Ethearnal Crowdsale Audit

November, 2017

Alexander Wade

Contents

1. Overview

This document serves as the official audit report of a crowdsale contract, created and published by Ethearnal. Ethearnal seeks to create an Ethereum implemented P2P freelance platform, where employers and freelancers are able to meet to create projects and products. Ethearnal seeks to make this interaction as trustless as possible, governing disputes through smart contracts and third-party moderators, and creating economic incentives for all parties to act in their best interests.

The project audited is Ethearnal's take on a next-generation ICO: dynamic and trustless, with power allocated to the token holders. The ICO itself continually mints tokens for each purchase according to a preset token value. At the end of the sale, the team is immediately given 25% of the tokens generated during the sale. In this way, the market is able to find a sort of equilibrium where token holders decide how much dilution of tokens they are comfortable with, along with the overall amount of Ether given to the team.

Concluding the ICO, the team is rewarded with 10% of the funds raised immediately. When the funds raised run out, the team makes a request to a voting contract asking for another 10%. Whether they receive their funds or not is entirely in the hands of the token holders, as the decision is made by majority vote. To safeguard against inactive token holders, each week the withdraw request does not reach the necessary 51% votes, the amount of votes required to make a decision lowers by 10%.

In addition, token holders can at any time call for a total refund using the same voting process. For this, the quorum of voters is still 51%, but the amount of "YES" votes required to enact a refund is 65%.

2. Introduction

2.1 Authenticity

The audited contracts are in the Ethearnal SmartContracts repository: https://github.com/Ethearnal/SmartContracts. The version used for this audit is commit **6f2fec5894f9c90137b7852f42c972ad4a5ffa0b**. The main contract for the crowdsale itself is located in EthearnalRepTokenCrowdsale.sol. The main contract for the treasury is located in Treasury.sol. The main contract for the voting function is located in VotingProxy.sol.

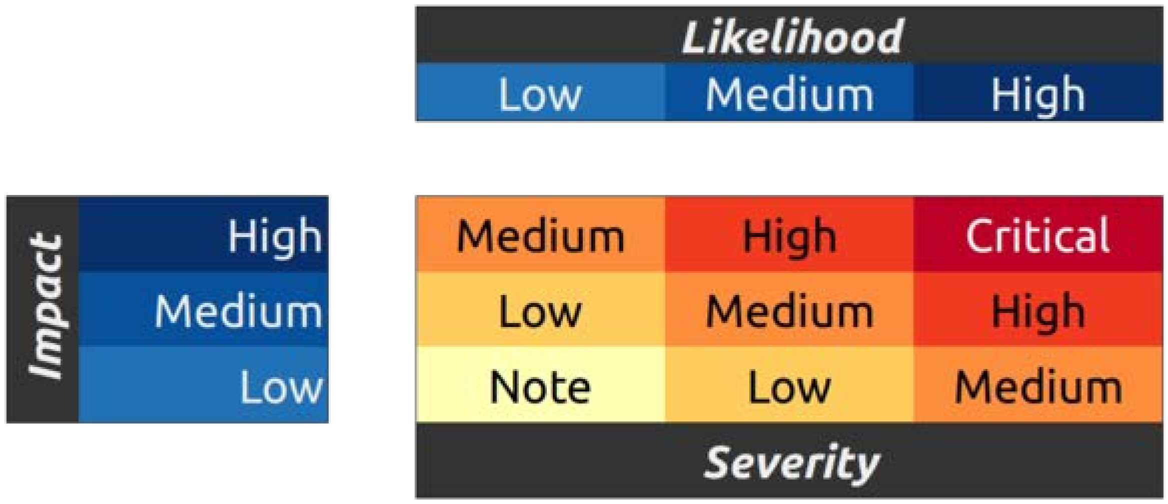
2.2 Scope

This audit covered all of the solidity files located in the contracts folder of the SmartContracts repository: https://github.com/Ethearnal/SmartContracts/tree/master/contracts. This audit does not cover files located in any other folder.

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 the Ethearnal team to scrutinize the contracts for execution of intent. The end goal of this audit is to help the team 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 the notes section of 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 subjectively given impact and vulnerability scores, which are used to give a more accurate estimation of a vulnerability's overall severity. 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 the Authio team, 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.

3. Findings

3.1 General

The Ethearnal team has undertaken a difficult responsibility in their attempt at creating a next-generation crowdsale. The resulting contracts were complicated and had a handful of conflicting interactions, which created a lot of potential for vulnerability. Several high severity issues were detected. However, the team was quick to respond to any concerns and has assuaged any idea of an insecure contract by fixing every issue found. The quality of the code itself is very good, although it lacks detailed commenting throughout the project. The codebase is clean and well-organized.

3.2 Contract Explanation

3.2.1 Function – Crowdsale

EthearnalRepTokenCrowdsale.sol: This contract houses the majority of the logic for the crowdsale itself. Users buy tokens by calling the fallback function, which calls the appropriate token purchase function. At the end of the crowdsale, an owner can call the finalizeByAdmin function to complete the crowdsale and unlock the tokens.

