

PUBLIC

# Security Audit

## of WBTC DAO's Smart Contracts

October 11, 2018












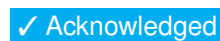
Produced for



 **Wrapped BTC**

by

 **CHAINSECURITY**

# Table Of Content

Foreword . . . . .	1
Executive Summary . . . . .	1
Audit Overview . . . . .	2
1. Scope of the Audit . . . . .	2
2. Depth of Audit . . . . .	2
3. Terminology . . . . .	2
Limitations . . . . .	4
System Overview . . . . .	5
Best Practices in WBTC DAO's project . . . . .	6
1. Hard Requirements . . . . .	6
2. Soft Requirements . . . . .	6
Security Issues . . . . .	7
1. Pausing is not complete   . . . . .	7
2. Hash collisions possible for Request   . . . . .	7
Trust Issues . . . . .	8
1. WBTC DAO's Trust Model . . . . .	8
Design Issues . . . . .	9
1. Outdated compiler version   . . . . .	9
2. Removed Custodians may violate invariants   . . . . .	9
3. Single-Custodian Design  . . . . .	9
4. Storage Leak in IndexedMapping.remove()   . . . . .	9
5. Potential Gas Savings in Request struct   . . . . .	10
6. Duplicate Events emitted   . . . . .	10

7. setCustodianBtcDepositAddress could check merchant argument   . 10

Recommendations / Suggestions . . . . . 11

Disclaimer . . . . . 12

# Foreword

We first and foremost thank WBTC DAO for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations, and results.

– ChainSecurity

# Executive Summary

CHAINSECURITY has analyzed the Wrapped-BTC contracts carefully under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts and manual review.

Minor security issues and a number of design issues have been obtained which were successfully resolved and acknowledged before deployment.

# 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 September 17, 2018<sup>1</sup> and an updated version on October 8, 2018<sup>2</sup>:

File	SHA-256 checksum
Controller.sol	86b123347fc968a25b0b07b8a562fe7b26c24cadb3dd66cb8046ef125007c6e5
ControllerInterface.sol	de2cd5cd878e13bdb95b150f40504c5e6e931d73c84f7a30a0f342960ed071df
Factory.sol	d2e62749c8d15b8648f0023b7bd0d99d90179ae21f5c1f9a331cde04d51ee6cd
Members.sol	043b3eb2443bf08f68910113c0451bd5efb3ccfdb432f436966c9b29c021daf5
MembersInterface.sol	d402db65ea486d205d50328b08b575c5d5d0ed9a7d55364074354bde3aa1abe9
WBTC.sol	ba786b7e686f915b59850449fe2535b6af03e80a218ca5a23d6306ef904009d0
IndexedMapping.sol	5fac841d40d7955781f8a26162d51d2ca0f2ea9387775a516d250b9010d6b18b
OwnableContract.sol	f2e0a2affe8c0c93f8058d188a7a35238395fafb899b679dda16f51ef706ebad
OwnableContractOwner.sol	0ac2f108d13048f0985db637f47de2693935cb00df3aabf6ab43dfe134ee6ca1

## Depth of Audit

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

- Scan 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, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology<sup>3</sup>).

**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 4 distinct categories, depending on their severities:










-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release





<sup>1</sup><https://github.com/WrappedBTC/bitcoin-token-smart-contracts/tree/144311389d5c46af5999b7cceb2bad9fec2a9a5e>



<sup>2</sup><https://github.com/WrappedBTC/bitcoin-token-smart-contracts/tree/ae1fcd4a68fab7d3931e53ff96b53444dfd222c1>

<sup>3</sup>[https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology)

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as  **No Issue**. If during the course of the audit process, an issue has been addressed technically, we label it as  **Fixed**, while if it has been addressed otherwise by improving documentation or further specification, we label it as  **Addressed**. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as  **Acknowledged**.

Findings that are labelled 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 what kind of issues were found during the audit.

# Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover 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 lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine 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

The WBTC is an ERC20 token, that represents Bitcoin as an (extended) ERC20 token on the Ethereum blockchain e.g. 1 BTC = 1 WBTC token. The involved entities are at least one Custodian (the current setup is tailored to exactly one) and multiple Merchants. The whole system has in general two main tasks:

1. **Minting WBTC:** If a matching amount of BTC is locked at a custodian's account (on BTC blockchain), the corresponding amount of WBTC tokens is minted (released) to the merchant (on the ETH blockchain).
2. **Burning WBTC:** When a merchant wants to convert his WBTC tokens back into BTC, he place a burn request (the specified amount of WBTC tokens are burned). If successful the custodian sends the merchant the requested amount of BTC (on bitcoin blockchain).

