Coinvest V3 Audit

September, 2018

Report by Authio Version 1.4

Alexander Wade, Alex Towle, Paul Vienhage

Contents

1. Overview 3
2. Introduction 3
    2.1 Authenticity 3
    2.2 Scope 3
    2.3 Methodology 3
    2.4 Terminology 4
    2.5 Disclaimer 4
3. Findings 5
    3.1 Contract Explanation 5
    3.2 Critical Severity 6
    3.3 High Severity 8
    3.4 Medium Severity 8
    3.5 Low Severity 9
    3.6 Notes & Recommendations 10
4. Documents & Resources 11
    4.1 Line-by-Line Comments 11
    4.2 Project Codebase 11
    4.3 Outside Resources 11
5. Conclusion 12

1. Overview

This document serves as the official audit report of a set of contracts created and published by Coinvest. Coinvest intends to deploy two tokens: COIN and CASH, where the COIN token is an update of their current live token. In addition to these two tokens, Coinvest intends on launching a cryptocurrency exchange where COIN and CASH tokens are used as the mediums of exchange. In addition to allowing users to trade many popular cryptocurrencies, the Coinvest exchange will also allow users to trade cryptocurrency inverses, which have a value inverse to the value of their tied cryptocurrency.

2. Introduction

2.1 Authenticity

The audited contracts are in the CoinvestV2Audit repository: https://github.com/RobertMCForster/CoinvestV2Audit.

The version used for this audit is commit 27178d3a3b07f9ab16c5efbaadc54b1197e64f68 .

2.2 Scope

This audit included all of the contracts in the CoinvestV2Audit/contracts/ directory, with the exception of the CoinvestTokenV2.sol file, which was reviewed by Authio in a past audit. These contracts as well as previous versions of the COIN token are contained in the CoinvestV2Audit repository:

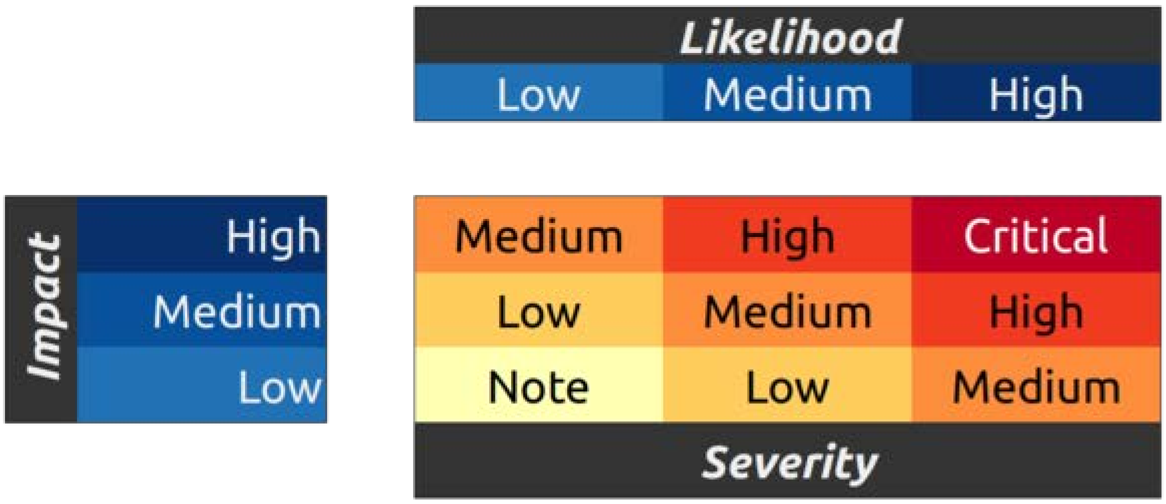
https://github.com/RobertMCForster/CoinvestV2Audit

The Coinvest platform will contain components that are off-chain. Both the security of these components and the security of their interaction with the on-chain components are outside the scope of this audit.

2.3 Methodology

This audit focuses heavily on not only inspecting the smart contracts for vulnerabilities and potential for losses in funds, but also on working closely with Coinvest to scrutinize the contracts for execution of intent. The end goal of this audit is to help Coinvest not only secure their contracts, but also to ensure their vision for the project is best represented by the project they put forward. As a result, additional concerns such as efficiency and design are included in this report as well.

2.4 Terminology



Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Note	Low	Medium
	Severity		

This audit categorizes vulnerabilities using the OWASP risk rating method based on impact and likelihood. Each vulnerability is given impact and vulnerability scores, which are used to give a more accurate estimation of the overall severity of a vulnerability. An additional factor in severity is the relative ease with which a vulnerability is fixed: an issue which requires extreme refactoring will be weighted higher than one with the same severity which is a quick fix.

