	`rar file provided`
EVM version	not defined
Initial Compiler	SOLC compiler, version >=0.4.25 <0.6.0
Final code update received	August 27, 2019
Final commit	`848c28948b626a2c1329e8ed6e0c091636b743fe`
Final Compiler	version 0.4.26 fixed in truffle config

The scope of the audit is limited to the following source code files.

In Scope	File	SHA-256 checksum
☑	./contracts/MonartToken.sol	38801c0d31f4ca210407ab10d875de4604123c845164d5246c9d9a49538472c8

For these files the following categories of issues were considered:

In Scope	Issue Category	Description
☑	Security Issues	Code vulnerabilities exploitable by malicious transactions
☑	Trust Issues	Potential issues due to actors with excessive rights to critical functions
☑	Design Issues	Implementation and design choices that do not conform to best practices

Audit Tasks

The security audit conducted by CHAINSECURITY consisted of the following tasks:

- Formalizing functional requirements and verifying the correctness of the MONART TOKEN contract with respect to the formalized properties.
- Analyzing the MONART TOKEN contract for generic security vulnerabilities.
- A thorough manual audit of the MONART TOKEN contract for compliance with best security practices.

Terminology

For the purpose of this audit, CHAINSECURITY has adopted the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business-related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorise the findings into four distinct categories, depending on their severities:

- Low: can be considered less important
-  Medium: should be fixed
-  High: we strongly recommend fixing it before release
-  Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the table below, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High			
Medium			
Low			

During the audit, concerns might arise or tools might flag certain security issues. After carefully inspecting the potential security impact, we assign the following labels:

- ✓ No Issue: no security impact
- ✓ Fixed: the issue is addressed technically, for example by changing the source code
- ✓ Addressed: the issue is mitigated non-technically, for example by improving the user documentation and specification
- ✓ Acknowledged: the issue is acknowledged and it is decided to be ignored, for example due to conflicting requirements or other trade-offs in the system

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview of what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities; even an audit in which no vulnerabilities are found is not a guarantee of a secure smart contract. However, auditing enables discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. This is why we carry out a source code review aimed at determining all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

System Overview

MONART TOKEN is a standard ERC20 token. The token has a maximum supply of 100000000000 MONART TOKEN and two decimals. No minting or burning is possible. At deployment, all 100000000000 MONART TOKEN are pre-minted and deposited into the deployer's account.

Best Practices in MONART's project

We list Hard Requirements which must be fulfilled in order for us to start the audit. Furthermore, we provide a list of best practices, the following of which allows the audit efforts to be focused on project-specific issues.

Hard Requirements

The fulfillment of the requirements below ensures that CHAINSECURITY can start the audit.

- ☑ **All files and software for the audit have been provided to CHAINSECURITY**
  The project needs to be complete. Code must be frozen and the relevant commit or files must have been sent to CHAINSECURITY. All third party code (like libraries) and third-party software (like the solidity compiler) must be exactly specified or made available. Third party code can be located in a folder separated from client code (and the separation needs to be clear) or included as dependencies. If dependencies are used, the version(s) need to be fixed.
- ☑ The code must compile and the required compiler version must be specified. When using outdated versions with known issues, clear reasons for using these versions are being provided.
- ☑ There are migration/deployment scripts executable by CHAINSECURITY and their use is documented.
- ☑ The code is provided as a Git repository to allow reviewing of future code changes.

Best Practices

These requirements are complimentary to the hard requirements and can further enhance the efficiency of the audit.

- ☑ There are no compiler warnings, or warnings are documented.
- ☑ Code duplication is minimal, or justified and documented.
- ☐ The output of the build process (including possible flattened files) is not committed to the Git repository.
  EXPLANATION: Node modules and build folder was part of the submitted code.
- ☑ The project only contains audit-related files, or, if this is not possible, a meaningful distinction is made between modules that have to be audited and modules that CHAINSECURITY should assume are correct and out-of-scope.
- ☑ There is no dead code.
- ☑ The code is well-documented.
- ☑ The high-level specification is thorough and enables a quick understanding of the project without any need to look at the code.
- ☑ Both the code documentation and the high-level specification are up-to-date with respect to the code version CHAINSECURITY audits.
- ☑ Functions are grouped together according to either the Solidity guidelines[2], or to their functionality.

