Ethos Audit Report

Version 1.0

056Security

December 11, 2024

Protocol Audit Report

056Security

December, 2024

Prepared by: 056Security

Lead Auditors:

- 0xb0k0
- stanchev

Table of Contents

Protocol Summary

`Ethos` in its essence is a platform that strives to solve the problem of "social validation" and allow users to properly validate their identity and the identity of others. Users create their profiles, review other profiles and vouch for them. All of these actions increase their online reputation score and help them get more verified. The platform also tries to build a sort of a reputation market, where poeple can buy and sell votes based on the reputation score. `056Security` was hired from the `Ethos` team to help them organize and prepare the financial part of the protocol (i.e `EthosVault` (which was removed after as it had critical issues), `EthosVouch`, `ReputationMarket`) for their upcoming contest on one of the leading platforms. The whole process took about ~1 month.

Disclaimer

056Security makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts. We highly recommend that additional audits be done to ensure the security of the protocol.

Risk Classification

		Critical	High	Medium	Low
	Critical	C	C/H	H	M
	High	C/H	H	H/M	M
Likelihood	Medium	H	H/M	M	M/L
	Low	M	M	M/L	L

Audit Details

Scope

- Commit Hash: Various commits reaching `215e22a78da37631812acf929b45cd3726a3e374`

- In Scope:

```
./contracts/
- EthosVault.sol
- EthosVaultManager.sol
- EthosFees.sol
- EthosSlashPenalty.sol
- EthosVouch.sol
- ReputationMarket.sol
```

- Solc Version: 0.8.26
- Chain(s) to deploy contract to: Base
- Tokens:
  - ETH

Executive Summary

*The audit resulted in **four** `critical`, **two** `high`, **four** `medium`, **two** `low`, **four** `gas` and **two** `informational` findings.*

Issues found

Severity	Number of issues found
Critical	4
High	2
Medium	4
Low	2
Informational	2
Gas	4
Total	18

Findings

Critical

[C-01] User can deposit directly into the vault without providing ETH

Description: Currently, the vault's `deposit()` function is not set as `payable`, which means that the user can deposit assets to the vault without spending actual funds. When the internal `_deposit()` function is called, it will take the `WETH` deposit amount from the vault's balance, instead of the user's balance. What is more, the `msg.sender` will get minted shares, which will be redeemable. A malicious user can quickly drain the vault by depositing assets and then calling `withdraw()` to get the funds back.

```
  function deposit(uint256 assets, address receiver) public virtual
      override returns (uint256) {
    uint256 stakes = previewDeposit(assets);
    _deposit(msg.sender, receiver, assets, stakes);
    return stakes;
  }

function _deposit(
    address /caller/,
    address receiver,
    uint256 assets,
    uint256 stakes
  ) internal virtual {
    super._mint(receiver, stakes);
    IWETH9(asset()).deposit{ value: assets }();
    emit DepositETH(msg.sender, assets, stakes);
    _updateSlashPoints(stakes, receiver);
  }
```

Impact: Vault funds will be drained.

Mitigation: Make the `deposit()` function `payable` and check if the user has provided the correct amount of ETH.

[C-02] Profile with multiple addresses that vouches may end up with sutck funds for addresses that did not initiate the vouch

Description: If we have a profile that represents an organization with multiple addresses, a problem could arise with vouching where after one address unvouches, the other addresses that also vouch for the same profile will have their shares stuck, as the vouch won't be redeemable anymore.

Impact: Stuck funds for users.

Proof of concept:

Organization Ethos&056 has 3 addresses registered to the profile (Ben,Boris,Ivan)

1. Ben vouches for Alice with 1 eth(vault is created shares are distributed)
2. Boris increases the vouch for Alice with 1 eth(vault is created shares are distributed)
3. Ivan increases the vouc for Alice with 0.1 eth
4. Ivan decides to unvouch as he doesn't like Alice anymore
5. Ivan gets his 0.1eth back pays the fee
6. Now Ben and Boris have stuck funds as the vouch is no longer redeemable(archived)

Recommended mitigation: Keep track of all vouches for a profile and make sure that whenever there is an unvouch, it is applied per address.

