

Security Audit

of REITBZ's Smart Contracts

May 28, 2019















Produced for

































by



Table Of Contents

Foreword	1
Executive Summary	1
Audit Overview	2
1. Scope of the Audit	2
2. Depth of the Audit	2
3. Terminology	2
Limitations	4
System Overview	5
1. Token Sale Overview	5
2. Token Overview	5
3. Extra Token Features	5
Best Practices in REITBZ's project	7
1. Hard Requirements	7
2. Soft Requirements	7
Security Issues	8
1. Frontrunning <code>removeFromWhitelist()</code>  	8
2. Missing Return Value Bug of ERC20 Tokens  	8
3. Potential DoS in <code>disburse()</code> function  	9
4. Multiple dividend payouts possible  	9
Trust Issues	10
1. Off Chain Decision on Dividend Payout Per Supported Token/ETH  	10
2. Owner May Withdraw Reserved Funds for Dividend Payout  	10
3. Only Owner Can Distribute Dividends  	10

Design Issues	12
1. Floating pragma  	12
2. Outdated Compiler Version  	12
3. Unused Function Parameters  	12
4. Owner Can Burn Tokens Only From Whitelisted Addresses  	13
5. Unused Global Array  	13
6. Add Batch Functions in TokenWhitelist  	13
7. Update to Most Recent OpenZeppelin Implementation  	13
8. Wrong Value for Variable x  	13
9. Use token.decimals() Instead of Hardcoding  	14
10. Unnecessary Ownable Initialization  	14
11. Tokens Have Decimals  	14
12. Same Error Message for Different Requires  	15
13. No Checking of Error Messages in Tests  	15
14. Owner Or Trusted User Allowed to Call _registerXXX() Functions  	16
15. Unnecessary Storage Writes  	16
Recommendations / Suggestions	17
Disclaimer	19

Foreword

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

– ChainSecurity

Executive Summary

BTG Pactual engaged CHAINSECURITY to perform a security audit of RBZ, an Ethereum-based smart contract system.

Managed by BTG Pactual, REITBZ is the first security token backed by Brazilian real estate. It allows token holders to participate in Brazil's real estate market. The RBZ token represents the right to participate in dividend payouts in a variety of currencies like stablecoins and ETH.

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 verification of generic vulnerabilities and custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure the highest security standards.

During the course of the audit, CHAINSECURITY was able to help REITBZ in addressing several security, trust and design issues of high, medium and low severity. CHAINSECURITY also recommends extending the project's documentation.

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

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on March 4, 2019. The latest update has been received on May 14, 2019.

File	SHA-256 checksum
./ReitBZ.sol	e64c4fb7c13e9eb58d63f9f0ac9d5a09e895dbdc34e9c8cb7031fce6179055e3
./ReitBZCrowdsale.sol	77f1c00bc03b50d8a6ef57378ae718a4c6aac72e8f85d949153ca6d1e160044c
./ReitBZDividend.sol	57c9ce781f74f78a09175c0b149bc7fba5a35a293f729532be58f1ddd6042ba9
./TokenWhitelist.sol	3ef93b62f0dd4fe310c7f4726d6a1141281de83e9f0b3edddfffbf7ec48eebd36
./access/TrustedRole.sol	04d09d00f3076e81850e351005eae6c564172b8704e484aafba116343a382437
./sale/ERC20MultiDividend.sol	ccf5ede07488cc9e89203727e1e8da4f2f5fbbc957739fd4a748fd9f8b368763
./sale/MultiTokenDividend.sol	0309473597244fcc8ad919e1d93088039823fedd82900089b5de94d97b19ae2f

Depth of the Audit

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

- Scanning 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 methodology¹).

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 categorize 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.

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

LIKELIHOOD	IMPACT		
	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**: during the course of the audit process, the issue has been addressed technically
-  **Addressed**: issue addressed otherwise by improving documentation or further specification
-  **Acknowledged**: issue is meant to be fixed in the future without immediate changes to the code

Findings that are labeled 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

Token Name & Symbol	REITBZ, RBZ
Decimals	18 decimals
Exchange Rate	variable
Refunds	periodic repurchases
Tokens issued	unlimited
Token Type	ERC-20
Token Generation	Mintable, Burnable
Token Benefits	Dividends possible
Pausable	Yes
KYC	Whitelist

Table 1: Facts about the RBZ token and the Token Sale.

