BOB

FusionLock

Smart Contract Audit

Mar 29, 2024

Common Prefix

Overview

Introduction

Common Prefix was commissioned to perform a security audit on BOB's `FusionLock.sol` contract at commit hash [afe34d57ff6ad61cd9593755b36c5250e53159f5](afe34d57ff6ad61cd9593755b36c5250e53159f5).

The contract was accompanied by detailed documentation, and an extensive test suite was provided.

Protocol Description

The `FusionLock` contract enables users to lock their tokens (ERC20 and native tokens) within the contract and then bridge them to another chain.

This contract is intended for deployment on an L1 chain (Ethereum mainnet). After a certain time, set by the contract owner, users can either bridge their tokens to L2 (BOB L2, based on the Optimism stack) or reclaim them. The owner has control over various parameters, including the list of tokens allowed for deposit, the withdrawal time (which can only be decreased), the address of the bridge, and the corresponding L2 address for each L1 token address. Additionally, the owner can pause deposits and bridging to L2. Hence, a level of trust is placed in the owner to configure these parameters accurately. Regardless, after the withdrawal time elapses, the users can always withdraw and reclaim their tokens.

We were also provided with a list of tokens, and we verified their compatibility with the Optimism bridge.

Disclaimer

Note that this audit does not give any warranties on the bug-free status of the given smart contracts, i.e. the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. Functional correctness should not rely on human inspection but be verified through thorough testing. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	Description
Critical	Logical errors or implementation bugs that are easily exploited and maylead to any kind of loss of funds
High	Logical errors or implementation bugs that are likely to be exploited and may have disadvantageous economic impact or contract failure
Medium	Issues that may break the intended contract logic or lead to DoS attacks
Low	Issues harder to exploit (exploitable with low probability), issues that lead to poor contract performance, clumsy logic or seriously error-prone implementation
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable and easier to maintain

Findings

Critical

No critical issues found.

High

No high issues found.

Medium

No medium issues found.

Low

LOW-1	Compatibility of the tokens with the Optimism bridge
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

As part of this audit, we evaluated the compatibility of the list of ERC20 tokens provided to us by the team with the Optimism bridge. Most of the tokens are expected to function seamlessly with the bridge.

However, we have some concerns regarding the Aave aTokens (aUSDT, aUSDC, aDAI, awBTC). These tokens have a balance that expands, accounting for accrued interest. While we do not anticipate any technical issues with moving these tokens from L1 to L2 or vice versa through the bridge, it's important to note that as long as the aTokens remain locked in the L1-side bridge contract, they will continue to accrue interest. Consequently, when a user attempts to withdraw their initially deposited amount, they will be able to do so, but the accrued interest will remain locked in the contract indefinitely. Therefore, from an economic standpoint, bridging aTokens may not be advisable.

Additionally, part of the deposited aTokens will be permanently locked to the `FusionLock` contract, corresponding to interest accrued as long as the user kept these tokens locked in the contract.

Furthermore, we observed that some tokens on the list are not native to the Ethereum mainnet: OP (the token of the Optimism L2), aBTC (BNB chain), SBTC, ALEX, and STX (Stacks L2), therefore cannot be used in the `FusionLock` contract.

Alleviation

The team has informed us that they will not permit the deposit of aTokens and other tokens mentioned above, which could face problems with the bridge. Additionally, they have introduced a new feature at commit hash 455416c3184585790f8bdcb66336e47fd96efed5: not all tokens will automatically utilize the standard bridge (`bridgeProxyAddress`), as the owner now has the ability to assign a different bridge address for each token. This customization option for the bridge may prove necessary for certain tokens.

LOW-2	No actions taken for tokens failing to be bridged
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

In the case that the token on the L2 side of the bridge does not recognize the token on the L1 side as its pair token, for instance, if the `FusionLock` owner has incorrectly set the pairs, the bridging process will fail, and the bridge will return the tokens to the sender on the original (L1) chain. However, it's important to note that the sender in this scenario is the `FusionLock` contract, not the user. Consequently, in this edge case, the returned tokens will be locked to the `FusionLock` contract, since there is no functionality to withdraw them.

