SMART CONTRACTS REVIEW

zokyo

October 30th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING BEYOND

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 2 Critical issues: 2 resolved = 0 points deducted
- 0 High issues: 0 points deducted
- 7 Medium issues: 4 resolved and 3 acknowledged = - 6 points deducted
- 8 Low issues: 8 resolved = 0 points deducted
- 7 Informational issue: 6 resolved and 1 acknowledged = 0 points deducted

Thus, 100 - 6 = 94

TECHNICAL SUMMARY

This document outlines the overall security of the Beyond smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Beyond smart contract/s codebase for quality, security, and correctness.

Contract Status

There was 1 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Beyond team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Beyond repository:
Repo: https://github.com/beyond-btc/cross-chain-monorepo/tree/main/packages/cross-chain-evm

Last commit - a5cfce3a51a58b986e3fa04b4a32bd1fb9a6f91e

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./WrappedERC20.sol
- ./wrappedBTC/PausableToken.sol
- ./wrappedBTC/WrappedBTC.sol
- ./access/BridgeRoles.sol
- ./libraries/ChainId.sol
- ./WrappedTokenBridge.sol
- ./OriginalTokenBridge.sol
- ./TokenBridgeBase.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Beyond smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

Beyond Protocol is a native Layer 1 (L1) interoperability protocol for Bitcoin, providing secure, cross-chain bridge infrastructure. This protocol enables users to bridge various tokens to, from, and within the Bitcoin L1 network, expanding Bitcoin's interoperability with other blockchain networks.

The audit covered several core contracts that facilitate the interoperability features, each with specific roles and functionalities. Below are the details:

1. **Core Contract Components:**
- **WrappedERC20.sol:** Manages a general-purpose ERC-20 wrapped token. This token is only mintable and burnable by the bridge, ensuring controlled creation and destruction as part of bridging operations.
- **WrappedBTC.sol:** A contract handling the wrapping and unwrapping of Bitcoin into a pegged token at a 1:1 ratio. This wrapped Bitcoin token is designed for smooth integration with ERC-20 tokens within the bridging process.
2. **Access Control and Utility Libraries:**
- **BridgeRoles.sol:** Defines role-based access control, ensuring that only authorized entities can interact with bridge functionalities, such as minting or burning tokens.
- **ChainId.sol:** Provides support for chain identification within the protocol.
3. **Bridge Contracts for Token Transfers:**
- **OriginalTokenBridge.sol:** A versatile bridge for interoperable transfers between ERC-20, Bitcoin, BRC-20, and Runes. Key capabilities for users include:
  - **ERC-20 to ERC-20 bridging**
  - **ERC-20 to BRC-20 bridging**
  - **ERC-20 to Runes bridging**
  - Bridging of native Ethereum (ETH) tokens
  - BTC wrapping and unwrapping functionality
- **WrappedTokenBridge.sol:** This bridge focuses on bridging ERC-20, BRC-20, and Runes tokens, integrating various token standards into a unified bridge mechanism.
4. **Token Bridge Base Library:**
- **TokenBridgeBase.sol:** Abstract contract which serves as the foundational layer for the bridge, implementing core bridging functionalities and shared logic that OriginalTokenBridge and WrappedTokenBridge rely upon.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Beyond team and the Beyond team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Not enough Chainlink checks can make the protocol to behave incorrectly	Critical	Resolved
2	Tokens will get locked forever after pausing the contract	Critical	Resolved
3	Ether amount sent above fee or fee + amount will get locked in the contract	Medium	Resolved
4	It is not possible to unregister tokens if they are no longer wanted to be supported	Medium	Resolved
5	Emitted events are not unique, which could lead to double spending on the receiver chain	Medium	Resolved
6	Fees can grow up to ‘infinite’	Medium	Acknowledged
7	Missing Validation for Same Token Addresses in registerToken	Medium	Acknowledged
8	Lack of Boundaries on Fee Setting	Medium	Acknowledged
9	Inconsistent and Potentially Insecure Role Transfer Mechanism	Medium	Resolved
10	Fees are set to `0` at the first instance	Low	Resolved
11	Datafeed addresses are not replaceable	Low	Resolved
12	Bridge address can not be changed if needed	Low	Resolved
13	`NonReentrant` modifier is missing	Low	Resolved
14	Amount is used as native fee in events	Low	Resolved
15	Token transfers depend on a centralized authority	Low	Resolved
16	Lack of Two-Step Ownership Transfer	Low	Resolved
17	NonReentrant Modifier Not Positioned First	Low	Resolved