[C-03] Malicious user could use the `EthosVault::withdraw()` function to drain the vault contract

Description: Due to improper checks in the `_withdraw()` function, a malicious user can drain the vault contract by withdrawing other users funds. Currently, the `_withdraw()` function does no checks if the `msg.sender` is actually the owner of the shares. This leads to the `msg.sender` being able to withdraw other users funds.

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner
  ) public virtual override returns (uint256) {
    uint256 stakes = previewWithdraw(assets);
    _withdraw(msg.sender, receiver, owner, assets, stakes);
    emit WithdrawETH(receiver, assets, stakes);
    return stakes;
  }

  function _withdraw(
    address caller, // vouch contract
    address receiver, // vouch contract
    address owner, // voucher
    uint256 assets, // amount of ETH to withdraw
    uint256 stakes // amount of stakes to burn
  ) internal virtual returns (uint256) {
  @>  require(caller == receiver, "Only the vouch contract can withdraw
     assets"); // @audit - both caller and receiver are inputs to the
    withdraw function
    // the owner holds N stakes of M assets
    // in order to "give back" their stakes, we burn them
    super._burn(owner, stakes);
    // in exchange, we retrieve the assets
    // first by converting from WETH to ETH
    IWETH9(asset()).withdraw(assets);
    // then sending the ETH to the receiver (in this case, the Vouch
       contract, so we can apply fees)
    (bool success, ) = payable(receiver).call{ value: assets }("");
    require(success, "Failed to send ETH");
    emit WithdrawETH(owner, assets, stakes);
    return assets;
}
```

Impact: Vault funds will be drained.

Mitigation: Add checks to make sure that the `msg.sender` is actually the owner of the shares or make the `withdraw()` function only callable by the vouch contract.

[C-04] Vault provides a `donate()` function which implies for the classical donation attack

Description: The `donate()` function is a function that allows users to donate ETH directly to the vault. This opens up the way for a classical donation attack, where a malicious user can deposit `wei` amounts and directly donate a large amount of ETH, making every other deposit worthless, as it will not mint any shares. This allows the donator to withdraw his/her `wei` amount of share, which will be worth all the ETH in the vault.

Impact: Users will not get any shares when depositing, and their actual funds will be drained.

Mitigation: Remove the `donate()` function and potentially re-think the design of the `EthosVault`
.

High

[H-01] Math rounding in `EthosVault` is not ERC4626-complicant: `previewWithdraw` should round ceil

Description: Quote from https://eips.ethereum.org/EIPS/eip-4626#security-considerations:

> Finally, EIP-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users: (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down. If (1) it's calculating the amount of shares a user has to supply to receive a given

> amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

This could be a little confusing, but in order to understand it fully you can open OZ's erc4626 implementation. There they have all the previewWitdraw, previewDeposit implementation and where they should round to.

Impact: Users will not get the correct amount of shares when withdrawing.

Mitigation: Add the following changes to the `previewWitdhraw` function:

```
function previewWitdraw(uint256 assets) public view virtual override
    returns (uint256) {
     return
       totalSupply() == 0
         ? assets
         : Math.mulDiv(
           assets, // New assets being added
           totalSupply(), // Current total stakes
           totalAssets(), // Current total assets
           Math.Rounding.Ceil
         );
    }
```

Or simplify the vault to look exactly like erc4626 OZ in order to have proper structure preview -> convertToAssets/Shares

[H-02] `EthosVault` is missing slippage protection on it's functions

Description: The current implementation of `ERC4626` does not include slippage protection mechanisms for deposit, mint, withdraw, and redeem operations as recommended by `EIP-4626`. This omission could lead to unexpected losses for users, especially EOAs interacting directly with the contract.

If implementors intend to support EOA account access directly, they should consider adding an additional function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved.

Impact: Users will not be able to withdraw the exact amount of ETH they want.

Mitigation: Add slippage protection in the following functions:

```
function depositWithSlippage(uint256 assets, address receiver, uint256
    minSharesOut) public returns (uint256 shares);
```

