Smart Contract Audit Report

Button Convertible Bond Box

20th of September 2022

Contents

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

1. Preface

The developers of the **Button Convertible Bond Box** contracted byterocket to conduct a smart contract audit of their Bond Box contracts. The Button Convertible Bond Box "*is an enhancement to ButtonBonds that allows for borrowers to deposit collateral and borrow stablecoins, while having the guaranteed option to repay before maturity in order to get the collateral back.*".

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 15th of August and finished on the 19th of September 2022.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Protocol/Logic Analysis*
- *Automated Code Review*
- *Formal Report*

byterocket gained access to the code via a private GitHub repository. We based the audit on the main branch's state on August 18th, 2022 (*commit hash aaa466357414072bf35a1eb0f485b6d44aaf362b*).

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different people went through the smart contract independently and compared their results in multiple concluding discussions.

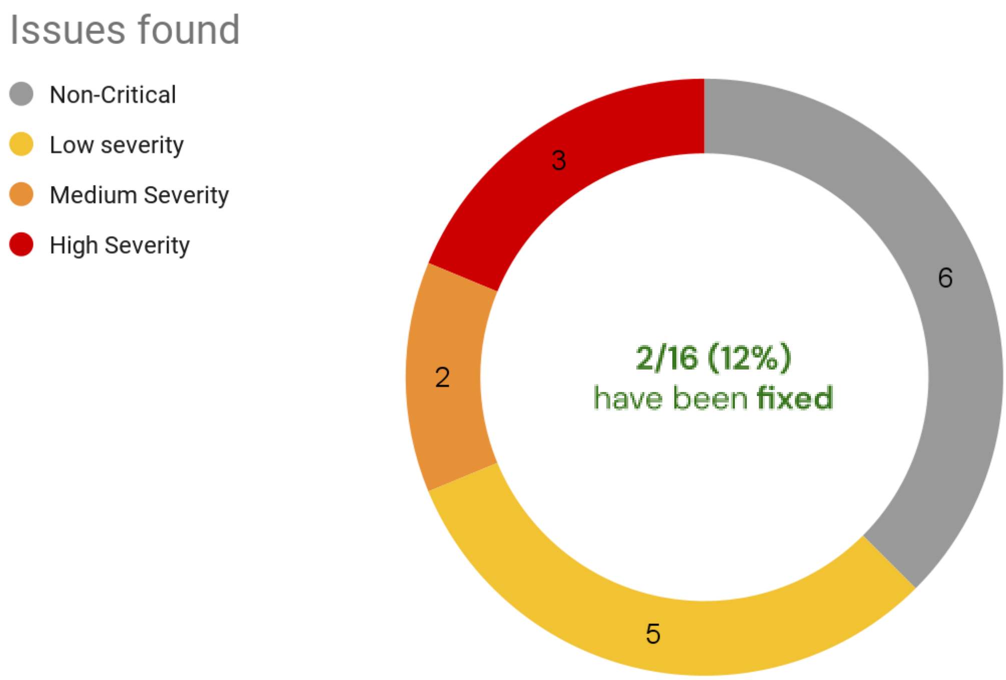
The manual review and analysis were additionally supported by multiple automated reviewing tools, like Slither, GasGauge, Manticore, and different fuzzing tools.

2.1 Severity Categories

We are categorizing our findings into four different levels of severity:

Non-Critical	Does not impose immediate risk but is relevant to security best practices. Includes issues with - Code style and clarity - Versioning - Off-chain monitoring
Low Severity	Imposes relatively small risks or could impose risks in the long-term but without assets being at risk in the current implementation. Includes issues with - State handling - Functions being incorrect as to specification - Faulty documentation or in-code comments
Medium Severity	Imposes risks on the function or availability of the protocol or imposes financial risk by leaking value from the protocol if external requirements are met.
High Severity	Imposes catastrophic risk for users and/or the protocol. Includes issues that could result in - Assets being stolen/lost/compromised - Contracts being rendered useless - Contracts being gained control of

2.2 Summary



On the code level, we **found 16 bugs or flaws, with 3 of high, 2 of medium, 5 of low severity, and 6 non-critical ones.** Out of these findings, **2 have been fixed** already. Our automated systems and review tools did **not find any additional ones.** Additionally, we found **3 gas improvements**.