Smart Contract Test Suite

In this section, CHAINSECURITY comments on the smart contract test suite of MONART TOKEN. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

There are no tests available.

[2] https://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions

Formal Functional Properties

CHAINSECURITY investigated selected functional requirements. Below, we list the considered requirements and indicate whether they have been successfully verified (marked with label ✓ Verified) or not (marked with label ✕ Does not hold).

Property syntax and semantics

The formalization of the properties uses the syntax of Solidity extended with temporal operators (such as `always` and `prev`) and additional logical connectives (such as implication, written with `==>`). For example, `always` quantifies over all contract states and `prev` refers to the previous state (before a transaction has executed).

The expression `FUNCTION == Token.transfer(address,uint256)` evaluates to `true` if the current transaction is a call to function `transfer(address,uint256)`. The expression `Token.transfer(address,uint256)[0]` returns the first argument of the function `Token.transfer(address,uint256)`. For brevity, we name arguments in function declarations and refer to them by their name. For example, given a function with named arguments `Token.transfer(address to, uint256 tokens)`, we write `to` to refer to its first argument.

The properties are interpreted over a sequence of contract states, induced by executing a given sequence of transactions. A property is *verified* if it holds for any possible sequences of contract states.

Properties related to the MONART contract

In the formalization of the properties below, the symbols `X` and `Y` represents arbitrary values (such as `0x123`).

2.1 Constant total supply ✓ Verified

THe total supply of tokens must be constant.

```
always( prev(MonartToken.totalSupply_) == MonartToken.totalSupply_);
```

2.2 Correct total supply ✓ Verified

The total supply of tokens always equals 100000000000.

```
always( MonartToken.totalSupply_ == 100000000000);
```

We note that this property implies the *Constant total supply* property given above.

2.3 Compliant transfer function ✓ Verified

The transfer function must comply to the ERC20 standard.

- The balance of the transaction sender is decreased by the provided amount of tokens.
- The balance of the receiver `to` is increased by the provided amount of tokens.

```
always(
  (FUNCTION == MonartToken.transfer(address to,uint256 tokens)) ==>
    (
      (
        (MonartToken.balances[msg.sender] ==
           (prev(MonartToken.balances)[msg.sender] - tokens)) &&
        (MonartToken.balances[to] == (prev(MonartToken.balances)[to] +
           tokens))
      ) ||
      (msg.sender == to)
    )
);
```

The condition `(msg.sender == to)` indicates that the balances must not increase/decrease if the transaction sender equals the receiver.

2.4 Compliant transferFrom function ✓ Verified

The transferFrom function must comply to the ERC20 standard.

- The balance of the `from` account is decreased by the provided amount of tokens.
- The balance of the receiving address `to` is increased by the provided amount of tokens.
- The allowance provided to the transaction sender was greater than the provided amount of tokens.
- The allowance provided to the transaction sender is decreased by the provided amount of tokens.

```
always(
  (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
     tokens)) ==>
  (
    (from == to) ||
    (
      (MonartToken.balances[from] == (prev(MonartToken.balances)[from] -
         tokens)) &&
      (MonartToken.balances[to] == (prev(MonartToken.balances)[to] + tokens))
    )
  )
);
```

```
always(
  (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
     tokens)) ==>
  (
    (prev(MonartToken.allowed)[from][msg.sender] >= tokens) &&
    (
      MonartToken.allowed[from][msg.sender] ==
         (prev(MonartToken.allowed)[from][msg.sender] - tokens)
    )
  )
);
```

2.5 Allowance increase safety ✓ Verified

The allowances provided to `Y` by `X` may be increased only by `X`.

```
always(
    (prev(MonartToken.allowed[X][Y]) < MonartToken.allowed[X][Y])
      ==> (msg.sender == X)
);
```

The above property states that if the allowance provided to a user `Y` by user `X` increases, then the transaction was made by user `X`.