```
function mintWithSlippage(uint256 shares, address receiver, uint256
    maxAssetsIn) public returns (uint256 assets);
function withdrawWithSlippage(uint256 assets, address receiver, address
     owner, uint256 maxSharesBurned) public returns (uint256 shares);
function redeemWithSlippage(uint256 shares, address receiver, address
    owner, uint256 minAssetsOut) public returns (uint256 assets);
```

Medium

[M-01] The slippage check in `ReputationMarket::buyVotes` round incorrectly in certain cases

Description: The current logic for slippage checks in `ReputationMarket::buyVotes` does not work as expected. In certain cases, as when a user sets a slippage of 0.001%, if he/she wants to buy 8 votes, but due to the market conditions, he/she gets 7 votes, the transaction will revert, even though the slippage tolerance is extremely low. Due to Solidity's rounding down, even the smallest possible slippage will be rounded down.

```
  /**
   * @dev Checks if the actual votes bought are within range of the
       expected votes bought given a slippage tolerance in basis points.
   * @param actual The actual votes bought.
   * @param expected The expected votes bought.
   * @param slippageBasisPoints The slippage tolerance in basis points
       (1 basis point = 0.01%).
   */
  function _checkSlippageLimit(
    uint256 actual,
    uint256 expected,
    uint256 slippageBasisPoints
  ) private pure {
    uint256 toleratedVotes = (expected * ((SLIPPAGE_POINTS_BASE -
        slippageBasisPoints))) /
      SLIPPAGE_POINTS_BASE;
    if (actual < toleratedVotes) {
      revert SlippageLimitExceeded(actual, expected,
          slippageBasisPoints);
    }
  }
```

Impact: Reverted transactions due to incorrect slippage checks.

Recommended mitigation: Use special Math libraries for proper floor/ceil rounding.

[M-02] Using `transfer` and send for ETH transfers in Reputation Market

Description: When transferring ETH to recipients, if Solidity's `transfer()` or `send()` method is used, certain shortcomings arise, particularly when the recipient is a smart contract. These shortcomings can make it impossible to successfully transfer ETH to the smart contract recipient.

The following cases could lead to failed transactions:

1. The recipient does not implement a payable fallback function.
2. The recipient implements a payable fallback function which would incur more than 2300 gas units.
3. The recipient implements a payable fallback function incurring less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Impact: Reverted transactions.

Recommended mitigation: Use `call` with its returned boolean checked in combination with re-entrancy guard.

[M-03] There is no way of withdrawing any dust ether

Description: Since the Vault can receive native ETH, it is inevitable that a surplus may accumulate and remain stuck in the contract, as there is no way to withdraw it.

Impact: Stuck funds.

Mitigation: Add a proper way to withdraw stuck funds in the contract

[M-04] Chance of reentrancy in Ethos protocol

Description: The code contains multiple low level calls that could lead to reentrancy vulnerabilities.

Impact: Malicious user could reenter the contract and vouch malicously.

Recommended mitigation: Add `NonReentrant` modifier whenever there is a low level call.

Low

[L-01] `EthosVouch::updateMaximumVouches` uses >= instead of >

Description: The default value for `maximumVouches` is 256, so if the admin changes it to a different value, it won't be able to set it to 256 again.

```
function updateMaximumVouches(uint256 maximumVouches_) external
    onlyAdmin whenNotPaused {
@>  if (maximumVouches_ >= 256) { // @audit - should be > so that it
    can be set to 256
      revert MaximumVouchesExceeded(maximumVouches_, "Maximum vouches
          cannot exceed 256");
    }
    maximumVouches = maximumVouches_;
  }
```

Recommended mitigation: Use > instead of >=.

[L-02] Remove `mint()` as it is not used anywhere in the code

Description: While it is public and could be called by anyone without giving any weth,eth at stake. It will probably revert because _deposit will wrap 0 eth which is impossible.

Mitigation: Remove `mint()`.

Informational

[INFO-01] Use `shares` instead of `stakes` for more understandable naming

Mitigation: Change naming of stakes to shares everywhere as it is more understandable

[INFO-02] No need to apply modifier checks in the `EthosVaultManager::initialize()` function as `__accessControl_init` checks them

Description: The modifiers in the `EthosVaultManager::initialize()` function are redundant as `__accessControl_init` checks them.

