AdHive Smart Contracts Security Analysis

Abstract

In this report we consider the security of the Adhive multisig vesting contracts. Our task is to find and describe security issues in the smart contracts of the exchange platform.

Procedure

In our analysis we consider the AdHive smart contracts code and documentation (version on commit 0d00760).
We perform our audit according to the following procedure:

- Automated analysis
  - We scan the project's smart contracts with the Solidity static code analyzer Smart-Check
  - We scan the project's smart contracts with several publicly available automated Solidity analysis tools such as Remix and Securify (beta version since full version was unavailable at the moment this report was made)
  - We manually verify (reject or confirm) all the issues found by the tools
- Manual audit
  - We manually analyze smart contracts for security vulnerabilities
  - We check the smart contracts' logic and compare it with the one described in the documentation
- Report
  - We reflect all the gathered information in the report

Disclaimer

The audit does not give any warranties on the security of the code. One audit can not be considered sufficient. We always recommend proceeding to several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, this security audit is not an investment advice.

About The Project

Project Architecture

The smart contracts code consists of the following files:

- MultiSigWalletTokenLimit.sol — core implementation vesting multisig. Allows you to create a period to withdraw a certain amount of tokens with the consent of contract owners.
  Contained contracts:
  - ERC20 contract - standard interface to interact with token’s functions
  - MultiSigWallet contract - standard implementation based on Gnosis Multisig
  - Receiver contract - contains token fallback function describing
  - MultiSigWalletToken - contains token vesting logic

Code Logic

The Code logic is described in detail in the documentation. We have checked the logic implemented in the code and compared in with the one described in the documentation. We found no discrepancies.

Checked vulnerabilities

We have scanned the AdHive smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered (the full list includes all these but is not limited to them):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference

Manual Analysis

We manually analyzed all contracts, checked their logic and compared it with the one described in the documentation. In addition, we manually verified all results from the automated analysis. We describe confirmed issues below.

High severity issues

Insufficient testing

Testing is a crucial process to make sure that smart contracts work as expected. To ensure an adequate level of security, we highly recommend writing unit tests for your smart contracts logic, using a modern framework such as Truffle. We recommend using Ganache + Truffle + Module and changing the time in virtual EVM to check the logic underlying the vesting multisig contract. During testing, you should pay special attention to the `updateCurrentPeriod()` method. You can rely on these materials (https://github.com/gnosis/MultiSigWallet/tree/master/test) for the main functions of the tests. After testing, you can check the coverage of the code using the Solidity-Coverage tool.

Medium severity issues

Medium issues can influence smart contracts operation in current implementation. We highly recommend addressing them.

Unchecked math

Solidity is prone to integer over- and underflow. Overflow leads to unexpected effects and can lead to loss of funds if exploited by a malicious account.
Recommendation: check against over- and underflow (use the SafeMath library).

Lines: 546, 468, 163, 551, 398, 378, 467, 166, 419, 161, 376, 550

Unused parameter

The function `tokenFallback` has an extra parameter of the bytes type called 'data'. Judging by the description of the function, it is only responsible for receiving tokens. Therefore, this parameter is superfluous, and you can delete it. It will then also be necessary to remove this parameter from the description of the interface of the function.

Line: 394, 7

Ambiguous Naming

In the function `addPeriod,` before adding a new period, you check the ability to update the period if timestamps match. Therefore, it would be better to rename the function `updateOrAddPeriod.`

Costly Loops

If there are many periods of vesting, then the execution of functions (`updateCurrentPeriod, addPeriod, deactivatePeriod`) can lead to exceeding the gas limit in the block. This may mean that further transactions will never be executed, and, consequently, tokens may be lost.
You could think about alternative architecture, or neatly add periods. If the number of periods is no more than 50, then this is not critical, although operations will be more expensive.
However, we strongly recommended to initially set a higher possible number of periods, as is done with the maximum number of Multisig contract owners.

Low severity issues

Low severity issues can influence smart contracts operation in future versions of code. We recommend to take them into account.

Constant functions

The function is declared as `constant`. Currently, for functions, the `constant` modifier is a synonym for `view` (which is the preferred option). Consider using `view` for functions and `constant` for state variables.
Recommendation: Declare functions which promise not to modify the state as `view` and not `constant`. At the present time `constant` is an alias to `view`.

Multisig logic Issue

Currently, if the number of confirmations is greater than required, then the transaction is not considered to be confirmed. This derives from a bug in the standard multisig contract. It is more logical to assume that if the number of exposures is, for some reason, greater than it should be, the transaction is also deemed to be confirmed.

Line: 269 Function: `isConfirmed`

Compiler version not fixed

Solidity source files indicate the versions of the compiler they can be compiled with.

pragma solidity ^0.4.17; // bad: compiles w 0.4.17 and above
pragma solidity 0.4.17; // good : compiles w 0.4.17 only

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Other

In any period, there may be tokens that remain unlocked until the end of the period. At the time the next period begins, we reset the number of tokens to a predefined maximum. However, the current code does not withdraw any of the remaining unlocked tokens. This means there are more tokens unlocked than the maximum, but only the defined maximum number of tokens can be withdrawn. This is not a bug because we can create another period and unlock any excess tokens, but it should be kept in mind.

Conclusion

In this report we have considered the security of AdHive smart contracts. We performed our audit according to the procedure described above.
The audit showed very high code quality and low number of vulnerabilities per number of lines of code. However, we have found one high severity issue and several medium and low severity issues. We highly recommend addressing them.

This analysis was performed by Zerion

Audit conducted successfully:

ZERION

Vladimir Tidva [illegible]