BTG Pactual deploys the REITBZ security token with the aim to provide a seamless opportunity for investment in Brazilian real estate assets. The RBZ token is centrally controlled and only whitelisted addresses can interact with the smart contract, including receiving and transferring RBZ tokens. There might be dividend payouts on a pro-rata basis to RBZ token holders. However, this is at the sole discretion of REITBZ. Note that what is referred to as "airdrops" in the whitepaper, is called "dividends" inside the smart contracts implementation. Whitelisted users can participate in the crowdsale off-chain by contributing tokens accepted by REITBZ.

Dividends are paid out manually by REITBZ to whitelisted addresses via a dividend contract. To enable payouts, REITBZ needs to set the payment method for each whitelisted address separately. The dividends will be deposited for each account, in the individual payment method token which applies to this account. Then, the account holder can withdraw the dividends from the contract.

Token Sale Overview

REITBZ takes investment from the whitelisted investors off-chain and mints new RBZ tokens for them on-chain on their Ethereum wallet address. Investors cannot invest directly using the crowdsale contract. Only REITBZ can mint an unlimited number of RBZ tokens via crowdsale contract, no one else is allowed to mint from crowdsale contract. As REITBZ takes investment from their investors off-chain, they can choose to accept any currency which they support as an investment.

Token Overview

The REITBZ token is a centrally controlled, asset-backed ERC-20 token. The REITBZ tokens are backed by Brazilian real estate. All addresses interacting with the REITBZ token must be whitelisted first. A whitelisted RBZ token holder can only transfer the RBZ token to another whitelisted address. Each RBZ token holder is paid out dividends (aka airdrops) on a pro rate basis with the number of RBZ tokens hold. However, dividend distribution is done at the sole discretion of REITBZ.

Extra Token Features

Airdrop Proceeds of investments may either be reinvested in further assets or airdropped to RBZ holders in ETH or a stable coin. Note, that this is on the sole discretion of REITBZ and must be manually triggered.

The dividend contract may be changed by REITBZ at any time. The current dividend contract works as follows:

Every action on balances (e.g. transferring or receiving tokens) triggers `updateAccount()`. This first checks if there are dividends owed by this account. If so, the account's entry is updated with the eligible payout. To prevent claiming of past dividends, REITBZ keeps track of already paid out dividends.

To pay out dividends, REITBZ needs to do the following:

First, the total amount of tokens or ETH to be distributed needs to be transferred to the contract. Secondly, REITBZ can successfully call `addDividends()` which updates the dividend amount to be paid out for this payout method.

Finally, dividends are paid out to an address when REITBZ calls `disburse()` for this address. An investor can also call `withdraw()` to withdraw his share of dividends if present in dividend contract.

Offline contribution REITBZ accepts an offline contribution from investors and mints new RBZ tokens to their respective addresses.

Whitelist Only whitelisted addresses may interact with the token.

Burnable RBZ token holders can burn their own tokens. REITBZ can burn RBZ tokens from any user's address.

Pausable REITBZ can pause the token, which prevents RBZ token holders from transferring tokens.

Best Practices in REITBZ's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when REITBZ's project fitted the criterion when the audit started.

Hard Requirements

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

- ☐ The code is provided as a Git repository to allow reviewing of future code changes.
- ☒ Code duplication is minimal, or justified and documented.
- ☒ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with REITBZ's project. No library file is mixed with REITBZ's own files.
- ☒ The code compiles with the latest Solidity compiler version. If REITBZ uses an older version, the reasons are documented.
- ☒ There are no compiler warnings, or warnings are documented.

Soft Requirements

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

- ☒ There are migration scripts.
- ☒ There are tests.
- ☒ The tests are related to the migration scripts and a clear separation is made between the two.
- ☒ The tests are easy to run for CHAINSECURITY, using the documentation provided by REITBZ.
- ☒ The test coverage is available or can be obtained easily.
- ☐ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☒ 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.
- ☒ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☐ Functions are grouped together according either to the Solidity guidelines², or to their functionality.

²<https://solidity.readthedocs.io/en/latest/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.