#	Title	Risk	Status
18	The old Openzeppelin version is being used	Informational	Resolved
19	Unused errors	Informational	Resolved
20	Lack of zero address checks	Informational	Resolved
21	Should be immutable	Informational	Resolved
22	Unnecessary Inheritance from PausableToken in WrappedBTC Contract	Informational	Resolved
23	Potential for Conflicting Token Registrations in OriginalTokenBridge Contract	Informational	Acknowledged
24	Shadowed Variable in Fee Estimation Functions	Informational	Resolved

Not enough Chainlink checks can make the protocol to behave incorrectly

The `TokenBridge.Base.sol` smart contract implements a `getChainlinkDataFeedLatestAnswer()` function which uses Chainlink Price Feeds to retrieve asset's prices:

```
/**
    * Returns the latest answer
    * @param dataFeed Chainlink data feed
    * @return answer The latest answer
    */
   function getChainlinkDataFeedLatestAnswer(
       AggregatorV3Interface dataFeed
   ) internal view returns (int) {
       // prettier-ignore
       (
           /* uint80 roundId */,
           int answer,
           /* uint startedAt */,
           /* uint updatedAt */,
           /* uint80 answeredInRound */
       ) = dataFeed.latestRoundData();
       return answer;
   }
```

However, retrieving asset's prices using Chainlink Price Feeds requires several checks that must be implemented to ensure its correct behaviour:

1. The `latestRoundData()` call should be wrapped by a try-catch block because multisig accounts can block the access to price feeds, leading the execution to revert.
2. It is needed to check if the sequencer is down: Optimistic rollup protocols move all execution off the layer 1 (L1) Ethereum chain, complete execution on a layer 2 (L2) chain, and return the results of the L2 execution back to the L1. These protocols have a sequencer that executes and rolls up the L2 transactions by batching multiple transactions into a single transaction. If a sequencer becomes unavailable, it is impossible to access read/write APIs that consumers are using and applications on the L2 network will be down for most users without interacting directly through the L1 optimistic rollup contracts. The L2 has not stopped, but it would be unfair to continue providing service on your applications when only a few users can use them. Check the official documentation: https://docs.chain.link/data-feeds/l2-sequencer-feeds
3. The staleFeedThreshold should be unique for each data feed because their heartbeats varies.

Recommendation:

Implement the necessary mentioned checks to ensure the correct behavior of price feeds. You can consult this thread for a more in detail explanation on how to implement each fix: https://x.com/saxenism/status/1656632735291588609

CRITICAL-2 | RESOLVED

Tokens will get locked forever after pausing the contract.

The `PausableToken` smart contract is an ERC20 token defined as `Pausable`. The token implements `pause()` and `unpause()` functions to define if the contract is paused or not. The ERC20 token only allows `transfer()`, `transferFrom()`, `approve()`, `increaseAllowance()` and `decreaseAllowance()` to be executed while the contract is not paused as these functions implements a `whenNotPaused` modifier.

```
function transfer(
        address to,
        uint256 amount
    ) public override whenNotPaused returns (bool) {
        return super.transfer(to, amount);
    }
```

However, there is a mistake when implementing the `unpause()` function as it is internally, calling `_pause()` instead of `_unpause()`. This means that if the contract gets paused, there is no possibility to unpause it again, leading to all the mentioned functions not being able to be executed again.

```
/**
     * @dev called by the owner to unpause, returns to normal state
     */
    function unpause() external onlyOwner {
        _pause();
    }
```

The described issue lead to a scenario where tokens get locked forever as `transfer()` and `transferFrom()` are not executable again.

Recommendation:

Fix the error: replace `_pause()` by `_unpause()` within the `unpause()` function.

Ether amount sent above fee or fee + amount will get locked in the contract

Every function within `WrappedTokenBridge.sol` and `OriginalTokenBridge.sol` are payable functions. This is because the user executing these functions needs to pay either fees or fees + amount if ether is wanted to be bridge.
All these functions implements some safety checks to ensure that `msg.value` is not lower than the required amount (fee or fee + amount). Here are some examples:

```
if (msg.value < totalFee) {
            revert InsufficientFee();
        }
```

```
if (msg.value < totalFee + amount) {
            revert InsufficientValueSent();
        }
```

However, if a user sends an amount above the required one, it will get locked in the contract.

