Timvi Smart Contract Audit

1. Introduction

iosiro was commissioned by Timvi to conduct a smart contract audit on the Timvi stablecoin and financial tool system. The audit was performed between 23 October 2019 and 11 November 2019. A review was performed on 21 November 2019 to verify changes made to the contracts based on the initial audit results, with the last commit at the time being de79b2e.

This report is organized into the following sections.

- **Section 2 - Executive Summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit Details:** A description of the scope and methodology of the audit.

- **Section 4 - Design Specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed Findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to better understand the risk exposure of the smart contracts, and as a guide to improve the security posture of the smart contracts by remediating issues identified. The results of this audit are only a reflection of the source code reviewed at the time of the audit and of the source code that was determined to be in-scope.

2. Executive Summary

This report presents the findings of an initial audit performed by iosiro on the Timvi smart contracts. The purpose of this audit was to achieve the following.

- Identify potential security flaws.
- Ensure that the smart contracts functioned according to the documentation provided.

Assessing the economics, game theory, or underlying business model of the platform were beyond the scope of this audit. Therefore determining whether the collateralization ratios were safe, the capability of the algorithms to stabilize the market price, or whether the ability to recapitalize other peoples' assets would be a useful game mechanic, were all beyond the scope of the audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.

- Current best practices should be followed where possible.

For the purposes of the audit, the only documentation provided by the Timvi team was the whitepaper (MD5=d1366924287f8c171c620e8d54076037).

At the conclusion of the audit one low risk issues and several informational level issues, including design comments, were open.

The low risk and informational findings included security and functional recommendations that could be used to improve the system, either by providing additional layers of security or following best practice.

At times, the code quality impacted on the auditability of the codebase, as the documentation was fairly limited with few concrete examples of expected behavior. Inline comments were also used sparingly, which impacted on the readability of the codebase. Recommendations to improve the code quality to facilitate future audits are given in the Design Comments section.

The security posture of the smart contracts could be further strengthened by:

- Remediating the issues identified in this report and performing a review to ensure that the issues were correctly addressed.
- Perform additional iterations of audits. Security best practices, tools, and knowledge change over time. Additional audits over the course of the project's lifespan ensure the longevity of the codebase.
- Continue the use of a bug bounty program to encourage the responsible disclosure of security vulnerabilities in the system.

3. Audit Details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files is considered to be out-of-scope. Out-of-scope code that interacts with in-scope code is assumed to function as intended and introduce no functional or security vulnerabilities for the purposes of this audit.

3.1.1 Timvi Smart Contracts

Project Name: Timvi

Commits: 9324706

Files: Logic.sol, TimviSettings.sol, TimviToken.sol, BondService.sol, Gate.sol, ExchangeService.sol, LeverageService.sol, PriceGetter.sol, ManagerRole.sol, TBoxToken.sol

3.2 Methodology

A variety of techniques were used while conducting the audit. These techniques are briefly described below.

3.2.1 Code Review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic Analysis

The contracts were compiled, deployed, and tested in a Ganache test environment, both manually and through the Truffle test suite provided. Manual analysis was used to confirm that the code operated at a functional level, and to verify the exploitability of any potential security issues identified. The coverage report generated by solidity-coverage of the Truffle tests included in the project is given below.

File	% Stmts	% Branch	% Funcs	% Lines
Gate.sol	0	0	0	0
Logic.sol	100	100	100	100
TimviSettings.sol	100	100	100	100
TimviToken.sol	100	100	100	100
BondService.sol	100	98.78	100	100
ExchangeService.sol	100	100	100	100
LeverageService.sol	100	100	100	100
All files	85.71	85.54	85.71	85.71

3.2.3 Automated Analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually analysed to remove false positive results. True positive results would be indicated in this report. Static analysis was conducted using Slither, Securify, as well as MythX. Tools such as the Remix IDE, compilation output, and linters were also used to identify potential areas of concern.

3.3 Risk Ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High Risk** - The issue could result in a loss of funds for the contract owner or system users.
- **Medium Risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low Risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design Specification

The following section outlines the intended functionality of the system at a high level. The specification is based on the implementation in the codebase and any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

4.1 Timvi Stablecoin

The Timvi stablecoin is a cryptocurrency that is intended to track the price of the US dollar. It should be represented by an ERC20 compliant token with the following values.

