HashTimeLock
Smart Contract. Audit. Final

Mikhail Vladimirov and Dmitry Khovratovich

17th March 2020

This document describes the audit process of the HashTimeLock smart contract performed by ABDK Consulting.

1. Introduction

We've been asked to review the HashTimeLock smart contract given in a repo, commit `1e83518`. We have identified a number of issues, and all the important ones have been fixed in the new version. Those remaining do not affect the security of the contract.

2. HashTimeLock.sol

In this section we describe issues found in HashTimeLock.sol

2.1 Critical Flaws

This section lists critical flaws, which were found in the smart contract.

Line 86: if the function `newContract` is called twice with the same parameters and `msg.value`, then the same ID will generated. The information about the previous contract with such ID will be lost. The ether associated with the previous contract will become inaccessible.
Answer: This cannot happen while using the application. One must manually and intentionally send the same parameters. However, this issue was **fixed**.

2.2 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 2: there should be ^0.5.0 instead of >=0.5.0. It would look more common.
   Answer: This issue was **fixed**.

2. Line 17-18: the storing the full strings could be gas consuming. Consider storing only the hashes of these strings.
   Answer: This is left as the strings are expected to be short (e.g.. ETH, BTC, AE)

3. Line 21, 29: `Withdraw` and `Refund` events are contain redundant data that can be retrieved from `NewContract` event by using the id.
   Answer: This is left as applications already use this interface.

4. Line 24, 41: it might be better to index `hashLock` not `sender`.
   Answer: This is left as applications already use this interface.

5. Line 30: probably, `hashLock` can be used for the `id` as it should never occur twice.
   Answer: This is left as non-important.

6. Line 49, 60, 155: the `memory` modifier may cause the whole structure will be read into the memory, including the fields that are not needed for this modifier. Consider using `storage` modifier instead.
   Answer:This issue was **fixed**.

7. Line 116, 124: the `contracts[id]` already calculated inside `withdrawable` and `refundable` modifiers. Consider refactoring.
   Answer: This issue was **fixed**.

8. Line 126, 127: the `c.sender` is the same as the `msg.sender`, but the `msg.sender` is cheaper to access.
   Answer: Left as non-important.

9. Line 119, 127: the `c.hashLock` was already read from the storage inside `withdrawable` modifier. Consider refactoring.
   Answer: This issue was **fixed**.

10. Line 131: the function `getContract` seems redundant as the function `contracts` is already public.
    Answer: This issue was **fixed**.

11. Line 137: the `contracts[id].status != SwapStatus.INVALID` relies on the non obvious fact that `INVALID` is the default value for `SwapStatus` enum.
Answer: This issue was **fixed**.

12. Line 154: the code could be made more readable and less error prone by replacing `SwapStatus enum` with a collection of separate constants.
Answer:This issue was **fixed**.

2.3 Unclear Behaviour

Line 68: the variable `outputAmount` is not used and is only logged.
Answer: This was found to be on purpose.

2.4 Documentation and other Issues

This section lists documentation and other minor issues which were found in the token smart contract.

1. Line 7: the `EXPIRED` is the status between the `ACTIVE` and the `REFUNDED`, which is not obvious. Some comment is needed.
Answer: This issue was **fixed**.

2. Line 120, 128: functions `withdraw` and `refund` always return true
Answer: This issue was **fixed**.

3. Line 159: the condition `tempContract.expiration < block.timestamp` can be inaccurate when queried out of execution.
Answer: This is left out as OK, but an application should take that into account when querying the contract.

3. Summary

We found no critical or serious issues in the new version of code where our comments were taken into account.