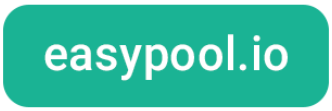

EasyPool Smart Contract Security Audit v2

April 11, 2019 Ariel Yabo No Comments

Reading Time: 8 minutes

Contents

Introduction

CoinFabrik has been hired to audit the EasyPool smart contracts.
We start this report writing a summary with the smart contracts provided by the client and a list of the analysis methods used to audit the contracts. Next, we will make a summary of the files we analysed and the public facing functions provided by the ProPool contract. Then we detail our findings ordering the issues by severity, followed by all observations we considered important to add. Furthermore, we ended up this audit report with a conclusion explaining how do the auditors value the code reviewed, and what are the most important things that need to be corrected to it to make it work flawlessly and securely.

Summary

The contracts audited are from the EasyPool repository at https://github.com/gitigs/easypool. The audit is based on the commit *17a1e1ae336a92e3d4d7686aa1cb26aaea3f1f82*.

The audited contracts are:

- ProPool.sol: Proxy contract for the investor pool manipulation functions
- common/Affiliate.sol: Affiliate management functions
- common/CMService.sol: Contract that provides functions for deploying a new pool contract and setting the fee service, the pool factory and the registry.
- common/FeeService.sol: Contract that manages the fee in a given pool
- common/PoolFactory.sol: Pool deployment function
- common/PoolRegistry.sol: Pool registering implemented through the emission of events
- common/Restricted.sol: Operator management functions for restricting use to given operators.
- library/ProPoolLib.sol: Code for pool initialization and lifetime functions, managing deposits, whitelist and withdrawal.
- library/QuotaLib.sol: Share claiming functions for manipulating the quota given by the pool

The *ProPool* contract provides a public interface for managing the pool. The implementation is in the *ProPoolLib* contract. This latter contract uses another library, defined in *QuotaLib*, to define an internal data structure within the contract's representation of an investment pool. There's also a dependency on a pair of functions from the FeeService contract.

As for *CMService*, it depends on the interface for *PoolRegistry* and *PoolFactory*, which inherit from the *Restricted* contract, adding operator management functions. *PoolFactory* also depends on *ProPool*, since it deploys new instances of the *ProPool* contract. *CMService* allows setting fee services, changing the pool factory contract, and such functions. PoolRegistry brings a function to register new pools, emitting an event.

Fortunately, on analysis of these contracts only a few minor issues, mostly with maintainability, were found. We proceed to detail the checks we've made, the improvements to the contracts which could help development, and a conclusion.

As for the audit, the following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

File Summary

ProPool.sol

No vulnerabilities were found in this contract, but some compiler warnings were noted. This is a contract that provides public-facing presale pool management functions, which are implemented in the ProPoolLib library.

Affiliate.sol

A lack of use of SafeMath was noted, but no issues were found. This contract has functions for controlling affiliates within the contract.

CMService.sol

No vulnerabilities were found, as this contract was simple and straightforward. This contract can be called to set the addresses of the fee service, the pool factory, the pool registry and to deploy the ProPool contract by a call to *poolFactory*. This way the implementation can be changed without redeploying.

FeeService.sol

This contract does not use the SafeMath library consistently, but no major security concerns were found. The objective of this contract is to provide fee settings, which are then attached to the pool through the functions provided in CMService.sol

PoolFactory.sol

No vulnerabilities were found in this contract. Since it provides a simple function for the deploying of a new ProPool contract, no vulnerabilities are considered as possible.

PoolRegistry.sol

No vulnerabilities were found in this contract. It's a simple contract providing a function that emits an event when called, recording the pool address and details.

Restricted.sol

No vulnerabilities were found here. This is an implementation providing storage and functions that manage operators, which are used to restrict function execution in PoolFactory.sol, PoolRegistry.sol and Affiliate.sol

ProPoolLib.sol

The inconsistent use of SafeMath was noted, but no major issues were found. This contract is the one providing the implementation that ProPool.sol uses.

QuotaLib.sol

Inconsistent SafeMath usage was noted, along with a lack of documentation. While later commits added some, it still wasn't enough. Other than that, the contract was straightforward, without much complexity. This contract is used to manipulate the refund and token quotas in the ProPoolLib.sol contract.

Function summary

The ProPool contract has many public functions. We will describe how they fit together giving a brief description of what they do.

constructor

Calls init(), initializing the pool fields. It doesn't initialize the pool state, leaving it in the default of open. It creates the first group of eight calling *setGroupSettingsCore*.