Field	Value
Name	TimviToken
Symbol	TMV
Decimals	18

It should be possible for TBoxes to control minting of TMV. Users should be able to transfer tokens, and allow other users to transfer tokens on their behalf.

4.2 Timvi TBox

A TBox is a vault where it should be possible to lock ETH and should be owned by a single address. It should be possible for owners to transfer the TBox to another user.

ERC721 Token

TBoxes should be represented by an ERC721 compliant token with the following values.

Field	Value
Name	TBoxToken
Symbol	TBX

Mint

It should be possible to mint TMV when the collateralization ratios of the system and TBox are within the system defined limits.

TBox Collateralization Ratio

The collateralization ratio of a TBox should be calculated as the amount of ETH locked in the TBox, relative to the amount of TMV already minted by the box and the current ETH price.

Global Collateralization Ratio

The global collateralization ratio should be calculated as the amount of ETH locked in the system as a whole, relative to the total supply of TMV, and the current ETH price.

Minimum Collateralization Ratio

The minimum collateralization ratio should determine the amount of TMV and ETH that can be released by a TBox. For example, if 1 ETH is locked in a TBox and no TMV has been withdrawn, at a rate of $150/ETH, and the minimum TBox collateralization ratio is 150%, then a maximum of 100 TMV can be withdrawn from the TBox.

The minimum TBox collateralization ratio should be calculated based on the global collateralization ratio.

If the system is under-collateralized (i.e. the global collateralization ratio is less than the target global collateralization ratio, between 108,7% and 215,2%), the minimum ratio is set to the global target rate.

If the system is over-collateralized, then the minimum TBox collateralization ratio should be set to the smaller value between the "ratio" (which is between 108,7% and 115,2%) and the rate which would release the number of remaining tokens until the system becomes under-collateralized.

Capitalization

It should be possible for a TBox owner or owner approved user to capitalize a box when the collateralization ratio of a TBox is within a specified range (between the minimum and maximum stability rates). Users should be incentivized to capitalize a box by receiving a bonus of USER_COMM % in ETH from their collateral.

TBox Settings

The Timvi Settings contract should be used by TBoxes. The settings should have the following values.

Name	Description	Default	Min	Max
Minimum Deposit	Minimum amount of ETH possible to hold in a TBox	0.05 ETH	0 ETH	10 ETH
System Commission	System commission when a TBox is capitalized	3%	0%	3%

Name	Description	Default	Min	Max
User Commission	User commission when capitalizing a TBox	3%	1%	6%
Global Safety Bag	Amount above ratio the global target collateralization should be	34.783%	0	100%
Minimum Stability	The minimum ratio that a box can be capitalized	106%	100%	106%
Maximum Stability	The maximum ratio that a box can be capitalized	112.913%	106,522%	112.913%
Ratio	Withdrawal rate when system is over-collateralized	115.217%	108,695%	115,217%
Global Target Collateralization	The system collateralization percentage required to provide preferable withdrawal rates	150%	108,695%	215,217%

Close

In order to close a TBox, the owner or approved account of the TBox should need to burn the number of TMV minted by the TBox. In doing so, they should receive the ETH locked in the TBox.

Close Dust

When a box has less than $3 of ETH locked in it, it should be possible for any user of the system to close the box. In this instance, the user calling the function should receive the ETH bonus normally reserved for the owner of the box, and the rest of the ETH should be claimed by the system.

Oracle Functionality

An oracle using Oraclize should maintain an ETH/USD exchange rate used throughout the system to determine the TMV/ETH rate.

4.4 Timvi Financial Tools

The Timvi System contains several financial tools that can be used with the TMV stablecoin.

Gate Service

The gate service should allow users to exchange TMV for ETH at the current USD/ETH exchange rate. In order to convert TMV, a user must first approve the Gate contract to transfer the given amount of tokens. If the Gate contract contains enough ETH to perform the exchange, the TMV should be collected to the admin TMV wallet and the user should receive the equivalent ETH, minus system fees. If the Gate does not have enough ETH, a new order should be created. Orders should be able to be filled by either an admin or other users. Admins should be able to fill multiple orders at once, while normal users should only be able to fill one order per transaction. In the event that a user fills an order, they should be charged a system fee on the TMV that they receive. In either case, the user initiating the exchange should be charged a system fee in ETH. The admin should be able to withdraw ETH and ERC20 tokens from the contract and adjust the system commission up to 10%.