The contracts are written according to the latest standard used within the Ethereum community and the Solidity community's best practices. The naming of variables is very logical and understandable, which results in the contract being easy to understand. The code is very well documented. The developers provided us with a test suite as well as proper deployment scripts.

2.3 Findings

[FIXED] ~~[HIGH SEVERITY]~~ H.1 - Faulty check in ConvertibleBondBox::initialize

Location: ConvertibleBondBox.sol - Line 46

Description:
There is a faulty check in the `initialize` function of the `ConvertibleBondBox` contract. Here, the `penalty` is being verified to be greater than the `trancheGranularity` - which is **wrong**. Instead, this check should ensure that the `penalty` is higher than the `pentaltyGranularity`.

The current tests did not catch this mistake because both constants hold the same value.

These are the affected lines of code:

```
// Revert if penalty too high
if (penalty() > s_trancheGranularity) {
  revert PenaltyTooHigh({
    given: penalty(),
    maxPenalty: s_penaltyGranularity
  });
}
```

Recommendation:
Consider replacing `s_trancheGranularity` with `s_penaltyGranularity`.

Update on the 28th of September 2022:
The developers have fixed the issue by replacing the variable accordingly.

[HIGH SEVERITY] H.2 - Withdraw from StagingBox fails after CBB reinitialized

Location: StagingBox.sol - Line 135

Description:
There is a faulty check in the `withdrawLend` function of the `ConvertibleBondBox` contract. Here, the `lendSlipAmount` is being verified to be smaller than the `reinitAmount` - which is **wrong**. Instead, this check should ensure that the `lendSlipAmount` is greater than or equal to the `reinitAmount`.

These are the affected lines of code:

```
if (_lendSlipAmount < reinitAmount) {
  revert WithdrawAmountTooHigh({
    requestAmount: _lendSlipAmount,
    maxAmount: reinitAmount
  });
}
```

Recommendation:
Consider replacing

`_lendSlipAmount < reinitAmount`

with

`_lendSlipAmount >= reinitAmount`.

[HIGH SEVERITY] H.3 - Important Assumption not documented

Location: Throughout the project

Description:
A `ButtonToken` can be instantiated with any oracle, and therefore a `ButtonToken` **does not need** to rebase to $1 but could rebase to, e.g., $2 (take a “normal” $-oracle for your token and divide the value by 2).

Throughout the project, there is the assumption that 1 unit of tokens used as `StableToken` is worth as much as 1 unit of the `ButtonToken` inside the `Tranches`.

If this assumption is not given, due to a `ButtonToken` not rebasing to $1 or a `StableToken` not using USD as a peg (*or both*), the `ConvertibleBond` does not work as intended because the internal math is broken.

Recommendation:
Consider clearly documented that the price target of the `ButtonToken` used within the `ButtonBond` has to equal the value (*or at least peg*) of the `StableToken`.

[MEDIUM SEVERITY] M.1 - Invariant stated in StagingBox not (always) fulfilled

Location: StagingBox.sol - Line 13

Description:
The contract documentation states as an invariant: “*initial price must be < $1.00*”. However, this invariant is not properly enforced in the `StagingBox` contract itself. The `StagingBox` contract only enforces that the initial price is less than **or equal** to the price granularity given during the contract’s deployment (see `StagingBox::initialize`).

The price granularity given the `StagingBox` during deployment is a constant read from a `ConvertibleBondBox` instance (see `StagingBoxFactory::createStagingBoxOnly`).

Recommendation:
Consider properly verifying that the stated invariant is enforced.