Recommendation

Adding logic to automatically handle this edge case would be excessive and would significantly increase the complexity of the contract. As a mitigation, we suggest off-chain tracking of the transactions and implementing a `saveTokens` function callable only by the owner to send back the tokens to users that failed to be bridged. To ensure that the owner will only save the excess tokens and not tokens deposited by users, the contract should also maintain a variable `totalDeposits` for each token, which will change with each deposit and withdrawal. The `saveTokens` function should only be callable in cases where the actual balance exceeds the one accounted for by the variable. This function would also allow the owner to send back tokens to the users who directly send them to the `FusionLock` contract, without using the `depositERC20` function.

Alleviation

The team has addressed the issue at commit hashes 6f5489e6a443d591c0d882eb5779d260bf39ce35 and 9af92bd48f01b1f5a8e068e1d7900be3f37674d0. The owner now has the capability to return excess tokens (tokens which have been sent back to the contract by the bridge due to a failure or tokens sent to the contract directly by accident) to the users.

Informational/Suggestions

INFO-1	Redundant check that ETH is allowed
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

The native token (ETH in our case) is permitted for deposits in the constructor of the contract. Therefore, whenever a user attempts to deposit native tokens (NaT) by calling `depositETH()`, there is no need to check if this is allowed or not, as is currently done in the `isDepositAllowed` modifier.

Alleviation

The team addressed the issue at commit hash 5dced5f0330e953314b5406b5a9828f24caf9cab.

INFO-2	Consider using `.call` instead of `.transfer`
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

In the `withdrawSingleDepositToL1` function, native tokens are sent back to the user using the `.transfer` method. This method forwards a fixed amount of gas (2300) and it used to be considered a measure against reentrancy. However, this approach assumes constant gas costs, which is not the case as gas costs are subject to change. Any contract using `.transfer` takes a hard dependency on gas costs and could potentially break after a future gas costs update. Additionally, the use of .transfer limits interactions with other protocols that may require multiple actions or adjustments, and therefore higher costs, to accounting variables upon receiving tokens from the `FusionLock` contract.

Recommendation

Since the contract already follows the checks-effects-interactions pattern (zeroing the user's balance before sending them their NaT), therefore there is no reentrancy risk,we believe it is safe to replace `.transfer` with `.call`.

Alleviation

The team addressed the issue at commit hash [ba0b987cf5b5828e8ac729199d7177268d8e1fe8]().

INFO-3	Modifiers applied twice in `pause()` and `unpause()`
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

In the contract there is an external `pause()` function, callable only by the contract owner. This function has the whenNotPaused modifier and calls the internal function _pause of the Pausable.sol contract. But the _pause function has also the whenPaused modifier.

The same holds for `unpause()` and the whenPaused `modifier`.

Recommendation

We suggest removing the modifiers from the external functions to simplify the code and reduce the gas costs.

Alleviation

The team fixed the issue at commit hash [0885622b0e55d4d5000e7a13f2373366340d93e6](0885622b0e55d4d5000e7a13f2373366340d93e6).

INFO-4	Use the local variable to avoid accessing storage twice
Contract(s)	`FusionLock.sol`
Status	**Resolved**

Description

In `withdrawSingleDepositToL1` and `withdrawSingleDepositToL2` the `deposits[msg.sender][token]` is being read twice, first in the `require` and then to assign its value to the `transferAmount` local variable.

```
require(deposits[msg.sender][token] != 0, "Withdrawal completed or token never
deposited");
uint256 transferAmount = deposits[msg.sender][token];
```

Recommendation

We suggest accessing it only once, store it in the local variable `transferAmount` and then use this in the require:

```
uint256 transferAmount = deposits[msg.sender][token];
require(transferAmount != 0, "Withdrawal completed or token never deposited");
```

Alleviation

The team fixed the issue at commit hash dc7a8460ba0aa687c962561d337308d4e7873093.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.