Consider the following example:

1. A user wants to bridge 1 ether.
2. The fee is 0.01 ether
3. The user sends 1.002 ether in total

```
if (msg.value < totalFee + amount) {
            revert InsufficientValueSent();
        }

        // Deposit ether to get wrapped ether, lock WETH in contract
        IWETH(WETH).deposit{value: amount}();

        _bridgeToErc20(
            WETH,
            remoteChainId,
            amount,
            to,
            nativeFee,
            referralCode,
            LzLib.CallParams(refundAddress, address(0)),
            adapterParams
        );
```

4. The following check is performed: `1.002 < 0.001 + 1`. The condition is met.
5. Only 1 ether is used for bridging: `IWETH(WETH).deposit{value: amount}();`
6. As a result, 0.01 ether gets locked in the contract.

This issue is present in every function used for bridging within the `WrappedTokenBridge.sol` and `OriginalTokenBridge.sol` smart contracts.

Recommendation:
For every function, the safety check should be changed. Instead of checking that the amount of ether sent is not lower than the required one, check that the amount sent is exactly equal to the required one.
In some cases it could be difficult to predict the exact amount of ether needed to execute the transaction, so another solution could be refunding the unused ether.

MEDIUM-2 | RESOLVED

It is not possible to unregister tokens if they are no longer wanted to be supported

The `registerToken()` function within the `WrappedTokenBridge.sol` smart contract is used to register a certain pair of tokens for bridging them:

```
localToRemote[localToken][remoteChainId] = remoteToken;
remoteToLocal[remoteToken][remoteChainId] = localToken;
```

These mappings are used by bridge functions in order to check if certain token is allowed to be bridged:

```
address remoteToken = localToRemote[localToken][remoteChainId];
        if (remoteToken == address(0)) {
            revert UnsupportedToken();
        }
```

However, there is no possibility for unregistering certain token or certain pair is no longer wanted to be supported. This functionality is only available when bridging to BRC20 or bridging to Runes as they implement the `setCanBeBridgedToBrc20()` and `setCanBeBridgedToRunes` but not for the rest of bridge types.

It is possible that in a certain point of time a certain token or certain pair is no longer wanted to be supported for any reason, security for example.

The same issue is present within the `OriginalTokenBridge.sol` smart contract.

Recommendation:
Implement a function that allows the owner to unregister certain tokens.

Emitted events are not unique, which could lead to double spending on the receiver chain

When a token is bridged within the `OriginalTokenBridge.sol` smart contract, the smart contract from the origin chain emits an event that is listened by the receiver chain in order to know that a bridge has taken place.

This is exactly how the `unwrapBTC` function works:

```
emit UnwrapBTC(to, amount, referralCode);
```

The event uses `to`, `amount` and referralCode` as parameters. However, it does not contain any unique identifier. This could lead to double spending if the receiver chain reads 2 times the same event by error. As a result, the double amount of Bitcoin will be unlocked.

Recommendation:
Add unique identifiers to events in order to distinguish them. A good example could a simple counter that is increased in each transaction.

Fees can grow up to 'infinite'

The `TokenBridgeBase.sol` smart contract implement some functions to define certain protocol fees:

```
/**
    * @notice Set transfer fee for BRC-20
    * @param _brc20TransferFee Transfer fee for BRC-20 in satoshi
    */
   function setBrc20TransferFee(uint256 _brc20TransferFee) external onlyOwner {
       brc20TransferFee = _brc20TransferFee;
       emit Brc20TransferFeeSet(_brc20TransferFee);
   }
```

```
   /**
    * @notice Set transfer fee for Runes
    * @param _runesTransferFee Transfer fee for Runes in satoshi
    */
   function setRunesTransferFee(uint256 _runesTransferFee) external onlyOwner {
       runesTransferFee = _runesTransferFee;
       emit RunesTransferFeeSet(_runesTransferFee);
   }

   /**
    * @notice Set transfer fee for BTC
    * @param _btcTransferFee Transfer fee for BTC in satoshi
    */
   function setBtcTransferFee(uint256 _btcTransferFee) external onlyOwner {
       btcTransferFee = _btcTransferFee;
       emit BtcTransferFeeSet(_btcTransferFee);
   }
```

However, there is no upper limit for the new fees to be set so they can be set up to any number, including really high ones.

Recommendation:
Define an upper limit for the new fees and add a check to ensure that the new value is lower or equal to the upper limit when executing the mentioned functions.

Client comment: Marking this as acknowledged, if there are no objections, as there is no intention of changing the behavior.