Frontrunning `removeFromWhitelist()`

Upon detection of a transaction to `removeFromWhitelist()`, initiated by the Owner, affected users could try to front-run this transaction by transferring their funds to another whitelisted account. If they use a significantly higher gas price, this most likely will be successful.

Furthermore, if REITBZ wants to burn the balance of an account already removed from the whitelist, in the current implementation, this account must first be re-added to the whitelist before `burnFrom()` can be called successfully. Both burning and adding to the whitelist need to be done atomically in one transaction. Otherwise, affected users are able to interfere and could try to transfer their tokens away.

Likelihood: Medium

Impact: Low

Fixed: REITBZ changed the burn function to allow burning tokens of non-whitelisted addresses.

Acknowledged: REITBZ acknowledged that front-running is an issue. They will monitor the transactions and burn the amount from a receiver address after the sender address gets un-whitelisted.

Missing Return Value Bug of ERC20 Tokens

By calling `buyTokensWith()` in the `MultiTokenCrowdsale` contract, any whitelisted address can participate in the crowdsale by providing one of the approved ERC20 tokens.

The token transfer function calls are wrapped with a `require()` like below:

```
70 IERC20 paymentToken = IERC20(tokenAddress);
71 require(paymentToken.transferFrom(beneficiary, _wallet, tokenAmount), "Failed_
    to_receive_token_amount");
```

MultiTokenCrowdsale.sol

However, in practice there are two types of ERC20 implementations.

- Some older ERC20 tokens do not provide any return value when functions such as `transferFrom()` are called. Among these tokens, there are some popular ones such as OmiseGo³.
- Token contracts with a correct implementation of the standard return a `bool` value to let the caller know about the status of the transfer.

This has been a known issue and has caused problems⁴ with Decentralized Exchanges (DEXs) before. CHAINSECURITY recommends REITBZ to carefully check whether the `transferFrom()` call was successful.

Likelihood: High

Impact: Medium

Fixed: REITBZ fixed the problem by using OpenZeppelin version 2.2.0, which has a fix for the issue in the `SafeERC20` library.

³<https://etherscan.io/address/0xd26114cd6EE289AccF82350c8d8487fedB8A0C07#code>

⁴<https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca>

Potential DoS in `disburse()` function

Inside the loop, in the function `MultiTokenDividend.disburse()`, there is a call to the `_disburse()` function. This function initiates a transfer of either ETH or tokens to the beneficiary.

The ETH transfer is initiated by:

```
134 beneficiary.transfer(amount);
```

MultiTokenDividend.sol

If the beneficiary is a contract that does not have a **payable** fallback function, uses more than the available 2300 gas or reverts due to any other reason, the ETH transfer will revert. This would cause the entire transaction to revert. Additionally, there is no indication of which address (and therefore which beneficiary) is causing the revert. As a result, it is very hard for REITBZ to figure out which beneficiary's id should be excluded from the id list to get a successful transaction.

`address.send()` won't revert if the ETH transfer reverts. Instead, it would return the boolean `false`. If REITBZ chooses to change the implementation to use `address.send()` instead, extra attention needs to be paid to unsuccessful calls.

REITBZ is advised to review the way reverting ETH transfers are handled inside the loop.

Likelihood: Low

Impact: Low

Fixed: REITBZ is now using `address.send()` instead of `address.transfer()`. The returned boolean is now checked and processed: if the transfer failed, an event is emitted. Additionally, the amount is reset to the uncalled amount of dividends still to be payed out for this address.

Multiple dividend payouts possible

The `addDividends()` function in the `MultiTokenDividend` contract provides the functionality to add dividends for multiple tokens at once. It also needs to be done at once for all supported tokens (or at least before `disburse()` is called). Otherwise, users may trick the system to get multiple dividends paid out.

Nevertheless, this is not enforced. Inside the loop, tokens with insufficient balance are simply skipped while others, with sufficient balance, are processed normally. After dividends have been added by the owner, the owner can trigger the payout by calling `disburse()`.