2.5 Disclaimer

This document reflects the understanding of security flaws and vulnerabilities as they are known to Authio, and as they relate to the reviewed project. This document makes no statements on the viability of the project, or the safety of its contracts. This audit is not intended to represent investment advice and should not be taken as such. The Coinvest platform contains large off-chain components, and this audit should be viewed as only a component of a holistic assessment of the security of the platform.

3. Findings

3.1 Contract Explanation

3.1.1 Function - Token Functionality (Transfers, Approvals, etc)

CashToken.sol and CoinvestToken.sol: The CASH and COIN tokens extend the ERC20 token standard by adding functionality that allows users to increase or decrease the approval of another user. Another feature is the ability to approve and call another contract. Additionally, each token is compatible with the ERC865 standard, allowing users to interact with the token through a delegate by simply providing their signatures. These functions allow users to sign a transaction which can then be redeemed by other users of the contract in exchange for an agreed-upon fee paid in COIN or CASH tokens. This fee allows users to pay for transaction costs in COIN or CASH instead of ether. Finally, the CASH token also supports functionality that enables the owner of the contract to mint and burn new CASH tokens.

3.1.2 Function - Token Upgrade

TokenSwap.sol: The `TokenSwap` contract implements a swap mechanism through which holders of the COIN V2 token are able to exchange their tokens for the new COIN token.

3.1.3 Function - Exchange Functionality

Investment.sol : This contact retrieves cryptocurrency prices from an oracle using the Cryptocompare API and the Oraclize project. It enables users to create an audit trail of transactions of major cryptocurrencies and cryptocurrency inverses using the COIN or CASH token as payment. The contract uses external contracts for the data storage of COIN and CASH token balances, as well as user trade history.

3.1.4 Function - User Data Storage

UserData.sol: The `UserData` contract keeps an active record of the cryptocurrency holdings of the users of the application. This record is updated whenever a **buy** or **sell** transaction is processed through the Investment contract.

3.1.5 Function - COIN and CASH Storage

Bank.sol : The Bank contract holds COIN and CASH tokens for the Investment contract as a part of the payment process. This contract increases the modularity of the application by decoupling the Investment contract from holding payments.

3.1.5 Function - Utility Libraries

Strings.sol : A string library that provides efficient utility functions for performing string manipulations and comparisons. This is used extensively in the Investment contract to construct API requests.

SafeMathLib.sol : The `SafeMathLib` library provides arithmetic functions that prevent integer underflows and overflows from occurring.

Ownable.sol : The Ownable contract creates a permission system that distinguishes an owner address and an address for the Coinvest company. These distinguished addresses have access to administrative level functionality within the application.

3.2 Critical Severity

Price Manipulation in Investment and InvestmentTest Currency Price Decoders:

The Coinvest exchange's price decoder has a flawed implementation that allows for some undesirable behavior to occur. Another issue is the function's susceptibility to issues that result from the centralization of the Cryptocompare API. The combination of these issues presents several exploits that allow users to purchase cryptocurrencies at a discount or sell cryptocurrencies at a higher price point.

The findings from our review of this function are detailed below:

Code Errors:

1. A typo in the price decoder function incorrectly references CASH's internal ID. CASH and its inverse have IDs of 21 and 22, respectively, but the function references ID 20 and 21, COIN's inverse and CASH, respectively. This typo will cause the function to incorrectly handle the CASH token when it is being used as the medium of exchange for the transaction.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

We suggest that the typo be fixed, and that COIN and CASH are given contract-wide constants, reducing the possibility that typos are made in the future.

2. The current Cryptocompare API does not return multiple results if the price of a currency is queried multiple times. A flaw in the implementation of the price decoder causes a transaction that purchases one currency multiple times to set all instances of the cryptocurrency following the first to have a price of zero. This issue allows a party to purchase cryptocurrencies for free as the first purchase amount can be set to zero.
3. A similar situation occurs in the event that a party queries a currency multiple times, followed by a different currency. This will cause the second instance of the repeated currency to be set equal to the price of the unique currency. Unlike the previous problem, this situation allows parties to sell tokens for substantially more than they are actually worth.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9. Coinvest did not take our suggestion; however, their solution satisfactorily mitigated the above vulnerability.