[MEDIUM SEVERITY] M.2 - Faulty return value on non-reinitialized CBB

Location: ConvertibleBondBox.sol - Line 168 - 180

Description:
If a `ConvertibleBondBox` is not yet reinitialized, the `ConvertibleBondBox` does not have an `initialPrice` defined. It is not possible to calculate a `currentPrice`, being the result of a linear function of the `initialPrice`, without having an `initialPrice`. The `currentPrice` before `ConvertibleBondBox`'s reinitialization is therefore `undefined`.

These are the affected lines of code:

```
if (block.timestamp < maturityDate) {
  price = price - ((price - s_initialPrice) *
                  (maturityDate - block.timestamp)) /
                  (maturityDate - s_startDate);
}
```

Recommendation:
Consider reverting inside the `currentPrice` function in case the `ConvertibleBondBox` is not reinitialized.

[LOW SEVERITY] L.1 - Unconventional fee mechanism

Location: ConvertibleBondBox.sol - Line 186 - 206

Description:
The `repay` function is called with a `stableAmount` as an argument. However, the amount of tokens the user actually pays is `stableAmount + fees`. While this is not incorrect in itself, it leads to a weird UX as calls such as

`cbb.repay(USDC.balanceOf(address(this)))`

become impossible.
The user needs to compute the fee amount themself and subtract that from the amount they are able to repay.
Recommendation:
Consider adapting how the fee mechanism works, potentially allowing for a better user experience.

[LOW SEVERITY] L.2 - Unintentional underflow during error handling

Location: CBBFactory.sol - Line 138 - 142

Description:
It is possible to create a `ButtonBond` with `trancheCount` of 1. If such a bond were given to the `getBondData` function through the `createConvertibleBondBox` function, the function would revert due to underflow. The underflow occurs while computing the arguments for the `TrancheIndexOutOfBonds error`.

Recommendation:
Consider either finding a different way to handle the error message or enforcing that `trancheCount` can not be 1.

[LOW SEVERITY] L.3 - Inconsistent usage of msg.sender vs. _msgSender()

Location: Throughout the project

Description:
Throughout the code of the project, both `msg.sender` and `_msgSender()` are being used. While `_msgSender()` is being used in internal contracts, `msg.sender` is used in user-facing/public contracts. This could be a design decision, but it has not been documented anywhere.

Recommendation:
Consider deciding on a unified way of obtaining the user's address.

[LOW SEVERITY] L.4 - Misleading requirement documentation

Location: IConvertibleBondBox.sol - Line 65 & 82

Description:
The functions `lend` and `borrow` state in their doc: "*Requirements: initial price of bond must be set*".

However, having the `initialPrice` set is not a requirement of the functions itself, but rather that the `ConvertibleBondBox` has been reinitialized.

Recommendation:
Consider either updating the documentation of the requirement to match the implementation or actually enforcing the documented requirement.

[LOW SEVERITY] L.5 - Missing requirement documentation

Location: IStagingBox.sol - Line 80 & 92

Description:
The functions `transmitReInit` and `transferCBBOwnership` have the requirement that the `StagingBox` itself has to be the `ConvertibleBondBox`'s owner.

This is the default setting when using the Factory to deploy the two contracts simultaneously. However, when deploying them separately, the user must manually change ownership.

Recommendation:
Consider documenting this requirement to make users aware of having to change the `ConvertibleBondBox`'s ownership.

[FIXED] ~~[NON CRITICAL]~~ NC.1 - Console logging import left in a contract

Location: ConvertibleBondBox.sol - Line 9

Description:
In the `ConvertibleBondBox` contract, there is still an import for the `forge-std` console logging feature.

Recommendation:
Consider removing the import, as it should only be used for development and should not be required for production use.

Update on the 28th of September 2022:
The developers have removed the import accordingly.