Exchange Service

The Exchange Service allows an asker to exchange ETH for TMV by sending an amount above the minimum limit to the exchange contract. A taker accepts the exchange by sending enough ETH to generate a TBox that can withdraw enough TMV for the asker's exchange. The asker should be able to cancel the *ask* before an exchange happens by closing it, receiving their original ETH back. The admin should be able to withdraw system fees and adjust the system commission up to 10%.

An example of how the Exchange Service should operate is given below.

1. The asker sends 1 ETH to exchange, at a rate of $150/ETH (150 TMV).
2. The taker accepts the exchange by sending 1.5 ETH to the Exchange contract.
3. The Exchange contract creates a TBox with the taker's 1.5 ETH and withdraws 150 TMV.
4. The asker receives the 150 TMV from the contract, minus system fees.
5. The taker receives the TBox and the asker's 1 ETH, minus system fees.

Leverage Service

The Leverage Service should operate similarly to the Exchange Service, however, in this instance the order of actions is reversed. A leverage *bid* is created by the bidder who sends an amount of ETH and a withdrawal percentage (above the current minimum collateralization ratio for a new box). A taker accepts the bid by sending the

amount of ETH sent by the bidder divided by the percentage they specified when creating the bid. The admin should be able to withdraw system fees and adjust the system commission up to 10%.

An example of how the Leverage Service should operate is given below.

1. The bidder sends 1.5 ETH, at a rate of $150/ETH (225 TMV), and 150% as the withdrawal percentage to the Leverage contract.
2. The taker accepts the leverage by sending 1 ETH (1.5 ETH / 150%) to the Leverage contract.
3. The Leverage contract creates a TBox with the bidder's 1.5 ETH and withdraws 150 TMV.
4. The taker receives the 150 TMV from the contract, minus system fees.
5. The bidder receives the TBox and the asker's 1 ETH, minus system fees.

Bond Service

The Bond Service implements similar functionality to the Exchange and Leverage services, except the TBox is held by the Bond Service until the bond closes, with the initiator specifying the duration ([1..365] days) and interest rate ([0..25]%) of the bond. Again the initiator should be able to cancel the bond before it is matched. It should be possible to close the bond when either the emitter settles the TMV debt of the TBox or the bond passes its expiration date, in which case the bond owner can close the bond and claim the TBox for themselves. The admin should be able to withdraw system fees, and adjust the system emitter commission up to 10% and system owner commission up to 50%.

An example of how the Bond Service should operate when being initiated by the bond owner (similar to an exchange) is given below.

1. The bond owner sends 1 ETH to the Bond contract, at a rate of $150/ETH (150 TMV).
2. The emitter accepts the exchange by sending 1.5 ETH to the Bond contract.
3. The Bond contract creates a TBox with the emitter's 1.5 ETH and withdraws 150 TMV.
4. The bond owner receives the 150 TMV from the contract, minus system fees.
5. The emitter receives the asker's 1 ETH, minus system fees.
6. The system holds the TBox for the duration of the bond.

7. The bond can then be closed by the emitter by calling the Bond contract after approving it to transfer TMV worth the Tbox debt (150 TMV), system fees, and bond owner commission.
8. If the emitter fails to close the contract before the expiration date, the bond owner can expire the bond, in which case after the Bond Service claims system fees, based on the amount of withdrawable ETH of the TBox, the TBox is transferred to the owner.

If the emitter initiates the bond (similar to a leverage), the same process is followed as described above, except as in the leverage scenario, a withdraw percentage (above the current minimum collateralization ratio for a new box) should also be passed to the bond request by the emitter.

5. Detailed Findings

The following section details the findings of the audit.

5.1 High Risk

No high risk issues were present at the conclusion of the review.

5.2 Medium Risk

No medium risk issues were present at the conclusion of the review.

5.3 Low Risk

5.3.1 Possible for Emitter to Lose Funds through `takeBuyRequest(...)`

BondService.sol: Line 320

The `takeBuyRequest(...)` function validated whether enough ether was sent to the contract to generate the necessary TMV, but it did not validate whether too much

ether was sent. As a result, if the emitter sent more than the required amount of ether, at best the excess ether would be locked until the bond was closed, and at worst, in the event of the bond expiring, the extra ether would be transferred to the bond owner when the TBox was transferred to them.

It is recommended that `msg.value` is validated when calling `takeBuyRequest(...)` to ensure that the exact amount of ether is sent. This could be achieved by either reverting the transaction or simply returning excess ether back to the function caller.

