Coinspect Smart Contract Source Code Audit

Prepared for Bitgo • November 2018

v1113

1. Table of Contents

2. Executive Summary

In November 2018, Bitgo engaged Coinspect to perform a 16-hours security audit of the Wrapped BTC Token (WBTC) smart contracts.

The objective of the audit was to evaluate the security of the smart contracts. During the assessment, Coinspect identified the following issues:

High Risk	Medium Risk	Low Risk	Zero Risk
-	-	-	2

The smart contracts were found to be very solid from a security standpoint, and also well designed and elegant.

The code compiled without any warnings from the compiler. The tests were reviewed and verified to pass and have full coverage.

3. Introduction

WBTC is an ERC20 compliant token, with additional burning and minting functionality. The WBTC ecosystem consists of *custodians*, *merchants* and *end users*. WBTC tokens represent BTC in a 1 to 1 ratio, and they are fully backed: for each WBTC minted there is a BTC in custody. Essentially, WBTC tokens are minted when merchants deposit BTC in a custodian vault, and a merchant can burn WBTC tokens and in return receive BTC from the custodian.

The smart contracts keep track of all the minting and burning, as well as of the registered merchants and custodians and their associated bitcoin addresses.

The audit comprised the following Solidity files (shown with their respective SHA-256 hash):

```
1cb2333ba7589af0731b50589a691930343afa45ff23d0cd61c3e6317bd6c33b  Migrations.sol
de2cd5cd878e13bdb95b150f40504c5e6e931d73c84f7a30a0f342960ed071df  controller/ControllerInterface.sol
86b123347fc968a25b0b07b8a562fe7b26c24cadb3dd66cb8046ef125007c6e5  controller/Controller.sol
d2e62749c8d15b8648f0023b7bd0d99d90179ae21f5c1f9a331cde04d51ee6cd  factory/Factory.sol
d402db65ea486d205d50328b08b575c5d5d0ed9a7d55364074354bde3aa1abe9  factory/MembersInterface.sol
043b3eb2443bf08f68910113c0451bd5efb3ccfdb432f436966c9b29c021daf5  factory/Members.sol
9f9d5ad791482060ac531e6c5cfa7ed4e8d64aed91663798bf199c53c7e0e600  mock/BasicTokenMock.sol
096f2388195d42a7b0c1d3693610c3a8430f157f6d53146ac631880e6385f402  mock/IndexedMappingWrapper.sol
ba786b7e686f915b59850449fe2535b6af03e80a218ca5a23d6306ef904009d0  token/WBTC.sol
5fac841d40d7955781f8a26162d51d2ca0f2ea9387775a516d250b9010d6b18b  utils/IndexedMapping.sol
0ac2f108d13048f0985db637f47de2693935cb00df3aabf6ab43dfe134ee6ca1  utils/OwnableContractOwner.sol
f2e0a2affe8c0c93f8058d188a7a35238395fafb899b679dda16f51ef706ebad  utils/OwnableContract.sol
```

4. Summary Of Findings

ID	Description	Risk
WBTC-1	Merchant not validated in function `setCustodianBtcDepositAddress`	Zero
WBTC-2	Storage leak in deposit addresses	Zero

5. Findings

WBTC-1	Merchant not validated in function `setCustodianBtcDepositAddress`

Total Risk	Impact	Location
Zero	Low	Factory.sol
Fixed	**Likelihood**	**Status**
√	Low	Fixed

Description

The custodian calls function `setCustodianBtcDepositAddress` to set a deposit address for a given merchant address:

```
function setCustodianBtcDepositAddress(
	address merchant,
	string btcDepositAddress
)
	external
	onlyCustodian
	returns (bool)
{
	require(merchant != 0, "invalid merchant address");
	require(!isEmptyString(btcDepositAddress), "invalid btc deposit
address");

	custodianBtcDepositAddress[merchant] = btcDepositAddress;
	emit CustodianBtcDepositAddressSet(merchant, msg.sender,
btcDepositAddress);
	return true;
}
```

The argument `merchant` is not verified to be a registered merchant address.

Recommendations

This is not necessarily a problem, since the function `setCustodianBtcDepositAddress` can only be called by the custodian, and no other functionality will be available to the “merchant” unless it is a registered merchant. Nevertheless, consider verifying that the argument `merchant` is a registered merchant address by calling `controller.isMerchant(merchant)`.

WBTC-2 Storage leak in deposit addresses

Total Risk	Impact	Location
Zero	Low	Factory.sol
Fixed?	**Likelihood**	**Status**
✖	Low	Not Fixed

Description

Each merchant has two associated bitcoin addresses for sending and receiving BTC (one owned by the custodian, one owned by the merchant). These two addresses are stored in the `Factory` contract, in the mappings `merchantBtcDepositAddress` and `custodianBtcDepositAddress`.

Merchants can be removed by calling the function `removeMerchant` in the `Members` contract. When a merchant is removed in this way, they are not automatically removed from the two aforementioned mappings in the `Factory` contract.

Keeping the entries in the mappings will not incur in further gas use, while explicitly deleting them would have a cost. However, deleting the entries from the mappings gives a gas refund (see references below.) Deleting the mapping entries in the *same transaction* that calls `removeMerchant` would make the transaction cheaper.

References

See the Yellow Paper, section 9.2 and appendices I and H.

Recommendations

Consider deleting the entries from the mappings in the `Factory` contract in the same transaction that calls `removeMerchant` in the `Members` contract. For example, this could be implemented by adding to the `Factory` contract a function `cleanUpMerchant` that deletes the mapping entries for a merchant, and adding to the `Controller` contract a function `removeAndCleanUpMerchant` that calls both `removeMerchant` in the `Members` contract and `cleanUpMerchant` in the `Factory` contract.

8. Disclaimer

The present security audit is limited to smart contract code. It does not cover the technologies and designs related to these smart contracts, nor the frameworks and wallets that communicate with the contracts, nor the general operational security of the company whose contracts have been audited. This document should not be read as investment advice or an offering of tokens.