In case, not all different (token / ETH) dividends are added before payout to an address, this address can simply re-register for outstanding dividends in another currency by re-investing in the crowdsale with a minimum amount of applicable tokens. This would invoke `setPaymentMethod()` which changes the user's payout token to a token where the dividends haven't been added yet, `tokenDividends[tokenAddress].totalDividendPoints` will still reflect the old value. If the owner now adds dividends for this token, a malicious user will receive dividends multiple times. This is possible because before, the dividends were payed in another currency (token).

Note that the user has to wait until the owner has called `disburse()` on their address as `setPaymentMethod()` will otherwise fail due to the `require` statement at the beginning.

Likelihood: Low

Impact: Medium

Fixed: REITBZ added a **require** statement to `addDividends()` to prevent an operational error. Also the original issue does not exist anymore because the crowdsale part has been removed in the updated code. The removed part enabled users to change their payout method on their own. Therefore, it is not possible to claim dividends twice anymore.

Acknowledged: REITBZ acknowledged that dividends are paid out in the specified payout method. This payout method token can only be set by REITBZ for a whitelisted address. When REITBZ adds the dividends, the token needs to be paused in order to prevent further issues.

Trust Issues

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

Off Chain Decision on Dividend Payout Per Supported Token/ETH

REITBZ's whitepaper states:

"The directors of the Issuer will, at all times, have sole discretion as to whether the profits generated by the Target Assets (including any capital gains and/or distributable rent) are to be partially or fully distributed to token holders through Airdrops, reinvested or used for other purposes (including, but not limited to, satisfying liabilities, expenses, running costs and other fees);

All Airdrops are contingent and conditional;"

Dividend payouts are decided off-chain by REITBZ. Investors were able to participate/buy RBZ tokens using ETH or a supported token. The dividend is paid out in either one of these tokens or ETH, depending on what is set for this account.

For each of these supported tokens and ETH, the owner needs to, first, transfer the respective balance to the MultiTokenDividend contract and secondly, call `addDividends()`. In `addDividends()`, tokens are only distributed to all eligible recipients if the balance of a token is higher than the corresponding amount of unclaimed dividends. This is done by adding them to `dividend.TotalDividendsPoints`.

These tokens are transferred to their recipients only after the owner separately calls `disburse()`.

This procedure has some implications:

- While the calculation on the amount to be distributed is done off-chain, more users may buy tokens with either ETH or a token.
- RBZ token transfers between users with different payment methods can happen at any time, further complicating the correct payout.

Hence, it is unknown how it is guaranteed that across all different tokens and ETH, a fair distribution of the total dividends is achieved.

Acknowledged: REITBZ acknowledges that operationally they should pause the token during these periods to prevent the issue. CHAINSECURITY wants to emphasize how important this is.

Owner May Withdraw Reserved Funds for Dividend Payout

The owner adds the tokens / ETH to be paid out as dividend using the `addDividends()` function. This function makes sure to only distribute the tokens / ETH if the `tokenAmount` is higher than the `dividend.unclaimDividends`. This ensures that the contract's token / ETH balance is sufficient to pay out the users.

The function `collect()`, however, enables the owner to arbitrarily withdraw all ETH or any ERC-20 token balance from the contract. If this is done, payouts of dividends will fail because of an insufficient ETH or token balance.

Acknowledged: REITBZ acknowledges this and states this is needed as contingency in case of operational mistakes.

Only Owner Can Distribute Dividends

The `disburse()` function present in the MultiTokenDividend contract transfers the tokens / ETH of the outstanding dividends payout to the given accounts. This function has the `onlyOwner()` modifier and, thus, can only be invoked by the owner.

Before dividends can be paid out, the owner needs to call the `addDividends()` function and transfer the tokens / ETH to the contract. This creates a trust issue. Once the dividends have been added, the owner can choose to actually pay out the dividends to a selected subset of addresses only. Hence, withholding others from getting their legitimate shares of the dividend.

Fixed: REITBZ added a `withdraw` functionality which allows any user to withdraw outstanding dividends on their own, once they have been distributed by REITBZ through `addDividends()`. This mitigates this trust issue. Note, that by design, REITBZ calls `disburse()` for everyone and thus, covers the transaction costs.

Design Issues

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

Floating pragma

Contracts should be deployed with the same compiler version they have been tested with thoroughly. Locking the pragma helps ensure that contracts do not accidentally get deployed using another compiler version that might introduce bugs that affect the contract system negatively.