To accomplish a Bitcoin to WBTC swap and back, a merchant (e.g. Kyber, who wants to trade the WBTC) sends BTC to a custodian. The custodian confirms that this merchant has deposited a certain amount of BTC on bitcoin blockchain. A matching amount of WBTC is then minted by a custodian and can be used by the merchant. Accordingly, if a merchant wants to swap back the WBTC to BTC, the merchant files a request to burn the WBTC. The custodian transfers the BTC back to the merchant, if the burning of the WBTC was successful.

Overall, the smart contracts request and record the transaction details on Ethereum blockchain. Actual transaction of BTC are happening on bitcoin blockchain. Other tasks include managing (adding/removing) merchants and custodians.



# Best Practices in WBTC DAO'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 WBTC DAO's project fitted the criterion when the audit started.

## Hard Requirements

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

- The code is provided as a Git repository to allow the review 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 WBTC DAO's project. No library file is mixed with WBTC DAO's own files.
- The code compiles with the latest Solidity compiler version. If WBTC DAO 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 WBTC DAO.

- 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 WBTC DAO.
- 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 not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- There is no dead code.
- The code is well documented.
- The high-level specification is thorough and allow a quick understanding of the project without looking 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.
- Function are grouped together according either to the Solidity guidelines<sup>4</sup>, or to their functionality.

<sup>4</sup><https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions>

# Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

## Pausing is not complete

The specification states that the DAO can create a state where “all token transfers are paused”. Therefore, the pause and unpause functions exist. However, in the current implementation the pausing does not restrict the mint and burn functions, which can be used to perform token transfers. Hence, the state is not fully frozen and during an token hard fork, balances might be lost.

**Likelihood:** Low

**Impact:** High

**Fixed:** The problem is solved by WBTC DAO in function `mint` and `burn` of `Controller`. When contract is paused, DAO cannot mint or burn tokens.

## Hash collisions possible for Request

The function `calcRequestHash` and the design of the struct `Request` allow hash collisions to easily occur:

```
function calcRequestHash(Request request) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked(
        request.requester,
        request.amount,
        request.btcDepositAddress,
        request.btcTxid,
        request.nonce,
        request.timestamp
    ));
}
```

Factory.sol

As the subsequent fields `btcDepositAddress` and `btcTxid` are both declared as `string` and hence of variable length, they can be manipulated to allow a hash collision due to the functionality of `encodePacked`. In particular, the packing of “ab”, “c” is equivalent with the packing of “a”, “bc”. A complete example can be seen provided<sup>5</sup>.

According to our analysis, this collision can only be exploited in limited ways. The possibilities include:

- *Triggering the confirmation of an invalid mint request:* A mint request can be overwritten with a second one that has an identical hash using the `addMintRequest` function. If this happens just before `confirmMintRequest` is called, the custodian can be tricked into confirming a non-sense request. However, this would mostly affect auditing.
- *Triggering the rejection of a valid mint request:* A custodian could be sent a false mint request, which it would reject. Right before the rejection, the false mint request could be replaced with a correct mint request and hence, at first glance, it would look as if the custodian would be misbehaving.

**Likelihood:** Low

**Impact:** Medium

**Fixed:** WBTC DAO solved the hash collision problem in function `calcRequestHash` by using function `abi.encode` to encode the input data that is used to compute the hash.

<sup>5</sup><https://gist.github.com/ritzdorf/8fe8f79227000af01e176bbefac0a828>

# Trust Issues

Below CHAINSECURITY discusses the trust model of WBTC DAO. Apart from the explicit trust assumptions that were listed in the white paper, CHAINSECURITY identified no further trust assumptions for the system.

## WBTC DAO's Trust Model

As clarified in the white paper: "Custodians and Merchants are bound to a legal agreement and also curated and audited by the DAO governance system." As CHAINSECURITY's audit only focusses on the smart contracts, the following trust-related issues from the white paper are relevant:

### Strict trust

- The WBTC DAO (as a collective) is fully trusted.

CHAINSECURITY's *evaluation*: This property is absolutely critical. The WBTC DAO has absolute power. It can change the protocols for minting/burning and for custodian-merchant management. The WBTC DAO can also mint tokens (directly or indirectly). In the current implementation trustworthiness is provided by using a multi-signature wallet (which is not in scope of this audit) controlled by multiple independent entities.

- A custodian is being trusted to confirm valid mint requests, i.e, a request followed (or preceded) by a BTC deposit to custodian deposit address with greater or equal amount.