Once created, the pool accepts deposits. If it's cancelled, the contributions can be withdrawn together with the remaining balance. If it proceeds, once the presale address is paid, the *confirmTokenAddress* function can be called, changing the state of the pool to *Distribution*, and sending the creator and service fees. Elsewise, if the refund sender address is set, the pool goes into *FullRefund* state, and investors can claim their refund share and tokens.

setGroupSettings

Calls *setGroupSettingsCore*, and then calls *groupRebalance*, which balances contributions made to the group.

cancel

Sets the state of the pool to Cancelled.

deposit

Contributes to a certain group. Internally, it calculates the contribution, sets the group as existing and updates the group contribution.

modifyWhitelist

Enables whitelist, includes participants who should get in and excludes those who shouldn't, as sent by parameter, in a certain group. Afterwards, it rebalances the group.

payToPresale

If presaleAddress isn't set yet, it does so at this moment. Then sets the pool in PaidToPresale mode, and sets fee-to-token mode, if needed: commission is sent to the presale address. Then the funds are set to the presale address by calling addressCall, which calls the address with the needed value and data, and emits an event.

lockPresaleAddress

Sets presaleAddress and sets the *lockPresale* boolean, which should be used for verification. This boolean is part of a locking mechanism which should be used, but is never again used in the code.

confirmTokenAddress

Changes state to distribution, saves parameter tokenAddress in the tokenAddresses array if balance > 0.

setRefundAddress

Sets the refund sender for the pool (for use in the fallback function)

Fallback function

Verifies sender is the specified refund address.

withdrawAmount

Withdraws from a group: all of participant remaining balance and some of participant contribution. Then it transfers this amount to the sender.

withdrawAll

What this function does varies according to what state the pool is in:

In FullRefund or Distribution states it calls withdrawRefundAndTokens, which calls withdrawAllRemaining, taking the group's remaining balance and transferring it to the sender. Then it calls withdrawRefundShare. This is the part that withdraws the refund shares. Then withdrawRefundAndTokens checks if the pool is in fee-to-token mode (where the fee for the creator is sent directly to the presale), and if there was an effective contribution to the token addresses they are transferred.

In cancelled or open states it calls withdrawAllContribution, which takes all of the sender's contribution and remaining from each and sends him that amount.

In PaidToPresale state it only withdraws the remaining balance of the participant and sends it his way.

tokenFallback

Emits an event for registering. This function comes from ERC223, but this proposal didn't prosper much.

getPoolDetails1

Getter for pool structure.

getPoolDetails2

Nothing if not in Distribution or FullRefund states.

Else, calculates the balance it will refund and returns the token addresses and balance in each

getParticipantDetails

Gets contribution, remaining and whitelist array.

getParticipantShares

Calculates refund shares and calculates shares (total – claimed) of each tokenAddress in the pool for a certain address

getGroupDetails

Simple getter for group details, returning the respective fields.

getLibVersion

Gets the library version.

Detailed findings

Critical severity

None found.

Medium severity

None found.

Minor severity

Lack of usage of "lockPresale" boolean

The pool has two fields: *presaleAddress* and *lockPresale*. These control where the balance will be sent when calling *payToPresale* and whether the address is already set. On functions *init* and *payToPresale* the *presaleAddress* field can be set via a parameter. However, the *lockPresale* boolean isn't set, defaulting to false. This means that the internal state will be inconsistent. Furthermore, checks to see if the presale address was set are made based on whether it is zero or not, and the boolean, when set (which is only done when calling *lockPresaleAddress*), is never actually used for making the check on if it was locked or not.

The state of the pool is not explicitly initialized

When calling the init function, the pool's state is never explicitly initialized. This means that it'll default to the "Open" state, but it still is advised to change this as it can impact when changing the code and versions.

Warnings on compilation

The contracts emit warnings when compiled. It would add to the auditability of the contracts themselves if they were removed. While they may not cause any problem directly, they can occlude other more important warnings. We recommend fixing these before deployment, mostly as a way of ensuring that everything's going well.

Inconsistent usage of SafeMath

There's not enough usage of SafeMath in some lines in QuotaLib.sol, yet it is declared as being used in the following statement:

```
library QuotaLib {

using SafeMath for uint;
```

We recommend either removing the statement, or following through in the usage consistently, since this could lead to errors and possible future changes could cause bugs. We also recommend the use of SafeMath in the other contracts, which it can be seen that it was considered yet decided against in some commented out parts of the code, which declare

```
// We could use SafeMath to catch errors in the logic,

// but if there are no such errors we can skip it.

// using SafeMath for uint;
```

This is something which was found in ProPoolLib.sol, in Affiliate.sol and in FeeService.sol. While it is true that SafeMath has no purpose if the code is correct, it can act as a safeguard if that isn't the case. It'd be advisable for the team to reconsider this design decision, since it'd mean arithmetical operations are guarded against overflow.