CHAINSECURITY recommends locking the pragma version⁵.

Fixed: All contracts now have fixed pragmas to enforce Solidity compiler version 0.5.7.

Outdated Compiler Version

REITBZ's contract source files contain a floating pragma. Hence, they compile with any solc version above 0.5.0. More recent versions of solc have been released since 0.5.0, containing multiple bugfixes. Without a documented reason the newest compiler version should be used.

Fixed: All contracts now enforce Solidity version 0.5.7, the most recent release at the time REITBZ did the updates. CHAINSECURITY has no concerns about this version.

Unused Function Parameters

In the ReitBZCrowdsale contract the following functions are defined:

```
20 function _updatePurchasingState(  
21     address beneficiary,  
22     uint256 weiAmount  
23 )  
24 internal  
25 {  
26     // Set payment method as ETH  
27     _dividend.setPaymentMethod(beneficiary, address(0));  
28 }  
29  
30 function _updatePurchasingStateWith(  
31     address token,  
32     address beneficiary,  
33     uint256 tokenAmount  
34 )  
35 internal  
36 {  
37     // Set payment method as the token  
38     _dividend.setPaymentMethod(beneficiary, token);  
39 }
```

ReitBZCrowdsale.sol

Both of these functions have an unused function parameter:

- The function `_updatePurchasingState()` does not use the `weiAmount` argument.
- The function `_updatePurchasingStateWith()` does not use the `tokenAmount` argument.

⁵<https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-103>

Fixed: The functions mentioned above have been removed from the codebase because the crowdsale will be done off-chain and these functions are not needed.

Owner Can Burn Tokens Only From Whitelisted Addresses

The `burnFrom()` function has the modifier `onlyOwner`. Hence, only the owner is allowed to burn tokens by calling `burnFrom()` in the `ReitBZ` contract. Additionally, `burnFrom()` only allows to burn tokens from whitelisted addresses. Thus, the owner cannot burn tokens from addresses which are not whitelisted. E.g. if `REITBZ` detects stolen funds, they cannot be burned directly. The account needs to be whitelisted first and only then the owner can burn these tokens.

Fixed: The requirement that addresses need to be whitelisted in order to burn their tokens has been removed, which resolves this issue.

Unused Global Array

The contract `ReitBZ` defines a global `address[] public addressList;` array. Nevertheless, it is never used in the source code.

`REITBZ` should either remove this array or correct the code where this array is intended to be used.

Fixed: The unused global array was removed.

Add Batch Functions in `TokenWhitelist`

`ReitBZ` already features `addToWhitelistBatch()` and `removeFromWhitelistBatch()`. Two functions enabling batched operations on the whitelist. But this can be improved further.

The Whitelist is managed by an external `TokenWhitelist` contract. While the `ReitBZ` contract features these batch functions, the functions need to call the actual `TokenWhitelist` contract separately for every address to be added/removed. This function call incur a minimum overhead of 700 gas for every call. `CHAIN-SECURITY` recommends re-evaluating the implementation of batch functions in `TokenWhitelist`.

Fixed: Batch functions in `TokenWhitelist` have been implemented as recommended.

Update to Most Recent OpenZeppelin Implementation

`REITBZ`'s code currently uses OpenZeppelin 2.1.1. The most recent release, containing some bugfixes, is version 2.1.3.

Fixed: `REITBZ` updated to and enforces the use of OpenZeppelin 2.2.0.

Wrong Value for Variable `X`

The constant `X` is present in the contract `MultiTokenDividend` and it is defined as:

```
37 uint256 constant X = 10e18;
```

MultiTokenDividend.sol

The above would set `X` to 1 followed by 19 zeros means `10 ** 19`.

Fixed: `X` is now set as parameter of the constructor. There are no checks on the value of this parameter which is unfavorable.

Use `token.decimals()` Instead of Hardcoding

In the contract `MultiTokenDividend` the state variable `X` is assigned with a hard-coded value, defined as shown below:

```
36 // Constant to allow division by totalSupply (from the article)
37 uint256 constant X = 10e18;
```

MultiTokenDividend.sol

However, the value for `X` can be initialized in the constructor and can use `token.decimals()` according to the dividend token used.

```
39 constructor(IERC20 token) public {
40     _sharesToken = token;
41 }
```

MultiTokenDividend.sol

Fixed: This parameter is important to mitigate rounding errors while calculating dividends. Note, that setting this value to 0 breaks the contract as division by 0 will cause transactions to revert. `ReitBZDividend` inherits from the `MultiTokenDividend` contract and sets this parameter to `uint256(token.decimals()) + 10`. This resolves the issue.