MEDIUM-5 | ACKNOWLEDGED

Missing Validation for Same Token Addresses in `registerToken`

In the `registerToken` function of the `OriginalTokenBridge` contract, there is no validation to ensure that `localToken` and `remoteToken` are not the same addresses. Registering the same address for both local and remote tokens could cause unexpected behavior in cross-chain operations or token handling.

Recommendation:
Add a validation check to ensure that the `localToken` and `remoteToken` addresses are not the same. This will prevent the registration of the same token address on both sides of the bridge.

Client comment: we decided to Acknowledge this finding

MEDIUM-6 | ACKNOWLEDGED

Lack of Boundaries on Fee Setting

The `setFee` function in the `OriginalTokenBridge` contract allows the owner to update the fee amount without any restrictions. There are no upper or lower bounds imposed on the fee, which could lead to excessive fees or zero fees, potentially causing issues for users or undermining the contract's financial model.

Recommendation:
Implement minimum and maximum limits for the fee to ensure it remains within a reasonable range.

Client comment: we decided to Acknowledge this finding

Inconsistent and Potentially Insecure Role Transfer Mechanism

Description

The BridgeRoles contract implements a role-based access control system with two key roles: superAdmin and btcBridge. However, the current implementation of role transfer functions presents inconsistencies and potential security risks:

1. Inconsistent Access Control:

The transferSuperAdmin function is restricted to onlySuperAdmin, while the transferBtcBridge function is restricted to onlyBtcBridge. This inconsistency could lead to confusion and potential security vulnerabilities.

2. Direct Role Transfer:

Both transferSuperAdmin and transferBtcBridge functions allow for immediate transfer of roles without any additional verification or acceptance step. This could be risky if an incorrect address is specified, potentially leading to loss of control over critical contract functions.

3. Lack of Recovery Mechanism:

There is no mechanism in place to recover control if either the superAdmin or btcBridge address is set to an invalid or inaccessible address.

4. Centralization Risk:

The current design allows for a single address to hold significant power over the contract, which could be a point of failure if that address is compromised.

Recommendation:

1. Consistent Access Control:

Consider restricting both transferSuperAdmin and transferBtcBridge functions to be callable only by the superAdmin for consistency and enhanced security.

2. Implement Two-Step Transfer Process:

Introduce a two-step transfer process for both roles where the new address must accept the role to complete the transfer. This can prevent accidental transfers to incorrect addresses.

3. Add Recovery Mechanism:

Implement a recovery mechanism or timelock for critical operations to allow for intervention in case of compromised accounts.

4. Consider Multi-Signature Approach:

For enhanced security, consider implementing a multi-signature wallet approach for the superAdmin role, requiring multiple parties to agree on critical changes.

LOW-1 | RESOLVED

Fees are set to 0 at the first instance

When the `TokenBridgeBase.sol` smart contract is deployed, the `brc20TransferFee`, `runesTransferFee` and `btcTransferFee` are not being set, therefore they are 0.

```
constructor(
        address _endpoint,
        address _btcBridge,
        address _superAdmin,
        address _btcDataFeed,
        address _nativeDataFeed,
        uint8 _nativeDecimals
    ) BridgeRoles(_superAdmin, _btcBridge) NonblockingLzApp(_endpoint) {

        btcDataFeed = AggregatorV3Interface(_btcDataFeed);
        nativeDataFeed = AggregatorV3Interface(_nativeDataFeed);
        nativeDecimals = _nativeDecimals;
    }
```

If the corresponding functions to modify these fees are not executed, then they will remain as 0.

Recommendation:
Define the values for the mentioned fees within the constructor. Do not forget to also define upper limits for these fees within the constructor.

LOW-2 | RESOLVED

Datafeed addresses are not replaceable

The `TokenBridgeBase.sol` smart contract receives 2 addresses by constructor to be used as Chainlink datafeeds: `btcDataFeed` and `nativeDataFeed`. However these addresses can not be replaced by different ones if a new datafeed is deployed by Chainlink or needs to be changed by other any reason, for example security concerns.

Recommendation:
Create 2 ‘set functions’ to modify these DataFeed addresses. Do not forget to make the functions only callable by the owner.

Bridge address can not be changed if needed

The `bridge` address used within the `WrappedERC20.sol` smart contract is set via constructor when the contract is the deployed. The address is set to an `immutable` variable and is used to ensure that certain functions are only called by the bridge.

```
constructor(
        address _bridge,
        string memory _name,
        string memory _symbol,
        uint8 _decimals
    ) ERC20(_name, _symbol) {
        if (_bridge == address(0)) {
            revert InvalidBridge();
        }

        BRIDGE = _bridge;
        TOKEN_DECIMALS = _decimals;
    }
```

Setting the bridge address to an immutable variable can lead to clear problems if a new address is needed to be used for any reason, for example security concerns.