MultiOwnable.sol: The crowdsale contract inherits from MultiOwnable, which allows the contract to be set up with multiple different owners, which can whitelist users as well as finalize the crowdsale.

3.2.2 Function – Voting

VotingProxy.sol: This contract allows users to interact with the treasury, both to request a refund of the sale, and to allow the team to withdraw more funds. Motions for either of these are created using the startincreaseWithdrawalTeam and startRefundInvestorsBallot functions.

Ballot.sol and RefundInvestorsBallot.sol: These contracts are created by VotingProxy, and allow the token holders to vote on allowing the team to withdraw and request a refund of the sale respectively. They both inherit from *IBallot.sol*, which defines some the basic voting and decision functions.

3.2.3 Function – Fund Storage

Treasury.sol: This contract holds team funds and interacts with the voting contracts to release or return funds. Like the crowdsale contract, it inherits from MultiOwnable. It also serves as an interface through which the team can withdraw funds, as well as the token holders in event of a refund.

3.2.4 Function – Token

EthearnalRepToken.sol: This is the token itself. It inherits from MintableToken and LockableToken, and inherits all ERC20-compliant functions.

LockableToken.sol: This contract is inherited by the EthearnalRepToken. It inherits from OpenZeppelin’s Ownable and StandardToken contracts, and implements its own logic to allow ERT to be lockable and unlockable. It redefines both ERC20 transfer functions to track token movement, in order to block users from voting from more than one address.

3.3 Critical Severity

Refund Investor and Increase Team Withdrawals Blocking: This issue allows anyone to prevent the creation of a refund or team withdrawal ballot. If there is a refund ballot, the public startBallot function can be called to set votingActive to true. Because the withdraw team ballot creation function checks to ensure that there is no refund ballot with active voting, calling startBallot will make it impossible for a withdraw team ballot to be created. The same goes for the reverse – a refund ballot can be entirely blocked from being created by calling startBallot in the current withdraw team ballot.

This vulnerability has two potential ramifications. It can cause token holders to be unable to refund their investment, and it can cause the team to be entirely unable to withdraw funds. Because the startBallot function can be called by anyone, even non token-holders, this issue is high impact and high likelihood. As a result, it has been marked as critical severity.

This issue can be fixed by removing the startBallot function entirely, and simply setting the ballotStarted and votingActive variables on creation of the contract. **This issue was fixed in commit 323eb08.**

Token holders can single-handedly refund the crowdsale or increase team withdrawals: The votesByAddress mapping is never set for the sender, meaning a token holder can repeatedly call the vote function for any ballot, with each sequential call adding more votes to the total. This can allow any token holder to enact a total refund for the entire crowdsale. As this likelihood of a single token holder discovering this unset variable is high, and the impact of abusing this issue can refund the entire crowdsale, this issue has been marked as critical.

The issue was fixed in commit 7c2c2f4, with the addition of a line that set the votesByAddress mapping for the sender.

Token holders can transfer their tokens to another address and vote again: The lastMovement mapping is meant to track token transfers in order to prevent sending tokens to another address to vote again. However, for both the transfer and transferFrom functions, lastMovement was improperly set and allowed token holders to easily get around this feature.

This issue would make it possible for a token holder to have sole decision over any voting features, similar to the previous vulnerability mentioned. As such, it has also been marked critical.

The issue was fixed in the same commit, 7c2c2f4, with the addition of lines in transfer and transferFrom that correctly set lastMovement.

Crowdsale owners can set Ether rate in USD such that refunds are unable to be processed: This function was intended to set the rate of ETH in USD so that the tokens, priced in USD, could be exchanged for an exact amount of Ether. However, because the function can be accessed at any time and contains no checks for absurd values being set, a host of issues arise, as the rate of ETH in USD is an important variable in the contract.

The worst potential vulnerability allows the team to set the rate of Ether in USD so high that the refund investors function calculates the amount to refund to an investor as 0 every time. This can be enabled by any owner at any time, and has fairly wide-ranging consequences, so the issue has been marked as critical severity.

The recommended fix was to remove the setEtherRateUsd function entirely, which was done in commit 3477bf1.

3.4 High Severity

Crowdsale owners can potentially set fraudulent voting contract: Because the treasury is not required to be deployed with an already-set voting proxy contract, an owner could neglect to set a voting proxy contract until after the crowdsale is complete, at which point they could set a contract with additional functions that would allow for immediate and complete withdrawal of funds from the treasury, bypassing the voting feature entirely.

The Ethearnal team uses a deployment script that will ensure this is not the case at the time of deployment, by setting a voting proxy contract as soon as the treasury is deployed. However, the deployment script is out of the scope of this audit, as all safeguarding logic should be baked into an unalterable smart contract. As a result, this issue was marked high severity.