Unnecessary `Ownable` Initialization

In the constructor of the contract `ReitBZ` the `Ownable()` constructor is called:

```
16 constructor() public
17 Ownable()
18 ERC20Detailed("ReitBZ", "RBZ", 18) {
19     whitelist = new TokenWhitelist();
20 }
```

ReitBZ.sol

However, the explicit `Ownable()` constructor call is not required as Solidity inheritance automatically calls the constructor of the `Ownable` contract.

Fixed: `REITBZ` removed the unnecessary `ownable` initialization.

Tokens Have Decimals

According to the specification and code the `RBZ` token has 18 decimals. However, when doing calculations on token values the decimals must be taken into account.

The `RBZ` token has 18 decimals:

```
16 constructor() public
17 Ownable()
18 ERC20Detailed("ReitBZ", "RBZ", 18) {
19     whitelist = new TokenWhitelist();
20 }
```

ReitBZ.sol

Looking at the tests in `reitbz.token.spec.js`:

```
17 contract('transfer_test', function () {
18     const senderBalance = new BN(100);
19     const receiverBalance = new BN(50);
20     const transferAmount = new BN(25);
21 }
```

```

22     before(async function() {
23         this.token = await ReitBZ.deployed();
24         await this.token.addToWhitelist(sender, { from: owner });
25         await this.token.addToWhitelist(receiver, { from: owner });
26         await this.token.mint(sender, senderBalance, { from: owner });
27         await this.token.mint(receiver, receiverBalance, { from: owner });

```

reitbz.token.spec.js

REITBZ tests with 100 RBZ, 50 RBZ and 25 RBZ tokens respectively. However, as the RBZ token has 18 decimals, these 100 RBZ tokens are in fact only 0.0000000000000001 RBZ.

While this works out in `buyTokensWith()`, when the other token also has exactly 18 decimals, this is by no means guaranteed. When other token have different decimals than 18 decimals, the calculation could give wrong results. Correcting this with an adapted conversion rate stored in `tokenRates[tokenAddress]` is very limited as the Ethereum Virtual Machine operates on integer values only.

The decimals of the RBZ token and the other Token must be included in the calculation. CHAINSECURITY strongly recommends testing this thoroughly with different cases, tokens and decimals. Currently there are only two test cases which actually execute the `buyWithToken` function successfully.

Here are some examples of Stablecoins with different decimals:

- <https://etherscan.io/token/0xdb25f211ab05b1c97d595516f45794528a807ad8>
- <https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48>
- <https://etherscan.io/token/0x056Fd409E1d7A124BD7017459dFEa2F387b6d5Cd>

Fixed: REITBZ acknowledged that the RBZ token has 18 decimals the tests were updated to reflect this.

Same Error Message for Different Requires

The function `transferFrom()` present in the contract `ReitBZ` has the same error messages for two different require calls:

```

85     require(whitelist.checkWhitelisted(msg.sender), "Sender is not whitelisted.");
86     require(whitelist.checkWhitelisted(from), "Sender is not whitelisted.");

```

ReitBZ.sol

CHAINSECURITY recommends providing unique error messages for `require` call, to easily identify the reason for the failure.

Fixed: REITBZ's require statement now contains a unique error message for each call.

No Checking of Error Messages in Tests

Inside the truffle tests, REITBZ currently does not check the thrown error messages. A test which is assumed to throw a certain error might actually throw a different error. Hence, leading to the assumption that the code is working as expected while it actually is not.

To make sure a specific error is thrown, REITBZ could consider checking the error message in their test cases. Please refer to the documentation of the `shouldFail` helper used inside the tests. The functionality is provided⁶.

Fixed: REITBZ improved the tests to ensure the expected error was thrown wherever possible.