A similar dangerous pattern was identified in ExchangeService.sol for the taker of an exchange ask in the `take(...)` function. However, due to the fact that the taker received the TBox instantly, the risk was mitigated. In the interest of defensive programming, it is recommended that validation is still used to prevent excess ether from being sent to the function.

Comment

This issue was downgraded from high risk to low risk after a control was added to the `expire(....)` function to return any ETH to the emitter above the amount required to maintain the TBox at the global target collateralization ratio in c4d4f6e and 8b3bea8. This fix still risks the emitter locking excess ETH funds for the duration of the bond, which exposes them to changes in market conditions without the ability to withdraw their ETH. Additionally, in the event that the TBox expires and the price of ETH has gone down, the emitter will still lose ETH to the owner as they forfeit ETH to increase the collateralization ratio of the TBox.

Timvi Response

We think it's more of a business model question. The issuer decides for itself how much ETH to pack as collateral. Greater collateral makes the TBox more secure and reduces the risk of recapitalization. The problem was that the issuer could lose all collateral, we thought about it with our business group and decided to limit his losses, but the question of the size of the collateral is still under consideration by the issuer.

5.4 Informational

5.4.1 Discrepancies with Whitepaper

Several differences were identified between the system described in the whitepaper and the code implementation. These are described below.

No "Insurance" ETH Claimed on Gate Exchange

The whitepaper mentioned that a 2% insurance was required by the counterparty when exchanging fees through the Gate.

> When the transaction is executed the counterparty sends a fixed sum in ETH, which is enough to carry out the deal, + 2%. 2% is a return insurance sum. It is required in the case if there is a sharp change in the ETH rate at the time of the transaction. (pg.18)

No such functionality was found in the code.

Fees are Maintained Off-chain

The whitepaper references the fact that system fees are determined based on market conditions.

> "During a sharp reduction in ETH prices, the system commission for recapitalization drops to 0." (pg. 11)

It should be noted that this logic was not implemented within the smart contracts. Rather, it was possible for a fee manager to set the rates through the TimviSettings contract.

TBond Interest Rates Determined By Users

The whitepaper referenced the fact that the interest rate of a TBond was determined based on the number of TBonds issued through a TBox.

> "The interest rate is set by the user who generates the TBond issue request. The rate can vary from 0 to 10%, depending on the number of TBonds issued through a TBox." (pg. 15)

In the smart contracts, there was only ever a one-to-one relationship between a TBox and TBond. The interest rate was determined by the creator of the bond request, and the only validation performed on the rate was to ensure that it was below 10%.

Incorrect Minimum TBox Collateral

Logic.sol: General

The whitepaper referenced that the minimum collateral percentage range for a TBox should be between 109.78% and 150%.

> The minimum possible collateral of a TBox is in the range between 150% and 109,78% and depends on the global collateral of the system. (pg. 7)

In the code implementation, this was enforced by the global target collateralization ratio, calculated as:

```
ratio() + GLOBAL_SAFETY_BAG;
```

When `GLOBAL_SAFETY_BAG` was at its default 50%, the upper boundary was in fact 150%, however, the variable could be increased by the fee manager to 100%, in which case the upper boundary became 215.21%.

Timvi Response

Updated whitepaper will be published soon.

5.4.2 Insufficient Unit Test Quality and Coverage

General

Certain contracts were found to have insufficient unit test quality and coverage. In an effort to improve maintainability, and decrease the likelihood of introducing functional or security issues into the codebase, it is highly recommended that these cases be addressed.

- *Gate.sol*: No unit tests.
- *PriceGetter.sol*: No unit tests.

Timvi Response

New tests will be published soon.

5.4.3 State Variable Changes After External Calls

Logic.sol: lines 140, 169, 175, 320, 339, 357 BondService.sol: lines 359, 361, 396, 400

Instances were found where state variables were modified after calls to external contracts. This could in theory lead to cases of re-entrancy style vulnerabilities. However, these calls were to other system contracts, which mitigated the potential risk of exploitation. Best practice would be to ensure that state variables are modified prior to external calls.

Timvi Response

Immutable contracts do not requires code maintenance. Unification of existing codebase can affect optimality and execution cost.

5.4.4 Design Comments

Actions to improve the functionality and readability of the codebase are outlined below.