We recommended checking in the crowdsale constructor that the voting contract (and token contract) had been set in the treasury already. **The Ethearnal team agreed, and the issue was fixed in commit d60e2fc.**

3.5 Medium Severity

No medium severity issues were found.

3.6 Low Severity

Crowdsale can be started with pre-minted, unlocked, or un-mintable tokens: The address that deploys the token contract can call the mint or finishMinting functions before passing over ownership to the crowdsale contract. This could result in users investing in a project that has already been heavily diluted without their knowledge, but as the exploit would be readily apparent, it is unlikely many users would be affected.

This issue was fixed in commits d60e2fc, af6d68e, and 3e1765a, with the addition of checks in the crowdsale constructor which prevent the setting of a pre-minted, unlocked, or un-mintable token contract.

Treasury owners not set in Constructor: Treasury.sol is MultiOwnable, which means that multiple owners are set in an array for accessing onlyOwner functions. The owners are not set in the constructor like they are in the other MultiOwnable contract. It is possible, but unlikely, that the person deploying the Treasury contract could set up fraudulent owners at a later date without the team knowing. It is unlikely that the crowdsale would get very far without the team noticing, so this issue has been marked as low severity.

We recommend fixing this by setting the owners in the constructor.

Overflow risk: Especially with exponentiation, the risk for overflow here is particularly high. Use the SafeMath library to ensure there will be no overflow.

3.7 Notes & Recommendations

Efficiency concerns:

The method of creating a new ballot contract for every vote seems like a waste of gas and could probably be improved upon by creating and updating structs that represent a vote instead. This would also have the effect of reducing the overall complexity of the voting procedure as all of the voting logic would be located in the same contract. This would require significant refactoring, but would result in safer and more reusable code.

Many of the contract import statements could be simplified to import contract interfaces instead of complete contracts. With minimal refactoring, the deploy costs of the main contracts could be reduced significantly.

Ensure the ETH rate in USD is correctly updated before the crowdsale is started.

Upgrade to the latest version of solidity, and mark various functions as pure:

The functions that can be marked as pure are denoted as such in the commented code, supplied at the bottom of this document.

Create a notLocked modifier for the LockableToken contract:

‘require(!isLocked);’ is used three times. For readability and re-usability, moving this check to a modifier is recommended.

This line can be simplified to:

“return initialQuorumPercent < (weeksNumber * 10) ? 0 : initialQuorumPercent.sub(weeksNumber * 10);”.

Ensure valid addresses are supplied to VotingProxy: Add a check for address 0x0.

Add a check to these functions for invalid addresses:

getDaysPassedSinceLastRefundBallot

getDaysPassedSinceLastTeamFundsBallot

Mark functions as constant:

getOwners

isReadyToFinalize

Recommend implementing and using a validAddress modifier in Treasury.sol, as addresses will be checked multiple times for inequality with 0x0.

Superfluous require statement:

If the sender is the crowdsale contract, the address cannot be 0x0, so this check is redundant.

Make VotingProxy MultiOwnable:

If VotingProxy is meant to be team-run, it should be multi-ownable like the crowdsale and treasury.

Withdraw event will always be zero:

weiUnlocked and weiWithdrawed are set equal to each other on line 85, so the Withdraw event will always emit 0.

4. Documents & Resources

4.1 Line-By-Line Comments

Line-by-line commenting can be found in the EthereumAuthio github repository:

https://github.com/EthereumAuthio/Audits/tree/master/EthearnalCrowdsale

4.2 Project Code

The current Ethearnal crowdsale code can be found in their public github repository: https://github.com/Ethearnal/SmartContracts

5. Conclusion

The Authio team would like to recommend that the Ethearnal team now continues with this auditing and security process by posting public bug bounties and asking for community input. More eyes are always better, and after a rigorous audit, a bug bounty can serve as a valuable sanity check for all parties involved. It would be wise to include auditors in the consulting and building process for upcoming projects. When security is part of the whiteboarding process, critical issues can be caught before they become problems down the line.

In addition, it is worth noting that some of the issues mentioned in this report would most likely not have been issues during the actual crowdsale. The Ethearnal team had a deployment script ready that would have properly deployed their project in such a way that issues like "potentially setting a malicious voting contract" would be been dealt with early on in the deployment process. However, it is our view at Authio that such safety and logic should be hardcoded in the contracts themselves, as a deployment script is not unalterable, and could be changed in such a way that the contracts could be deployed maliciously. For this reason, we included only the smart contracts themselves in the scope of our audit.

Ethearnal has undertaking a difficult project with their vision for a next-generation ICO. Their code was complicated and had many contracts interacting to create a lot of potential for vulnerability. Generally, a restructuring of the code would have prevented a lot of these issues from cropping up in the first place. However, after scrupulous review, and working closely with the Ethearnal team, all issues of note have been fixed to the satisfaction of the Authio team.