Recommendation:

Implement a `setBridge` function which allows the owner to set a new value for the variable. Do not forget to convert WrappedBTC into an ownable contract so that the `setBridge` function is only callable by the owner.

```
function setBridge(address _bridge) external onlyOwner {
        if (_bridge == address(0)) {
            revert InvalidBridge();
        }
        bridge = _bridge;
        emit BridgeSet(_bridge);
    }
```

LOW-4 | RESOLVED

`NonReentrant` modifier is missing

Most of the functions within `OriginalTokenBridge.sol` implements a `NonReentrant` modifier to avoid reentrancy. However, some of them are not implementing the modifier, like wrapBTC, unwrapBTC, etc. Even if reentrancy is not likely to take place because of access restriction modifiers like `onlyBtcBridge`, protecting against it is recommended.

Recommendation:
Add a `NonReentrant` modifier to the functions that are not implementing it.

LOW-5 | RESOLVED

Amount is used as native fee in events

When calling `_bridgeToRunes` within `bridgeNativeToRunes`, the fifth parameter should be `nativeFee` but `amount` is passed instead.
It does not have high severity implications as it is only used in the emitted event but could lead to incorrect informational interpretation on the receiver part when reading the events in order to know what type of bridge transaction has taken place.

The same happens in `bridgeNativeToBrc20`.

Recommendation:
Fix the amount passed to be the nativeFee and not amount itself.

LOW-6 | RESOLVED

Token transfers depend on a centralized authority

The functions `transfer()`, `transferFrom()`, `approve()`, `increaseAllowance`and `decreaseAllowance` override the ERC20 default ones to add a `whenNotPaused` modifier. This modifier only allows executing the functions if the contract is not paused, which means that the transferability depends on a centralized authority. Therefore, it's advisable to use a multisig account to prevent accidental actions or to protect against the owner account being compromised.

Recommendation:
Use a multisig account for owner's account.

LOW-7 | RESOLVED

Lack of Two-Step Ownership Transfer

The Beyond smart contracts does not implement a two-step process for transferring ownerships. In its current state, ownership can be transferred in a single step, which can be risky as it could lead to accidental or malicious transfers of ownership without proper verification.

Recommendation:
Implement a two-step process for ownership transfer where the new owner must explicitly accept the ownership. It is advisable to use OpenZeppelin's Ownable2Step.

LOW-8 | RESOLVED

NonReentrant Modifier Not Positioned First

The `nonReentrant` modifier should be applied as the first modifier in any function to prevent reentrancy attacks. Placing it after other modifiers may leave the function vulnerable, as external calls or conditions evaluated by earlier modifiers can still be subject to reentrancy exploits

Recommendation:
Reorder the `nonReentrant` modifier to be the first modifier in all functions where it's applied.

INFORMATIONAL-1 | RESOLVED

The old Openzeppelin version is being used

The project is using an 'old' version of the Openzeppelin dependencies while newer ones are available. Using the newest version is always recommended.

Recommendation:
Update the Openzeppelin dependencies.

INFORMATIONAL-2 | RESOLVED

Unused errors

In the TokenBridgeBase contract, InvalidWethAddress() and AlreadyUnwrapped() errors are not used.

Recommendation:
Remove unused errors.

INFORMATIONAL-3 | RESOLVED

Lack of zero address checks

In the TokenBridgeBase contract, the constructor is missing zero address checks for _btcDataFeed and _nativeDataFeed variables.

Recommendation:
Add zero address checks for such variables.

INFORMATIONAL-4 | RESOLVED

Should be immutable

In the TokenBridgeBase contract, the nativeDecimals variable is only set in the constructor.

Recommendation:
Make the variable immutable.

Unnecessary Inheritance from PausableToken in WrappedBTC Contract

Location: WrappedBTC.sol

Description

The WrappedBTC contract inherits from PausableToken, but it does not utilize any of the pausable functionality provided by this parent contract. This inheritance introduces unnecessary complexity and potential confusion in the contract's structure and functionality.

The PausableToken contract typically provides methods to pause and unpause token transfers, which can be useful for emergency situations or during upgrades. However, the WrappedBTC contract does not implement or override any of these pause-related functions, nor does it use the whenNotPaused modifier that is usually provided by PausableToken.
This unused inheritance:

1. Increases the contract's complexity without adding value.
2. May lead to confusion for developers or auditors reviewing the code.
3. Potentially increases gas costs due to the additional inherited functions and state variables.
4. Could introduce unintended vulnerabilities if the pause functionality is accidentally exposed or activated.