⁶<https://github.com/OpenZeppelin/openzeppelin-test-helpers#shouldfail>

Owner Or Trusted User Allowed to Call `_registerXXX()` Functions

In the contract `MultiTokenDividend` the following functions are defined:

- `_registerBurn()`
- `_registerMint()`
- `_registerTransfer()`

These functions should only ever be called when a burn/mint/transfer event happens. owner or trusted can call these function anytime, which would interfere with the dividend tracking as the `totalSupply` of `Dividend` struct could be manipulated.

Fixed: REITBZ added an `onlyToken` token modifier to these functions, to ensure they can only be called by the RBZ token contract.

Unnecessary Storage Writes

All interactions with the REITBZ token which change the balances, are calling `updateAccount()` to update the dividend tracking. Even if `lastTotalDividendPoints` doesn't change, it is still updated. On every execution, this consumes 5000 gas.

Dividends however are paid out rarely. Thus, `lastTotalDividendPoints` rarely changes. Most of the times, these 5000 gas paid are unnecessary. Adding a simple check if the value actually changed would save significant amounts of gas in these cases while adding little overhead if the value actually needs to be updated. Overall significant amounts of gas would be saved in e.g. transfers of tokens.

Fixed: REITBZ added a check and only writes to storage if the value changed.

Recommendations / Suggestions

- ✓ The contract TokenWhitelist is defined in the Solidity file tokenwhitelist.sol. One contract refers the tokenwhitelist.sol contract as TokenWhitelist.sol. This results in compilation failure in operating systems with case sensitive filesystems. All the other file names are following InitCap styling.
- ✓ The function addDividends() present in the contract MultiTokenDividend is doing the following dividend related calculation:

```
107 if (tokenAmount > dividend.unclaimedDividends) {
108     tokenAmount = tokenAmount - dividend.unclaimedDividends;
109     dividend.totalDividendPoints = dividend.totalDividendPoints.add(
110         tokenAmount.mul(X).div(dividend.totalSupply)
111     );
112     dividend.unclaimedDividends = dividend.unclaimedDividends.add(
113         tokenAmount);
114 }
```

MultiTokenDividend.sol

The SafeMath library is used for some calculation but not used for calculating tokenAmount in the above code snippet.

- ✓ In the TokenWhitelist contract, there are the functions enableWallet() and disableWallet(). The enableWallet() function checks if _wallet is address(0). However, the disableWallet() function does not have the check. The validation checks are not consistent in these functions.

```
12 function enableWallet(address _wallet) public onlyOwner {
13     require(_wallet != address(0), "Invalid_wallet");
14     whitelist[_wallet] = true;
15     emit Whitelisted(_wallet);
16 }
17
18 function disableWallet(address _wallet) public onlyOwner {
19     whitelist[_wallet] = false;
20     emit Dewhitelisted(_wallet);
21 }
```

TokenWhitelist.sol

- ✓ In the ReitBZ contract the following functions are defined with visibility public:

```
36 function addToWhitelistBatch(address[] memory wallets) public onlyOwner {
```

ReitBZ.sol

```
46 function removeFromWhitelistBatch(address[] memory wallets) public
    onlyOwner {
```

ReitBZ.sol

Also, in the MultiTokenDividend contract:

```
89 function addDividends(address[] memory tokens) public onlyOwner {
```

MultiTokenDividend.sol

Restricting the visibility to external, allows to save gas because public functions copy array function arguments to memory⁷. This can be expensive.

⁷<https://solidity.readthedocs.io/en/latest/control-structures.html#external-function-calls>

- ☒ The `addDividends()` function is protected with the `onlyOwner()` modifier. The owner could pass an unsupported token address. If the contract has a balance of this token, then a new dividend storage array for this token address will be generated. As for this token the unclaimed dividends will be 0, `totalDividendPoints` and `unclaimedDividends` will get updated.
- ☐ `addDividends()` distributes the previously deposited funds as dividends. Note, that only the minimum amount to be distributed is controlled by REITBZ as anyone could deposit funds/token to this contract. These funds would also be distributed and possibly resulting in a discrepancy to the expected/calculated values.

Post-audit comment: All reported issues have been addressed by REITBZ. In particular, all security and design issues have been eliminated with appropriate code fixes.

Disclaimer

UPON REQUEST BY REITBZ, 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..