[NON CRITICAL] NC.2 - Faulty error messages

Location: Slip.sol - Line 41 & 45

Description:
In the `Slip` contract, there are two error messages in the init function that are labeled with `Tranche` instead of `Slip`.

Recommendation:
Consider updating the error messages to match the contract.

[NON CRITICAL] NC.3 - Unused constant

Location: SlipFactory.sol - Line 13

Description:
In the `SlipFactory` contract, there is a constant variable `DEFAUL_ADMIN_ROLE`, which is never used.

Recommendation:
Consider removing the variable if it is not required.

[NON CRITICAL] NC.4 - Different version pragma

Location: SBImmutableArgs.sol - Line 2

Description:
The `version pragma` used throughout the project is `^0.8.13`, but the `SBImmutableArgs` contract uses `^0.8.7`.

Recommendation:
Consider defining a unified `version pragma` that is ideally used throughout the whole project - in this case, `^0.8.13`.

[NON CRITICAL] NC.5 - Use if-else instead of two if's

Location: StagingBox.sol - Line 210 - 219 & 221 - 235

Description:
In the `transmitReInit` function, there are two `if` clauses that are covering the exact opposite of each other. This would be an ideal scenario to make use of an `if-else` clause.

These are the affected lines of code:

```
if (_isLend) {
  // [...]
}

if (!_isLend) {
  // [...]
}
```

Recommendation:
Consider making use of an `if-else` clause instead of two `if`-blocks.

[NON CRITICAL] NC.6 - Use ERC20 contract copied into project instead of importing from dependency

Location: Slip.sol - Line 210 - 219 & 221 - 235

Description:
The ERC20 dependency is imported from

`buttonwood-protocol/tranche/contracts/external/ERC20.sol`.

However, the same ERC20 contract has already been copied into the project in the `external/` directory.

Recommendation:
Consider either using the already copied `ERC20` contract or deleting the `external/` directory.

2.4 Gas Optimizations

[FIXED] [Gas Optimization] **GO.1 - Don't load constants into memory**
Location: Throughout the project

Description:
Throughout the project, constant values are loaded into memory variables that are then never reassigned.

The affected occurrences are:
- `ConvertibleBondBox::reinitialize`
- `ConvertibleBondBox::lend`
- `ConvertibleBondBox::repay`

Recommendation:
Consider using constants directly, as the current way of accessing them via a memory variable is only costing more gas.

[FIXED] [Gas Optimization] **GO.2 - Use constant instead of memory variable**
Location: ConvertibleBondBox.sol - Line 170 - 179

Recommendation:
Inside the price calculation of the `currentPrice` function, consider using the `s_priceGranularity constant` instead of a local price variable. They hold the same value, but constants are cheaper.

[Gas Optimization] **GO.3 - Make variable immutable**
Location: SlipFactory.sol - Line 15

Recommendation:
The project intentionally follows `ButtonBonds Factory` pattern. Therefore, consider making the `target` variable in the `SlipFactory immutable` and renaming it to `implementation`.

---

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The byterocket team went through the implementation and documentation of the implemented protocol.

The repository itself contained tests and documentation. We found the provided unit tests that are coming with the repository execute without any issues and cover the most important parts of the protocol.

According to our analysis, the protocol and logic are working as intended, given that any findings with a severity level are fixed. When making use of the Mainnet forking method, we were able to successfully execute the protocol.

We were **not able to discover any additional problems** in the protocol implemented in the smart contract.

4. Summary

During our code review (*which was done manually and automated*), we **found 16 bugs or flaws, with 3 of high, 2 of medium, 5 of low severity, and 6 non-critical ones.** Out of these findings, **2 have been fixed** already. Our automated systems and review tools did **not find any additional ones.** Additionally, we found 3 gas improvements.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse besides the ones that have been uncovered in our findings.

In general, there are some improvements that can be made, but we are **very happy** with the overall quality of the code and its documentation. The developers have been very responsive and were able to answer any questions that we had.