Potential to Further Secure Sensitive Functionality

Generally the Timvi system managed to minimize the use of trusted roles within the system (e.g. centralized admin functionality), and included assertions to minimize the exploitation of a compromised role. Functionality was also included to renounce the settings manager role, which could control the address of the ERC20 token used by the system.

However, the fee manager and settings manager roles still had the potential to disrupt the functioning of the system in the event of a compromise. For example, the fee manager could set a malicious oracle address.

One on-chain approach to mitigate this risk would be to restrict sensitive functionality based on time. For example, the settings manager could automatically renounce its ability after a set amount of time (e.g. one month after deployment). Conversely, any update to the system, such as setting a new oracle contract could require a 48 hour period before taking effect, with the ability for the fee manager to cancel the update at any stage. In the event of a compromise, this would allow the legitimate owner to revoke any actions performed by the malicious actor.

The risk of a single rogue actor or a compromised private key can be further mitigated by ensuring that a high standard of security is observed by the sensitive account. For example, the fee and settings manager accounts should be multi-signature and signed by properly secured hardware backed accounts.

No Upgrade Mechanism

Verified smart contracts allow anyone to inspect the code to determine how the contract will behave when they interact with it. This determinism minimizes trust requirements for users, as "what you see is what you get". In the world of blockchain, smart contract upgradeability has been a controversial topic as it directly conflicts with the sentiment of decentralization as upgradeable contracts typically provide a centralized point of weakness for smart contracts.

However, early projects can often benefit from the ability to upgrade smart contracts, whether it be for functional or security improvements. As such, it is worth noting that the Timvi platform had no mechanism for upgrading its smart contracts, which improves the decentralization of the project, but might impact its ability to recover from a compromise or make functional improvements.

Timvi Response

It improves the decentralization of the project.

Events Did Not Use Indexing

The events used throughout the project did not make use of indexing. Indexing provides an efficient way for front-ends to filter event data according to relevant parameters. For example, the `BondCreated` event in the BondService contract could be modified to index the `id` and `who` parameters so that a front-end could easily filter data about a specific bond ID or determine all of the bonds created by a single address. In order to index a parameter, the `indexed` parameter should be used, e.g.

```
event BondCreated(uint256 indexed id, address indexed who, uint256 deposit, uint256 percent, uint256 expiration, uint256 yearFee);
```

Timvi Response

We use the back-end for this purpose.

Inline Code Comments Used Sparingly

Inline comments are an effective way of improving the readability of smart contracts. Without inline comments, readers need to manually determine how code is intended to operate, rather than the author explicitly defining it. As such, providing inline comments can greatly improve the maintainability and auditability of the codebase. Additionally, high level descriptions of the smart contract codebase can be tremendously helpful in helping readers familiarize themselves with how things are intended to work at a high level before attempting to decipher the particulars.

While the whole codebase would benefit from this recommendation, complex functionality in particular (e.g. the whole TBonds contract, the `collateralPercent(...)` and `withdrawPercent(...)` functions in Logic.sol, and the `convert()` function in Gate.sol) would benefit from more descriptive comments.

Code Implementation Inconsistencies

Inconsistencies were found in the implementation of certain functionality within the codebase. These are highlighted below.

In BondService.sol, the `changeEmitter(...)` and `changeOwner(...)` perform similar authentication, however, they differed in the style of the implementation. In `changeEmitter(...)` modifiers were used to ensure that only the emitter could call the function, and before it was matched. In `changeOwner(...)` a require statement was used to ensure that only the owner could call the function, and before it was matched.

In the `multiFill(...)` function in Gate.sol, fees were calculated via calls to `tmv2eth(...)` (which in turn called `chargeFee(...)`), whereas `fill(...)` calculated fees inline.

Using consistent methods of implementation improve readability, and therefore the maintenance and auditability of the codebase. When similar operations are performed using different methods, it can be unclear whether there is a functional reason for doing so, or whether it is simply an inconsistency in implementation.

Logic Optimizations

Gate.sol: line 224

The `fill(...)` function initiated an ether transfer even if there were no funds to return to the caller. A more optimal approach would be to check whether `msg.value.sub(eth)` is greater than 0 and only then perform the transfer.

5.5 Closed

5.5.1 Susceptible to Frontrunning Attacks (High Risk)

General

The Timvi system was found to have no protective mechanisms to defend against frontrunning attacks. Frontrunning entails having advance knowledge of a trade, and using that knowledge to capitalize on the trade. In the case of Ethereum, this is most often achieved by an attacker who monitors incoming transactions (e.g. by watching transactions in the mempool) for information about a future value of the system before the transaction has been mined. When a relevant transaction is detected, the attacker will send their own transaction with a very high gas price to benefit from the obtained information.