2.6 Allowance decrease safety ✓ Verified

Only the issuer or the aquirer of an allowance may decrease the allowed amount.

```
always(
    (prev(MonartToken.allowed[X][Y]) > MonartToken.allowed[X][Y])
      ==> ((msg.sender == X) || (msg.sender == Y))
);
```

2.7 Balance decrease safety ✓ Verified

The balance of a user `X` may decrease only by:

- Transactions made by user `X`;
- Transactions to function `transferFrom(address from, address to, uint256 tokens)` where `from` equals `X` and the allowance provided to the transaction sender by `X` exceeds the provided amount of tokens.

```
always(
  (prev(MonartToken.balances[X]) > MonartToken.balances[X]) ==>
  (
    (msg.sender == X) ||
    (
      (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
         tokens)) &&
      (from == X) &&
      (prev(MonartToken.allowed)[X][msg.sender] >= tokens)
    )
  )
);
```

2.8 No overflow in transfer ✓ Verified

Function `transfer` must not decrease the balance of the token receiver.

```
always(
    (FUNCTION == MonartToken.transfer(address to,uint256 tokens)) ==>
    (MonartToken.balances[to] >= prev(MonartToken.balances)[to])
);
```

2.9 No underflow in transfer ✓ Verified

Function `transfer` must not increase the balance of the transaction sender.

```
always(
    (FUNCTION == MonartMonartToken.transfer(address,uint256)) ==>
    (MonartToken.balances[msg.sender] <=
       prev(MonartToken.balances)[msg.sender])
);
```

2.10 No overflow in transferFrom for balances ✓ Verified

Function `transferFrom` must not decrease the balance of the receiving address `to`.

```
always(
    (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
       tokens)) ==>
    (MonartToken.balances[to] >= prev(MonartToken.balances)[to])
);
```

2.11 No underflow in transferFrom for balances ✓ Verified

Function `transferFrom` must not increase the balance of the `from` account.

```
always(
    (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
       tokens)) ==>
    (MonartToken.balances[from] <= prev(MonartToken.balances)[from])
);
```

2.12 No underflow in transferFrom for allowance ✓ Verified

Function `transferFrom` must not increase the allowed amount of the transaction sender.

```
always(
    (FUNCTION == MonartToken.transferFrom(address from, address to, uint256
       tokens)) ==>
    (
        prev(MonartToken.allowed)[from][msg.sender] >=
           MonartToken.allowed[from][msg.sender]
    )
);
```

2.13 Correct balances ✓ Verified

The sum of balances must always equal the total supply of tokens.

```
always(SUM(MonartToken.balances) == MonartToken.totalSupply_);
```

Security Issues

This section reports the security issues found during the audit.

Solidity compiler version is not fixed and consistent L ✓ Addressed

MONART uses a floating pragma `pragma solidity >=0.4.25 <0.6.0;`. This spans over breaking compiler changes. The OpenZeppelin library files specify version `pragma solidity ^0.4.21;`. Hence, it it is not consistent and will not work for many versions.

Furthermore, contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively[3]. This issue is about locking the version. The addendum additionally mentions the problem about outdated compiler versions.

Likelihood: Low
Impact: Medium

Addressed: MONART did not solve this issue in the solidity file but fixed the compiler version in the truffle config file. As long as the code is deployed with exactly this truffle setup, the compiler is fixed to version `0.4.26`

Use strict truffle config solc compiler version L ✓ Fixed

The pragma in the `MonartToken.sol` contract is set to `pragma solidity >=0.4.25 <0.6.0;` as mentioned before. In the latest truffle version it is also possible to set the specific solc compiler to use in the `truffle-config.js` file. MONART has not defined a static version. Because the version is not fixed in the solidity files, it is not clear which specific solc compiler version is used in the project. MONART is advised to set a static consistent version in the solidity files and in the truffle config.