Recommendation:

If the pausable functionality is not required for the WrappedBTC token, it is recommended to remove the inheritance from PausableToken. This can be done by:

1. Removing the import statement for PausableToken:

```
// Remove this line
import {PausableToken} from "./PausableToken.sol";
```

2. Changing the contract declaration to inherit directly from the necessary base contracts (likely ERC20 and Ownable):

```
// Change this line
contract WrappedBTC is ERC20, Ownable {
```

3. Adjusting the constructor to call the ERC20 constructor directly:

```
onstructor() ERC20("Beyond Wrapped BTC", "bBTC") {}
```

4. Ensuring that any other necessary functionality (like Ownable features) is properly inherited or implemented.

INFORMATIONAL-6 | ACKNOWLEDGED

Potential for Conflicting Token Registrations in OriginalTokenBridge Contract

Location: OriginalTokenBridge.sol

Description

The OriginalTokenBridge contract contains three separate token registration functions: registerToken, registerBrc20Token, and registerRunesToken. These functions are designed to register tokens for different purposes (general bridging, BRC-20, and Runes respectively). However, the current implementation does not prevent a single token from being registered for multiple purposes, which could lead to conflicts and unexpected behavior.

Specifically:

1. The registerToken function doesn't check if the token is already registered for BRC-20 or Runes.
2. The registerBrc20Token function doesn't check if the token is already registered for general bridging or Runes.
3. The registerRunesToken function doesn't check if the token is already registered for general bridging or BRC-20.

This oversight could potentially allow a single token to be registered for multiple incompatible purposes, leading to confusion and potential vulnerabilities in the bridging process.

Impact:

If a token is registered for multiple purposes, it could lead to:

1. Inconsistent behavior in the bridging process
2. Potential exploitation of the bridge by malicious actors
3. Confusion for users and integrators of the bridge
4. Difficulties in managing and tracking token registrations

Recommendation:

To address this issue, consider implementing the following changes:

1. Create a single token registration function that handles all types of registrations.
2. Implement a token registration status that clearly indicates the purpose for which a token is registered (e.g., enum TokenRegistrationType { NONE, GENERAL, BRC20, RUNES }).
3. Add checks in each registration function to ensure a token is not already registered for another purpose:

```
function registerToken(address localToken, uint16 remoteChainId,
TokenRegistrationType registrationType) external onlyOwner {

   if (localToken == address(0)) {

       revert InvalidTokenAddress();

   }

   if (tokenRegistrationStatus[localToken] != TokenRegistrationType.NONE)
{

       revert TokenAlreadyRegistered();

   }


   // Perform registration based on registrationType

   if (registrationType == TokenRegistrationType.GENERAL) {

       supportedTokens[remoteChainId][localToken] = true;

   } else if (registrationType == TokenRegistrationType.BRC20) {

       // BRC20 specific registration logic

   } else if (registrationType == TokenRegistrationType.RUNES) {

       // Runes specific registration logic

   }


   tokenRegistrationStatus[localToken] = registrationType;

   protocolTokens[localToken] = true;

   emit TokenRegistered(localToken, remoteChainId, registrationType);

}
```

4. Update other functions in the contract to respect the new registration status.

Client comment: The token registration is divided into three separate functions: `registerToken`, `registerBrc20Token`, and `registerRunesToken`, which will make the business more flexible. Because some tokens do not have corresponding brc20 or runes.

Shadowed Variable in Fee Estimation Functions

In both the `estimateBrc20TotalFee` and `estimateRunesTotalFee` functions, the variables `brc20TransferFee` and `runesTransferFee` are being used as parameters in the `getBitcoinTransferFeeInNative` function. However, these variables are also shadowed by the parameters passed into the function, leading to ambiguity and potential errors during fee calculation. This may result in the wrong fee being used, which can cause over- or under-charging in transactions.

Recommendation:
Rename the local variables within the function to avoid shadowing and clarify the intent of the calculation.

	./WrappedERC20.sol ./wrappedBTC/PausableToken.sol ./wrappedBTC/WrappedBTC.sol ./access/BridgeRoles.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./libraries/ChainId.sol ./WrappedTokenBridge.sol ./OriginalTokenBridge.sol ./TokenBridgeBase.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Beyond team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Beyond team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo