PUBLIC

Security Audit

of PAXOS-GOLD Smart Contract

June 13, 2019

Produced for

PAXOS

by

CHAINSECURITY

Table Of Contents

Foreword

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

– ChainSecurity

Executive Summary

PAXOS engaged CHAINSECURITY to perform a security audit of PAXOS-GOLD, an Ethereum-based smart contract system. The PAXOS-GOLD smart contracts provide a gold-collateralized ERC20 stablecoin. PAXOS is able to mint and burn PAXOS GOLD TOKENs according to the supply of gold. Furthermore, the contract contains a variable fee rate, controlled by PAXOS, which is applied to each transfer.

Since previous versions of the code have already been audited multiple times, CHAINSECURITY only audited specific parts of the `PAXGImplementation` smart contract. Specifically, CHAINSECURITY audited the not yet audited code related to the fee model. Ensuring that the gold supply matches the token supply is out of the scope of this audit.

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.

CHAINSECURITY did not find any severe issues. The few issues that were found, were all of low or medium severity. PAXOS fixed all of the reported issues.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the changes regarding the fee. They have been reviewed based on SOLC compiler, version 0.4.24 and EVM version PETERSBURG. All of these source code files were received on May 23, 2019 as part of the git commit `37efebf8a8c5db66dddbce4b344939cc54eb1cef`. The latest update has been received on June 12, 2019 with the commit `1eef367250be47cf4a6a7c50cf73d67b8bff9400`.

File	SHA-256 checksum
contracts/PAXGImplementation.sol	`30be26d4a9b0d6a7a6bb9923697ff6e6a5ee6479ce8ed68567c58649cc59872f`

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

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

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:

- L Low: can be considered less important
- M Medium: should be fixed
- H High: we strongly suggest fixing it before release
- C 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.

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

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

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 ot 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.

System Overview

The audit of CHAINSECURITY was restricted to PAXOS's fee model. The fee model is newly added to the existing contracts and has the following components:

- A new role, `feeController`, was introduced which governs the `feeRate` and the `feeRecipient`.
- The `feeRecipient` will receive all fees and can be adjusted over time by the `feeController`.
- The `feeRate` is given in a hundredth of a basis point and the expected fee can be checked using the `getFeeFor()` function.
- When a token transfer is performed, through `transfer`, `transferFrom` or `betaDelegatedTransfer`, the fee is computed based on the transferred amount `X` and deducted from `X`. Hence, the receiver will get `X - _fee` tokens while the `feeRecipient` gets `_fee` tokens.

As part of the system, users have to trust PAXOS with their power to adjust the `feeRate` as PAXOS could selectively set high fees. However, PAXOS has announced that it plans to make it's fees transparent to avoid any misunderstandings.

Best Practices in PAXOS's project

CHAINSECURITY is determined to deliver the best results to ensure the security of a project. To enable us to do so, we are listing Hard Requirements which must be fulfilled to allow us to start the audit. Furthermore we are providing a list of proven best practices. Following them will make audits more meaningful by allowing efforts to be focused on subtle and project-specific issues rather than the fulfilment of general guidelines.

Hard Requirements

These requirements ensure that the PAXOS's project can be audited by CHAINSECURITY.

- ✓ **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

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable.

- ✓ 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: The following build files are committed: Flattend files for `PAXGImplementation.sol` and `AdminUpgradeabilityProxy.sol`
- ✓ 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 PAXOS-GOLD. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

PAXOS has a good test suite which covers the authorized and unauthorized usage and also checks for proper event emission.

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

Security Issues

This section relates to our investigation into security issues. It is meant to highlight times when we found specific issues, but also mentions what vulnerability classes do not appear, if relevant.

Floating pragma L ✓ Fixed

PAXOS did implicitly lock the compiler version by specifying it in the truffle config. But this is not done consistently. PAXOS uses the floating pragma version `^0.4.24` in the solidity contract. Hence, there would be no warning if accidentally compiled with a different version than specified in the truffle config. Contracts should be deployed with the same compiler version they have been tested with thoroughly. Locking the pragma in the contract, helps to ensure that it does not accidentally get deployed using another compiler version that might introduce bugs that affect the contract system negatively.

PAXOS should consider to lock the pragma version[3].

Likelihood: Low
Impact: Low

Fixed: PAXOS locked down the pragma to `0.4.24`.

[3] https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-103

Trust Issues

This section mentions functionality that is not fixed inside the smart contract and hence requires additional trust in PAXOS, including in PAXOS's ability to deal with such powers appropriately.

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 PAXOS's project. These recommendations highlight possible ways for PAXOS to improve the code further.

Fee events old value argument M ✓ Fixed

The FeeRateSet, FeeRecipientSet and FeeControllerSet events are emitted from setFeeRate, setFeeRecipient and setFeeController respectively. Each of these events has two arguments, the old and new value. The setFeeController function is implemented as:

```
function setFeeController(address _newFeeController) public {
  require(msg.sender == feeController || msg.sender == owner, "only
      FeeController or Owner");
  require(_newFeeController != address(0), "cannot set fee controller to
      address zero");
  feeController = _newFeeController;
  emit FeeControllerSet(feeController, _newFeeController);
}
```

PAXGImplementation.sol

The current implementation will send the new value as both the old and new value arguments. This applies to all three events/functions.

Fixed: PAXOS fixed the issue by emitting correct event arguments.

Unused constant L ✓ Fixed

The contract PAXGImplementation defines the constant feeDecimals but makes no further use of it. PAXOS might consider removing it if no other contract or DApp uses the constant.

Fixed: PAXOS added a solidity comment describing why the unused constant exists.

Recommendations / Suggestions

- ✓ The function `transferOwnership()` only checks if the provided address is `address(0)`. Then, the function does transfer the ownership directly to the address, given as a function argument. In case of any mistake, the ownership will be transferred to some random account and is most likely "lost". Therefore, client could consider using a scheme in which the new owner needs to claim the ownership, to finally get it. This makes sure that at least the account is controlled by some user.
- ✓ When a fee is paid during a token transfer, it is added to the `feeRecipient`'s balance. Even if the `_fee` amount is zero, the balance is still updated. This could be restricted to the existing check of a positive fee. Such a change would provide gas savings in case of a zero-fee.

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 Paxos'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 Paxos and potential users.

Outdated compiler version

ChainSecurity could not find obvious issues with the compiler version Paxos is using. While the latest compiler release is version 0.5.8, Paxos uses version solc compiler, version 0.4.24. If Paxos 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[4].

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.

Rounding Errors

(Unsigned) integer divisions generally suffer from rounding errors. The same holds true for divisions inside the EVM. Therefore, the results of arithmetic operations can be imprecise. The effects of these errors can be reduced by ordering arithmetic operations in a numerically stable manner. However, even then minor errors (e.g. in the order of one token wei) can occur.

In this case, rounding errors can provide an extremely tiny benefit with regards to fee payments. However, as the transaction costs far outweigh the effect of such rounding errors, it is not classified as an issue.

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

Disclaimer