CHAINSECURITY's *evaluation*: This property is essential as it is technically difficult to link Bitcoin and Ethereum. This property represents this link and also highlights the custodian's power, who can theoretically just accept the BTC deposits without return any WBTCs.

From the context of the white paper, it is clear that the custodian is trusted to *only* confirm valid mint requests. This is essential as the stability of the WBTC depends on this. Otherwise, a malicious custodian or a custodian-merchant collusion could mint illegitimate WBTCs and hence jeopardize the stability of the system.

- A custodian is being trusted to confirm burn requests and send BTC to merchant deposit address.

CHAINSECURITY's *evaluation*: This point is closely related to the property above and is a prerequisite for the stability of WBTC. If WBTCs cannot be redeemed correctly, which requires this property, their value is likely not to hold up. This property provides the link from Ethereum to Bitcoin. It once again highlights the potential power of the custodian but also his interest to behave correctly.

### Soft trust

- Merchant is trusted not to spam the custodian with dust requests.

CHAINSECURITY's *evaluation*: Spamming would be more of an inconvenience as requests can be confirmed in any order. Furthermore, as mentioned in the white paper, it could be easily detected through monitoring.

- Custodian is trusted not to frequently change his BTC deposit addresses.

CHAINSECURITY's *evaluation*: If the custodian changes its BTC deposit address, this can lead to confusion and possibly lost funds as BTC deposits could be made to an outdated address. However, in the white paper, multiple ways have been proposed to address this possible confusion. Furthermore, there is no apparent reason for the custodian to change its BTC deposit address which makes this even less of an issue.

# Design Issues

The points listed here are general recommendations about the design and style of WBTC DAO's project. They highlight possible ways for WBTC DAO to further improve the code.

## Outdated compiler version

Solidity version 0.4.25 was released on September 13th. The new version has two important fixes<sup>6</sup> and the latest compiler version should be used, unless there is an explicit and documented reason for using an older compiler version.

**Acknowledged:** WBTC DAO will be using 0.4.24 solidity version as the issues fixed in version 0.4.25 are not relevant for their code.

## Removed Custodians may violate invariants

As stated in the documentation, the DAO can remove custodians. However, if care is not taken, this can lead to a number of issues including a violation of invariants.

- If a custodian with BTC in its custodian wallet is removed, the assertion that *the amount of BTC in custodian wallet(s) is greater or equal to the total supply of WBTC* will likely not hold up any longer, as these BTC fall out of the equation. Therefore, special care has to be taken in this case.
- Issues arise in case the removed custodians had pending mint requests. The documentation talks about prevention mechanisms for erroneous mint requests. Similar schemes can be applied here. Additionally, “other” custodians could confirm the mint request in order to compensate the merchant or the `cancelMintRequest` function can be used if the BTC transaction has not been performed yet.
- In case there are pending burn requests, the effect is limited. The “other” custodians can fulfill this part.

**Addressed:** The design was changed with regards to the handling of custodians (see separate subsection). Due to that change and the handling of the functionality by WBTC DAO which will that status modifications do not violate the invariant, this issue is addressed.

## Single-Custodian Design

As stated in the documentation, these contracts are designed for a “single custodian and multiple merchants”. They cannot be repurposed for multiple custodians. This is because custodian-related authorization is role-based (by using `onlyCustodian`) and not address-based (by using `msg.sender`). Therefore, any custodian can act as any other custodian.

Hence, even though naming, such as `addCustodian` and `getCustodians` might suggest that the contracts support multiple custodians, it should never hold custodians that are controlled by different entities.

**Fixed:** WBTC DAO modified the design to support a single-custodian design in `Members`.

## Storage Leak in `IndexedMapping.remove()`

When an element is removed from the `IndexedMapping`, the mapping `valueIndex` is not updated accordingly. The `remove()` method is deleting an entry from `valueExists`, but is not deleting an entry from `valueIndex`. Therefore, `valueIndex` is constantly growing even though elements might be removed. Besides consistency, this would lead to a lower gas consumption because gas refunds are obtained once a storage cell is cleared.

**Fixed:** WBTC DAO solved the problem in `IndexedMapping`, and storage is correctly cleared when deleting elements.

<sup>6</sup><https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/>

## Potential Gas Savings in Request struct

The Request struct (a central data structure in the project) is designed as follows:

```
struct Request {
    address requester; // sender of the request.
    uint amount; // amount of wBTC to mint/burn.
    string btcDepositAddress; // custodian's BTC address in mint, ...
    string btcTxid; // bitcoin txid for sending/redeeming BTC in ...
    uint nonce; // serial number allocated for each request.
    uint timestamp; // time of the request creation.
    RequestStatus status; // status of the request.
}
```

Factory.sol

The size of the struct could potentially be reduced. If `btcDepositAddress` and `btcTxid` would be stored in binary format they could be stored inside `bytes32`. However, this would also lower usability.