Likelihood: Low
Impact: Low

Fixed: MONART solved the issue by fixing the compiler version in the truffle config file to version `0.4.26`.

[3] https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md%7D%7D

Trust Issues

This section reports functionality that is not enforced by the smart contract and hence correctness relies on additional trust assumptions.

CHAINSECURITY has no concerns to raise in this category of the report.

Design Issues

This section lists general recommendations about the design and style of MONART's project. These recommendations highlight possible ways for MONART to improve the code further.

State variables not declared constant L ✓ Fixed

MONART declares multiple state variables in the `MonartToken` contracts which have fix values and cannot be changed. These are:

```
string public name = 'MonartToken';
string public symbol = 'MART';
uint public decimals = 2;
uint public INITIAL_SUPPLY = 100000000000;
```

These variables should be declared constant by adding the keyword `constant`. Adding the keyword `constant` saves gas[4] because the compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective constant expression (which might be computed to a single value by the optimizer).

Fixed: MONART solved the issue by adding the keyword `constant` to the variable declarations.

[4] https://solidity.readthedocs.io/en/v0.5.3/contracts.html#constant-state-variables

Recommendations / Suggestions

- MONART uses the OpenZeppelin library and specifies version `^1.9.0` in the `package.json`. The lock file shows that the actual version in the project is `1.9.0`. The current version is `2.3.0`. The latest version has breaking changes compared to version `1.9.0`. Furthermore, the `SafeMath` library changed from using `assert` to `require`. MONART might consider using up to date library files. If MONART decides to stay with version 1, the reasons for this choice should be clearly documented.
- No optimization settings are specified inside the truffle configuration file. The default configuration value is to run the optimizer optimizing for 200 contract executions during the life cycle of the contract. Low values optimize for gas deployment costs and high values for gas usage costs. CHAINSECURITY recommends to explicitly define these settings.
- MONART uses truffle. Since truffle v. 5 it is possible to specify an EVM version. MONART might consider setting the EVM version to ensure that tests and audits (verification) are always done with the same EVM version.

Addendum and general considerations

Blockchains and especially the Ethereum Blockchain might often behave differently from common software. There are many pitfalls which apply to all smart contracts on the Ethereum blockchain.

CHAINSECURITY mentions general issues in this section which are relevant for MONART's code, but do not require a fix. Additionally, CHAINSECURITY mentions information in this section, to clarify or support the information in the security report. This section, therefore, serves as a reminder to create awareness for MONART and potential users.

Outdated compiler version

CHAINSECURITY could not find obvious issues with the compiler version MONART is using. MONART uses SOLC compiler, version >=0.4.25 <0.6.0. If MONART is aware of the compiler's behavior and bugs, there might be good reasons for using an older compiler version. While the latest version does contain bug fixes, it might introduce new bugs.

CHAINSECURITY does, however, recommend to use the same compiler version homogeneously throughout the project and to use the compiler version for deployment that was used during testing. Furthermore, for any used version it is helpful to monitor the list of known bugs[5].

Forcing ETH into a smart contract

Regular ETH transfers to smart contracts can be blocked by those smart contracts. On the high-level this happens if the according solidity function is not marked as `payable`. However, on the EVM levels there exist different techniques to transfer ETH in unblockable ways, e.g. through `selfdestruct` in another contracts. Therefore, many contracts might theoretically observe "locked ETH", meaning that ETH cannot leave the smart contract any more. In most of these cases, it provides no advantage to the attacker and is therefore not classified as an issue.

[5] https://solidity.readthedocs.io/en/develop/bugs.html

Disclaimer

UPON REQUEST BY MONART, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..