Timvi made use of a price oracle to determine the ETH/USD exchange rate. An attacker could exploit the system by monitoring for incoming price updates, and trade between ETH and TMV using either the Gate or Exchange services, which provided a method for exchanging between currencies in a single transaction, to reliably benefit from latency within the system.

Defending against frontrunning attacks is an area of active research and is a commonly faced when off-chain data is consumed on-chain. A potential defense mechanism would be to apply a modifier on sensitive functionality that validates `tx.gasprice` to ensure that it is below a threshold maintained by an oracle. If an excessive gas price is used, the transaction is deemed to be attempting to frontrun another transaction and it is rejected. The maximum gas price can be publicly disclosed to prevent legitimate users from having their transactions rejected. An example of this type of defense mechanism can be found in Synthetix's SIP12.

Comment

Fixed in 32443d3 and c6cb5b5.

5.5.2 Incorrect Variable Assignment (Low Risk)

BondService.sol: Line 284

In the `changeYearFee(...)` function, the `_oldYearFee` variable is mistakenly assigned to `bonds[_id].expiration`. It is recommended that on line 284, `_oldYearFee` be correctly assigned to `bond[_id].yearFee`.

Comment

Fixed in 2bd5a7f.

5.5.3 Unsafe Funding Method for Gate (Low Risk)

Gate.sol: Line 97

When ether was deposited through the fallback function of the Gate smart contract, it was automatically allocated to the admin's balance. This approach could result in users who are unfamiliar with the contract accidentally depositing ether in the contract through the fallback function, thinking that it is a necessary step in filling orders, only to find out that they had lost control of their funds. While it would theoretically be possible for the user to notify the admin and for the admin to withdraw the funds from the contract for the user, a more defensive approach would be to simply disallow funding the contract through the fallback function altogether (i.e. revert if the fallback function is called), and rather implement a specialized function for funding the admin account, e.g. `fundAdmin()`.

Comment

Fixed in a45675b.

5.5.4 Test Code Available in Audited Codebase (Informational)

PriceGetter.sol: lines 131 - 135

The `setPrice_TESTNET_ONLY()` function, which allowed the admin to manually set the USD/ETH exchange rate of the price oracle was found in the codebase at the time of the audit. It is recommended that it is removed before being deployed to a production environment.

Comment

Fixed in 5b25922.

5.5.5 Design Comments (Informational)

Centralized Oracle

The system used Oraclize as the price oracle for determining the ETH/USD exchange rate. Oraclize provides a means for accessing off-chain data through their API. This results in the system being reliant on a centralized oracle, as updates depend on a single node. An alternative approach would be to use Chain Link as the price oracle, which allows sourcing of data points from multiple nodes, improving the decentralization of the system.

Comment

Uses the Chainlink ETH/USD Aggregator. Implemented in 5b25922.

Low Degree of Code Reuse

The BondService, ExchangeService, and LeverageService smart contracts were found to each independently implement large potions of conceptually similar logic. This greatly increased the code size and complexity of the project.

It is recommended that rather than implementing the same logic several times, shared logic is abstracted or combined into a single contract and where necessary, service specific functionality be implemented separately.

Comment

Fixed in a89a9a6 and cab13cc.

Inappropriate Naming

The `Logic.sol` contract, implementing the Logic smart contract, was found to be poorly named. A more appropriate name describing the functionality of the contract would be "TBoxManager", as it was responsible for creating, closing, and managing collateralization ratios for TBoxes.

Comment

Fixed in e057e3b and 295f332.

Incorrect Variable Name Casing

A number of variables in TimviSettings.sol were named in all caps, e.g. `MIN_DEPO` and `SYS_COMM`. This style should typically be reserved for constant values, indicating that the values will not change. However, in TimviSettings.sol the variables using all caps

had associated setter functions. It is recommended that these variables are rather changed to caps casing, i.e. `minDepo` and `sysComm` respectively.

Comment

Fixed in 63ee24e.

Unclear Revert Behavior

Gate.sol: lines 150 & 154

The `convert()` function in Gate.sol required callers to first approve the Gate contract to transfer the necessary number of TMV tokens from their account, as the function used the ERC20 `transferFrom(...)` function. If the caller failed to first perform the approval, the contract would simply revert with the message `sub`, as the subtraction failed in `transferFrom(...)` as no tokens were available. From a user's perspective, this error message was unhelpful in determining the source of the problem.

It is recommended that before calling the `transferFrom(...)` function in `convert()`, that the approval balance for the contract is checked to verify that it has the ability to transfer the required number of tokens on the callers behalf. If not, an informative error message should be provided to the user, e.g. `Gate is not approved to transfer enough tokens`.

Comment

Fixed in e18ca9a.

Use of Single Line If Statements

Single line if statements were found throughout the project. Typically single line if statements decrease the readability of the codebase, and they could potentially lead to security or functional issues. As such, it is recommended that an explicit scope is defined using braces.

Comment

Fixed in c579b8a.

Unnecessarily Emitting Events

Gate.sol: lines 173

The `multiFill(...)` function was found to emit the `Funded` event, regardless of whether any ether was sent to the function. As the admin could either fund the contract through the fallback function or the `multiFill(...)` function, there could be instances where no value is transferred and an event is emitted, when the contract was in fact not funded. It is recommended that an if statement is used to determine whether `msg.value` is greater than 0 to determine whether funds were sent in the function call.

Comment

Fixed in 93f7dc1.

Logic Optimizations

BondService.sol: line 398

The logic to add the commission to the total ETH can be moved into the `if` statement on line 395 as adding 0 commission performs an unnecessary calculation.

Comment

Fixed in 4bdf3ac.

Spelling and Grammatical Errors

Several language mistakes were identified in the comments and revert messages in the codebase. Fixing these mistakes can help improve the end user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

- *BondService.sol: line 120* - `an buy` should be `a buy`.
- *BondService.sol: line 124* - `mathced` should be `matched`.
- *BondService.sol: line 131* - `couldn't` should be `can't`.
- *BondService.sol: line 247* - `ot` should be `or`.
- *BondService.sol: line 347* - `need` should be `needed`.
- *Gate.sol: line 70* - `couldn't` should be `can't`.
- *Gate.sol: line 162* - `your` should be `yours`.
- *Logic.sol: line 250 & 280* - `lamba` typo.

- *Logic.sol: line 335* - `It's possible to collapse only dust` should be `it's only possible to collapse dust`
- *Logic.sol: line 372* - `tokens enough` should be `enough tokens`
- *Logic.sol: line 399* - `that not` should be `that is not .`

Comment

Fixed in 31d34db, c4d4f6e, and 6349fda.

Misleading Function Names

The functions `changeOwner(...)` and `changeEmitter(...)` in BondService.sol are named in such a way that the assumption would be that the function changes the owner or emitter of the bonds. Rather, the functions were named according to whether the owner or emitter was initiating the change. For clarity's sake, it is recommended that the functions be renamed to `ownerChange(...)` and `emitterChange()` respectively, or an alternative that is unambiguous.

Comment

Fixed in d589cd1.

Incorrect Comment

Gate.sol: lines 50

The comment on line 50 of Gate.sol incorrectly refers to `EthWithdrawn` when the comment was referring to `TmvWithdrawn` . It is recommended that the comment is updated to accurately reflect the code it is describing.

Comment

Fixed in 636fa03.

Implied Visibility

BondService.sol: line 18

The visibility of `systemETH` was implicitly set as `private` by not defining any visibility. It is recommended that the visibility be explicitly set by using the `private` keyword to

improve readability.

Comment

Fixed in a43eaa4.

Unused Variable

TimviSettings.sol: line 30 The `COMM_DIVIDER` was declared and initialized, but never used in the codebase. It is recommended that it is removed.

Comment

Fixed in 010ac2f.

Unaudited Compiler Version

Version 0.5.11 of the Solidity compiler was used throughout the project. At the time of writing, no publicly available security reviews of the target version were available. A comprehensive security review was performed by the Zeppelin team on version 0.4.24 with several issues being identified and rectified in 0.4.25. Unfortunately, breaking changes were introduced from 0.4 to 0.5, so downgrading the compiler version would require a large restructuring of the codebase.

Comment

Fixed in e774276.

Outdated Interfaces

The IBond, IExchange, and ILeverage interfaces were outdated and did not accurately represent the current implementations of their respective contracts.

Comment

Fixed in eb93f3c.

Secure your system.

Request a service

START NOW →