To solve the two issues above as well as some efficiency issues with the crypto IDs system, we recommend that the price decoder function and some of the state variables be refactored. The new system would not include the `tiedInverses` mapping or the `isInverse` mapping. Instead, the layout of the `cryptoSymbols` mapping would be altered so that every even numbered index would represent a normal cryptocurrency, with all odd numbered indices representing inverse cryptocurrencies. Additionally, the index following a normal currency would always represent its inverse. This layout would provide the same functionality as the `tiedInverses` and `isInverse` mappings while reducing the overhead of much of the application's logic.

To simplify the `decodePrices` refactor, we suggest that a mapping from strings to uints be added that would be the inverse of the `cryptoSymbols` mapping. In the price decoder, we recommend switching to a system where a temporary memory array is allocated with a size equal to the max crypto ID in the list of cryptos to get prices for. The API callback string should be parsed so that on every iteration the function obtains a symbol and that symbol's price. Using the inverse of the `cryptoSymbols` mapping, the crypto ID associated with the symbol can be queried in memory. The price should be placed at the index of this crypto ID in the temporary array. Once the entire callback string has been parsed, the prices array can be calculated by looking up the prices of a given crypto ID in the temporary array. A check should be made such that no element of the prices array is set to zero, which will catch several of the potential issues with the API's callback string.

API Centralization:

The following issues will not present themselves under the current version of the Cryptocompare API. This said, we have identified several edge cases which would cause issues in the event that the API was changed. While the fact that the API needs to change to exploit these issues makes these vulnerabilities seem more benign, these issues present an opportunity for a party in control of the Cryptocompare API to change the API itself in an effort to profit from the current price decoder implementation.

1. When the previous price parsed was COIN, the contract will fail to "advance" to the next point in the API callback string when parsing the current price. As a result, the price of the current currency is set to that of COIN.
2. When the previous price parsed was the price of an "inverse" of the price before it, the contract will fail to "advance" to the next point in the API callback string when parsing the current price. As a result, the price of the currency is set to that of the price of the inverse of the previous currency.
3. In the event that a cryptocurrency that is supported by the Investment contract becomes unsupported by the Cryptocompare API, the API callback string will be improperly advanced. In the best possible scenario, the price of one

cryptocurrency will be set to zero. In a more likely case, several cryptocurrencies will be mispriced, with the last cryptocurrency's price set to zero.

Note: An explicit pausability mechanism was added in commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

To avoid potential problems with future changes to the Cryptocompare API, we suggest that a pausability mechanism be added to the Investment contract to allow the execution of the contract to be halted while fixes are made. This will not prevent a change in the Cryptocompare API; however, it will provide a mechanism through which Coinvest is able to adjust their on-chain settlement in an emergency.

3.3 High Severity

No high priority issues were found.

3.4 Medium Severity

Unsafe Oracle funding This problem occurs in `tokenEscape` in Investment and InvestmentTest:

The Investment contract relies on Oraclized API calls to determine currency prices, which require it to provide Ether with each request. To this end, the Investment contract provides a payable fallback function, to which any address can send Ether to fund API calls made by the contract. The contract's `buy` and `sell` functions both create API calls if the contract has sufficient balance, but are not themselves payable. This means any API calls made require that the contract already be funded.

Since the only method of funding the contract takes place via the fallback function, it can be assumed that Coinvest intends to provide sufficient ETH for the API calls; otherwise, the best option would be to allow payable `buy` and `sell` functions so that a user does not need to fund the contract and execute a buy or sell in separate calls to the contract. However, if it is the intention of Coinvest to provide ETH for API calls, the implementation does not make this sufficiently clear: a user is able to fund the contract themselves via the fallback as well, and are even able to do so safely by creating a contract that chains the two required contract calls together. If a user simply makes two separate, signed transactions (one to fund the contract, and the second to execute the API call), the user's Ether is at risk between the two transactions: either from consumption by another user, or from Coinvest's use of the `tokenEscape` function.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

In order to prevent what could be a very understandable mistake on the user's side, the Investment contract's implementation should make clear statements about what is the "intended" use, and what is the "allowed" use. If Coinvest wishes to fund the API calls of its users, the fallback function should not allow any user to submit Ether to the contract. If users are meant to provide their own payment for the API, the `buy` and `sell` functions should be payable, so that the user will be able to fund their API calls safely and simply.

Transferring Ownership to the Token Contract and Stealing Stuck Tokens in cashToken and CoinvestToken:

The `receiveApproval` function allows the contract to call itself, which allows users to share a privilege level with the contract; they are able to force the contract to call arbitrary functions with arbitrary data, as itself. Thus, if the token contract is given admin level access then every user will also have admin level access, which is an escalation of privilege vulnerability. The situation under which this vulnerability is the most dangerous is if Coinvest transfers ownership of the contract to the contract's own address as this would allow users to `mint` and `burn` tokens through the CASH token contract at will.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

We recommend that the call in `receiveApproval` be changed to a `delegatecall` because then the sending address would be preserved during the call.