Enhancements

Not enough documentation

In the QuotaLib.sol contract there's not enough documentation. It'd be advisable for the development team to fix this, as it helps auditability and legibility of the contracts. There are many comments of the form:

```
/**

* @dev ...

*/
```

Changing these to have meaningful commentary can avoid errors when handling the contract, so it's suggested to follow through.

Non declarative function naming

There are functions named successively (getPoolDetails1, getPoolDetails2,..., withdrawAllRemaining1, withdrawAllRemaining2,...). These are poor names that don't reflect what the functions actually do, and can lead to trouble when modifying the code in the future. Clearer names would reflect the differences between them.

Old compiler pragmas

The contracts have the following pragma directive:

pragma solidity ^0.4.24;

We remind the development team that the Solidity compiler has been updated. While this isn't critical in these particular contracts, there are bugfixes which if absent could've caused problems. As usual, it is recommended to stay on top of the new versions, which could avoid problems with things like the ABI encoding, parser issues, et cetera.

Missing deployment scripts

We inferred the deployment order for the contracts based on what they do, since they follow a factory pattern, and we gathered the following inheritance graph for the contracts (taking into account the libraries):

HasNoEther was removed from OpenZeppelin

The contracts CMService, PoolFactory and PoolRegistry inherit from HasNoEther, but it was removed from OpenZeppelin, since it could be misleading, as there's no actualy way to guarantee a contract won't have ether. The following discussion is cited:
https://github.com/OpenZeppelin/openzeppelin-solidity/pull/1254

Observations

We annotate some of the verifications done to the contracts to indicate that they were performed even when no critical issue was found during the audit.

- Misuse of the different call methods: *call.value()*, *send()* and *transfer()*.
  We found no incorrect use of *call()* or *send()*, and *transfer()*. Calls to these methods were done on checked input, with correct coverage of requirements.

- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.

- Race conditions such as reentrancy attacks or front-running.
  - The only call to an external contract is in the *addressCall* function, but considering it's private and is only called from *payToPresale*, which calls to this function with the *presaleAddress* that's set. Since this is only callable by the pool administrator and the state change of the pool is done before the call, the pool will have a consistent internal state, so reentrancy can't cause damage.
  - There are many setters for addresses that could if not enough checks are done on input, cause transfers to unwanted addresses. However, these functions have the correct modifiers so only the administrator of the pool can change these.

- Misuse of block timestamps, assuming things other than them being strictly increasing.
  - Timestamps aren't used in the contract at all, so attacking the contract from that angle is impossible.
- Contract softlocking attacks (DoS) / unbounded gas usage.
  No function in the contract has a loop that can be abused to cause a soft lock or an unbounded usage of gas.

Conclusion

No critical issues were found, yet some possible legibility issues were considered. Overall the contracts had decent documentation, except for some points, but in general they were simple enough to follow that it wasn't critical. There is some code which we recommend reviewing, but in general the state of the codebase is good, the only trouble being somewhat inconsistent snippets which reduce legibility, and could also lead to bugs if additions are made.

Disclaimer: This audit report is not a security warranty, investment advice, nor an approval of the EasyPool project since Coinfabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.

Related Posts

EtherParty Smart Contract Security Audit

Coinfabrik team has been hired to audit Etherparty smart contracts. Firstly, we will provide a...

RCN BasicCosigner Smart Contract Security Audit

Coinfabrik security audit's team was asked to audit the BasicCosigner contract. This contract is part...

EasyPool Smart Contract Security Audit

CoinFabrik has been hired to audit the EasyPool smart contracts. We start this PDF report...

Bricks4US Token Smart Contract Security Audit

CoinFabrik was asked to audit the contracts for the Bricks4Us token sale. Firstly, we will...

Tagged as: easypool, security audit, smart contract audits

Categorized in: Smart Contract Audit

←Dockerized Ethereum Private Testing Environment Compatible with MetaMask and Remix

A Short Guide Through the Universe of Blockchain Based Micropayment Systems→

Blockchain Development Company | Smart Contract Audit |
Windows Driver Development | Outlook 365 Plugin Development | 区块链技术