Brickblock (Phase 1) Audit Report by ConsenSys Diligence

1 - Introduction

1.1 - Review Goals

The focus of this review was to ensure the following properties:

Security:
identifying security related issues within each
contract and within the system of contracts.

Sound Architecture:
evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality:
a full review of the contract source code. The primary areas of focus include:

- Correctness (does it do what it is supposed to do)
- Readability (How easily it can be read and understood)
- Sections of code with high complexity
- Improving scalability
- Quantity and quality of test coverage

1.2 - Summary

General Overview of Contract System

The contracts under the scope of this audit were `BrickblockToken` and `CustomPOAToken`.

The set of smart contracts under audit are part of a token system (BBK + standard implementation of a Proof of Asset token) designed by Brickblock.

The contracts under inspection were:

- `BrickblockToken`
- `CustomPOAToken`

`BrickblockToken` is a standard EIP-20 compatible token in a dual-token minting system. These tokens, when staked, mint the fee-paying token: ACT, the Access Token.

`CustomPOAToken` is also an EIP-20 compatible token but has some non-typical implementations of the required method of the token standard. It single-handedly handles the property of the *asset* in question (and also the proportional payouts related to capital gains realized by that same asset).

Overview of Findings

- No Critical issues were identified in the course of the audit.
- While no issues of critical severity were found there are some of Medium severity which require your special attention. The audit team requests the careful analysis of all the issues presented and their possible solutions.
- Specification for the system under review was somewhat lacking for the `BrickblockToken` contract.

2 - General Findings

2.1 - Critical

No vulnerabilities of this severity were found.

2.2 - Major

No vulnerabilities of this severity were found.

2.3 - Medium

No vulnerabilities of this severity were found.

2.4 - Minor

More descriptive naming for methods

According to the logic implemented in some methods their names are not as accurate as they could be.

The methods in question are:

- `blacklistAddress()`
- `ethToTokens()`
- `tokensToEth()`

All present in `CustomPOAToken`.

Recommendation

Rename them to (ordered as above):

- `unwhitelistAddress()`
- `weiToTokens()`
- `tokensToWei()`

Pertinent linter output for `CustomPOAToken`

Running Solhint with a custom configuration (present in the appendix) renders pertinent issues:

```
contracts/CustomPOAToken.sol
  193:3  error  Event and function names must be different  no-simple-event-func-na
  317:3  error  Event and function names must be different  no-simple-event-func-na
  347:3  error  Event and function names must be different  no-simple-event-func-na
```

```
✖ 3 problems (3 errors, 0 warnings)
```

3 - Specific Findings

3.1 - Critical

No vulnerabilities of this severity were found.

3.2 - Major

No vulnerabilities of this severity were found.

3.3 - Medium

Bypass payment of the owner\'s fee on payout in `CustomPOAToken`

Although very costly (practically making this attack unfeasible) the *custodian* has the possibility to distribute payouts without paying any fee to the `owner`.

Given the way that the math is implemented in the `calculateFee()` method if the `msg.value` sent by the *custodian* is smaller than `1000 / feeRate`, then those 200 Wei will be distributed as a payout without incurring any fee.

Note: It's worth pointing that this attack would be *very* costly since for every 200 Wei payout you'd be losing at the very least ~1 GWei of gas.

Recommendation

Implement a check in the `payout()` method like this:

```
require(msg.value > 1000/feeRate);
```

Explicit lack of trustlessness in `CustomPOAToken`

The existing `kill()` method breaks the supposed trustlessness stemming from using a platform like Ethereum (and, frankly, the whole direction took in the rest of the smart contract

system being audited).

The fact that the method can be called at any point in time (including **during the funding phase**) and reverts the totality of the gathered funds to the `owner` makes this not only unfair for contributors but even possibly dangerous given there is now **one** private key that is the sole single point of failure of the whole crowdsale.

Recommendation

Remove the method altogether.

Lack of minimum contribution checks in `buy::CustomPOAToken`

The `ethToTokens()` method implements logic that outputs `0` if the amount of Wei sent is below a certain threshold, which, in turn, means that if a contribution of that size or lower is made no tokens will be attributed to the contributor but the Ether will be kept in the contract. (i.e.: if msg.value < fundingGoal / initialSupply then the contributor will get no tokens)

Recommendation

Implement the following check in the `buy()` method:

```
require(msg.value >= fundingGoal / initialSupply);
```

Non-optimal handling of excess Ether in contributions in `CustomPOAToken`

If a contributor sends an amount not divisible by `initialSupply / fundingGoal` ratio, a small amount of remainder Ether will be kept by the contract.

This problem is twofold: the first and more obvious part to it is that the contributor is not getting rewarded by that extra small amount in tokens; the second part is that the last contributor will take advantage of all the accumulated "dust" of previous contributors to get a larger share of tokens.

Recommendation

The first part of this issue can be solved by, instead of simply keeping the remainder of the contribution math in the contract, attributing it to the `unclaimedPayoutTotals` mapping for the user in question so that they can easily claim it at a later time.

The second part should be solved by subtracting these accumulated "dust" amounts from the

final balance in this check `if (this.balance >= fundingGoal)` in the `buy()` method.

3.4 - Minor

Non-zero check for parameters in `CustomPOAToken` constructor

Since both the `initialSupply` and `fundingGoal` variables are used in logic throughout the contract including in math where they are divisors. There should be a check in the constructor to make sure they're not equal to `0`, otherwise many of the methods will fail every time they are run effectively freezing the contract's functionality.

Recommendation

Implement the following checks in the constructor:

```
require(fundingGoal != 0);
require(initialSupply != 0);
```

`toggleDead()` method does not adhere to specification in `BrickblockToken`

The `toggleDead()` method is preceded by a comment about its functionality in the code which reads:

```
// This is only for visibility purposes to publicly indicate that we consider this
```

Since the method does not implement a one-way setter of state it does not comply with the specification.

Recommendation

Change the method body to: `dead = false;`

Appendix 1 - Audit Participants

Security audit was performed by ConsenSys Diligence.

Appendix 2 - Terminology

A.2.1 - Coverage

Measurement of the degree to which the source code is executed by the test suite.

A.2.1.1 - untested

No tests.

A.2.1.2 - low

The tests do not cover some set of non-trivial functionality.

A.2.1.3 - good

The tests cover all major functionality.

A.2.1.4 - excellent

The tests cover all code paths.

A.2.2 - Severity

Measurement of magnitude of an issue.

A.2.2.1 - minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgement as to whether addressing these issues improves the codebase.

A.2.2.2 - medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.2.2.3 - major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.2.2.4 - critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.

Appendix 3 - Mythril Output

On both contracts under scrutiny Mythril found no issues:

```
The analysis was completed successfully. No issues were detected.
```

Giving this output for both contracts after proper configuration of the analysis setup.