Additionally, this function allows anyone to take any tokens that had been accidentally sent to the token contract's address. When the `receiveApproval` function calls the contract that it resides in, the sender address becomes the token contract's address. Since the sender address is now the token contract's address, users can use this function to call the token contract's approve function to enable themselves to take the tokens that are owned by the token contract. The purpose of the function `tokenEscape` is to allow the owner of the contract to remove these "stuck tokens", so the ability to remove "stuck tokens" through the `receiveApproval` function is an escalation of privilege vulnerability.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

To fix this issue, we recommend that the `call` in `receiveApproval` be changed to a `delegatecall`.

3.5 Low Severity

Array Out of Bounds Error for `returnHoldings` in UserData:

Instead of starting to set the return array at index zero, currently the loop starts by setting index `start`. This issue will cause the function to set incorrect elements of the array as long as `start` is greater than zero. Since Solidity checks for out of bounds assignments in arrays, this issue will cause the call to revert if start is greater than zero.

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

We recommend that the loop index be set to zero at the beginning of the loop rather than `start`. This will ensure that the correct indices of the returned array are being assigned. This change will also require updating the line inside of the for loop. Namely '`userHoldings[_beneficiary][i]`' should be updated to '`userHoldings[_beneficiary][i+start]`'.

Revocation is Permanent for `revokeHash` and `revokeHashPresigned` in CoinvestToken and Cash Token:

These functions should increase the nonce of the account whose transaction was revoked. If they do not the account could not issue the transaction again because the hash of the transaction at that nonce would be marked as invalid.

Note: This issue has been resolved in both the CASH and COIN token contracts over the course of several commits.

We recommend that both functions increment the signing party's nonce in the event that a hash is revoked.

3.6 Notes & Recommendations

Presigning Race Conditions : Every presigned function in COIN and CASH token is subject to a race condition. An attacker could see a dispatcher broadcasting a signed transaction on the behalf of a user, copy the data, and then race the dispatcher to claim the token reward for submitting it. This is only an efficient way to claim tokens if the specified gas prices allow arbitrage from the token cost.

Unused parameter in `__callback` in Investment and InvestmentTest:

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

The proof field is entirely unused. If the field is necessary, the compiler warning can be silenced by adding the line `proof;` to the contract.

Gas inefficiency for `bitConv` in Investment and InvestmentTest:

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

The first `|=` operation in the for loop can be changed to simply a = operation. This will be more gas efficient.

Bank Constructor:

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

In Solidity version `0.4.22`, a new syntax for constructors was introduced. There is a compiler warning in the Bank contract because the new syntax is not used. To silence this compiler warning, we recommend that the new constructor syntax is used.

Unnecessary Require for `_transfer` **in CashToken and CoinvestToken**: The second require statement is unnecessary as the use of the safe math library eliminates the possibility of integer underflows.

Unnecessary Safe Subtraction for `_decreaseApproval` **in CashToken and CoinvestToken**:

The use of safe subtraction in the else clause is unnecessary because

Unnecessary Require for `burn` **in CashToken**: The second require statement is unnecessary as the use of the safe math library eliminates the possibility of integer underflow.

Specify Compiler Version in Ownable and SafeMathLib:

Note: This issue has been resolved as of commit b937a0a0393b04bbae05152a5ec0c9296ea040b9.

There is currently a compiler warning in both of these files because a compiler version is not specified. We recommend that compiler version 0.4.24 be specified as the compiler version for `Ownable.sol`.

4. Documents & Resources

4.1 Line-by-Line Comments

https://github.com/authio-ethereum/Audits/tree/master/CoinvestTokenV3

4.2 Project Codebase

https://github.com/RobertMCForster/CoinvestV2Audit

4.3 Outside Resources

Two references on the ERC865 token standard are here and here.

The Cryptocompare API documentation is here and the Github repo for the Orcalize project version used is here.

5. Conclusion

Our review found one critical vulnerability in the Investment contract’s `decodePrices` function, as well as some medium and lower severity issues. Additionally, due to the nature of the vulnerabilities found in the Investment contract, our recommendation was to rewrite the contract. We provided recommendations for addressing each of the issues presented.

Aside from the Investment contract, the contracts audited presented only surface-level flaws. With some review to adhere to best practices, as well as careful consideration of every issue addressed, Coinvest will be able to ensure a consistent level of fidelity across the repository.