Alternatively, if the `timestamp` would be reduced to `uint128` (which is easily sufficient for its purpose), it could share a storage cell with the `status`. Further size reduction of the `timestamp` and reordering of the struct could also pack `timestamp`, `status`, and `nonce` in one storage cell, as `uint128` seems sufficient for a `nonce`.

**Acknowledged:** WBTC DAO is not going to update to the `struct` as the number of dependant transactions is going to be limited and as these transactions will be between custodians and merchants only. Also converting `string` to `bytes32` would require additional handling on the DApp side. Having both `btcDepositAddress` and `btcTxid` as `string` allows anyone to see BTC addresses and BTC txids on block explorers.

## Duplicate Events emitted

The Paused and Unpaused events duplicate the already existing Pause and Unpause events inside the OpenZeppelin contracts and could therefore be potentially omitted. While Paused and Unpaused are emitted by the Controller, Pause and Unpause are emitted by the token contract, but it is not clear that both are needed.

Furthermore, Mint and Burn events are duplicated with custom events. However, these contain extra information about the BTC transactions and hence add additional value.

**Acknowledged:** WBTC DAO is aware of this.

## setCustodianBtcDepositAddress could check merchant argument

The `setCustodianBtcDepositAddress` function inside the Factory contract does not validate the merchant argument, i.e. it does not check that the argument represents a valid merchant address. This could be added to avoid mistakes and inconsistencies.

**Acknowledged:** Even if the wrong merchant address is set by custodian, merchant will not be able to create mint requests.

# Recommendations / Suggestions

- The functions `compareStrings()` and `isEmptyString()`, `getTimestamp()` from the `Factory` can be made **internal**. There is no need for these functions to be `public`.
- The import of `HasNoEther.sol` can be removed from `OwnableContract` as it is not used.
- The Project uses `OpenZeppelin v1.12.0` (which is the latest stable release at the time of writing), but in the pre-release version `OpenZeppelin 2.0 RC 1`, ownership-related smart contracts have been removed<sup>7</sup> which are being used in the project. These contracts include `Claimable.sol` and `CanReclaimToken.sol`. We also recommend to use `OpenZeppelin v1.12.0`
- Given that the `renounceOwnership` function doesn't seem to be of use it can be overridden with an empty function for two reasons. First, it prevents accidental usage (which would have severe consequences) and second it reduces the code size slightly.
- The events of the contracts use an inconsistent scheme when it comes to indexing. As an example, some events (like `CustodianBtcDepositAddressSet`) index the merchant address, while others (like `MerchantAdd` and `MerchantRemove`) do not index the address. This could be unified according to the needs of the whole system.
- There seems to be a typo in:

```
event MerchantRemove(address custodian);  
kyberbtc.sol
```

- To further enhance the security of `IndexedMapping` additional assertions could be added. These could for example ensure that `self.valueList.length > 0` before `self.valueList.length--`. However, we see this as optional as it always seems to hold in the current implementation.
- Depending on the expected use cases the following functions could be added so that merchants and custodians can quickly identify relevant requests.
  - `mintRequestsFilterBy(status)` returns `(uint[] nonceList)`
  - `mintRequestsFilterBy(requester, status)` returns `(uint[] nonceList)`
  - `burnRequestsFilterBy(status)` returns `(uint[] nonceList)`
  - `burnRequestsFilterBy(requester, status)` returns `(uint[] nonceList)`
- The `RequestStatus` is defined as follows:

```
enum RequestStatus {PENDING, CANCELED, APPROVED, REJECTED}  
Factory.sol
```

Hence, the default value is `PENDING` and `getStatusString` returns `pending` for an empty status. This might not be intended.

- The documentation mentions several ways of addressing erroneous mint requests. Besides the already proposed techniques, a timelock could be used after which the BTCs are automatically returned to the sender if they have not been claimed by the custodian.

**Post-audit comment:** WBTC DAO has fixed some of the issues above and has is aware of all the implications of those points which were not addressed. Given this awareness, WBTC DAO has to perform no more code changes with regards to these recommendations.

<sup>7</sup><https://github.com/OpenZeppelin/openzeppelin-solidity/pull/1254>

## Disclaimer

UPON REQUEST BY WBTC DAO, 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..