```
function initialize(
    address _owner,
    address_admin,
    address _expectedSigner,
    address_signatureVerifier,
    address _contractAddressManagerAddr
  )
    public
    initializer
    validAddress(_owner)
    validAddress(_admin)
    validAddress(_expectedSigner)
    validAddress(_signatureVerifier)
    validAddress(_contractAddressManagerAddr)
{....}
```

Gas

[G-01] `EthosVouch::markUnhealthy(...)` has a duplicated `IEthosProfile(...).verifiedProfileIdForAddress(...)` call

The `markUnhealthy` function makes a call to `IEthosProfile(...).verifiedProfileIdForAddress(...)` twice, where the second call does utilize the returned data.

```
  function markUnhealthy(uint256 vouchId) public whenNotPaused {
    Vouch storage v = vouches[vouchId];
    uint256 profileId = IEthosProfile(
      contractAddressManager.getContractAddressForName(ETHOS_PROFILE)
    ).verifiedProfileIdForAddress(msg.sender);

    _vouchShouldExist(vouchId);
    _vouchShouldBePossibleUnhealthy(vouchId);
    _vouchShouldBelongToAuthor(vouchId, profileId);
@>  IEthosProfile(contractAddressManager.getContractAddressForName(
   ETHOS_PROFILE))
      .verifiedProfileIdForAddress(msg.sender); // @audit - duplicated
          redundant call

    v.unhealthy = true;
    // solhint-disable-next-line not-rely-on-time
    v.activityCheckpoints.unhealthyAt = block.timestamp;

    emit MarkedUnhealthy(v.vouchId, v.authorProfileId, v.
        subjectProfileId);
  }
```

[G-02] Store storage variables in local variables when used multiple times in a function

```
  function _rewardPreviousVouchers(
    uint256 amount,
    uint256 subjectProfileId
  ) internal returns (uint256 amountDistributed) {
    uint256[] storage vouchIds = vouchIdsForSubjectProfileId[
        subjectProfileId];

    // Calculate total balance of all active vouches
    uint256 totalBalance;
@>  for (uint256 i = 0; i < vouchIds.length; i++) { // @audit - store
   vouchIds.length in a local variable
```

```
      Vouch storage vouch = vouches[vouchIds[i]];
      // Only include active (not archived) vouches in the distribution
      if (!vouch.archived) {
        totalBalance += vouch.balance;
      }
    }

    // If this is the first voucher, do not distribute rewards
    if (totalBalance == 0) {
      return totalBalance;
    }

    // Distribute rewards proportionally
    uint256 remainingRewards = amount;
    for (uint256 i = 0; i < vouchIds.length && remainingRewards > 0; i++) {
      Vouch storage vouch = vouches[vouchIds[i]];
      if (!vouch.archived) {
        // Calculate this vouch's share of the rewards
        uint256 reward = amount.mulDiv(vouch.balance, totalBalance, Math.Rounding.Floor);
        if (reward > 0) {
          vouch.balance += reward;
          remainingRewards -= reward;
        }
      }
    }

    // Send any dust (remaining rewards due to rounding) to the subject reward escrow
    if (remainingRewards > 0) {
      _depositRewards(remainingRewards, subjectProfileId);
    }

    return amount;
  }
```

[GAS-03] Use errors instead of require statements as they consume less gas

Mitigation: Use errors instead of require statements as they consume less gas.

There are some errors that are not used:

```
error NotSupported();
error NotTransferable();
error OnlyVouchContract();
error OnlyVouchContract();
```

[GAS-04] Remove all occurrences of `payable` in `EthosVouch` as they are redundant

Description: `EthosVault(payable(vaultAddress))`, vault is working only with erc20's.

Mitigation: Remove all occurrences of `payable` in `EthosVouch`.