Etherium

Security review

Version 1.0

Table of Contents

1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $250,000. They have identified over 300 high and medium-severity vulnerabilities in both public contests and private audits.

2 About Etherium

Etherium is a non-profit ERC-20 token backed by ETH, featuring daily lotteries and auctions. Fees are accrued on minting, burning, and transferring, and these fees are used to reward lottery and auction winners.

The lottery runs daily, with each user's chance of winning being proportional to their token holdings.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	Etherium
Repository	Private
Commit hash	256aaade8bd6d8a8ebbc7bda3ae7afc24ce1fde1
Resolution	489747448c160b84aa659b4cc1dabb018d13ec7f
Documentation	In the repository
Methods	Manual review

Scope

src/Etherium.sol

Issues Found

Critical risk	0
High risk	4
Medium risk	1
Low risk	1
Informational	2

6 Findings

6.1 High risk

6.1.1 Finishing an auction will send `unclaimedPrizes` from `currentDay - 2` to public goods, or auction winner

Severity: *High risk*

Context: Etherium.sol#L877-L881

Description: `unclaimedPrizes` is shared variable for both auction winners and lottery winners. Both are executed daily, but `_executeLotteryInternal()` will write to `unclaimedPrizes` at index `currentDay - 1` and `_finalizeAuction()` will write at index `currentDay - 2` (because `currentAuction.auctionDay` is set to `currentDay - 1` at creation, but the distribution happens at `currentDay + 1`). This means that when we finalize auction, we will fetch index from `unclaimedPrizes` from `currentDay - 2`, which was filled with the lottery winner on the previous day. This will lead to the code interpreting that a prize from the last week has not been claimed and try to send it to public goods.

Impact:

Users that have not claimed their lottery from the previous day loose their tokens

Recommendation: Introduce two `unclaimedPrizes` variables:

- `lotteryUnclaimedPrizes`
- `auctionUnclaimedPrizes`

Resolution: Fixed in `5ff2586`

6.1.2 Using wrong ETH ratio for unclaimed prizes, leads to inflated ETH:Etherium ratio

Severity: *High risk*

Context: Etherium.sol#L634 Etherium.sol#L886

Description: The code incorrectly assumes that ETH & Etherium have a 1:1 ratio.

```
UnclaimedPrize storage prize = unclaimedPrizes[slot];
        if (prize.amount > 0) {
            // Try to send to public good
            address publicGood = PUBLIC_GOODS[currentPublicGoodIndex];
            currentPublicGoodIndex = uint8((currentPublicGoodIndex + 1) % PUBLIC_GOODS.
                length);

            (bool success,) = publicGood.call{value: prize.amount}("");

            if (success) {
                _burn(LOT_POOL, prize.amount);
                emit PublicGoodsFunded(publicGood, prize.amount, prize.winner);
            } else {
                // Add to current winner's prize
                currentAuction.etheriumAmount += uint112(prize.amount);
            }
        }
```

We can see that we attempt to send ETH as `prize.amount` which is in Ethereium, then we burn `prize.amount` which is correct.

`(bool success,)= publicGood.call{value: prize.amount}("");` this is incorrect as the contract assumes a 1:1 ratio between the two, which is not true, by default it's 1:1000, this leads to a severe loss of funds and broken tokenomics if the `call` succeeds.

Impact: Inflated ETH:Etherium ratio, broken tokenomics

Recommendation: Use the redeem formula when handling the unclaimed prizes for both ETH and Etherium

Resolution: Fixed in `1cc456a`

6.1.3 PepeUSD has 6 decimals instead of 18

Severity: *High risk*

Context: Etherium.sol#L35

Description: The code assumes that PepeUSD has 18 decimals, but in fact it has 6 decimals. This breaks the accounting around `mintFeeFree` as it's attempting to transfer 100e18 PepeUSD, which is more than the `totalSupply` of the token.

Impact: `mintFeeFree` becomes unusable

Recommendation: Use 6 decimal precision

Resolution: Fixed in `8e67d55`

6.1.4 Smart contracts are excluded from lottery invariant breaks if smart contract mints in constructor

Severity: *High risk*

Context: Etherium.sol#L487-L488

Description:

The contract exclusion mechanism `account.code.length > 0` fails during constructor execution, allowing smart contracts to bypass the exclusion and corrupt the Fenwick tree used for lottery selection.

During constructor execution, `msg.sender.code.length = 0` because the contract hasn't finished deploying yet.

```
// Line 571 in _updateCumulativeBalancesWithExplicitBalances
if (account.code.length > 0) return; // Skip contracts
```

Vulnerability Flow:

1. Deploy malicious contract that mints ETHERIUM in construct
2. Constructor bypasses exclusion -> Contract added to Fenwick tree
3. Transfer/burn tokens -> Exclusion check now works, but Fenwick tree not updated
4. Phantom entries remain -> Broken lottery accounting - double counting the same tokens for the lottery

Impact:

- Unfair Winner Selection -> Lottery probabilities are corrupted
- Accounting Inconsistency -> Fenwick tree doesn't match actual balances

Recommendation: If the address is already in the Fenwich tree, don't return

Resolution: Fixed in `ae17191`

6.2 Medium risk

6.2.1 Sandwich attack on `mint`

Severity: *Medium risk*

Context: Etherium.sol#L193-L194

Description: `mint` function initially mints at ratio 1:1000. When `mintingEndTime` passes, we use a pricing formula to maintatin the ETH backing ratio:

```
            uint256 ethBalance = address(this).balance - msg.value; // Exclude sent ETH
            if (totalSupply() > 0 && ethBalance > 0) {
                // Mint proportionally to maintain ETH backing ratio
                etheriumToMint = (msg.value * totalSupply()) / ethBalance;
```

The code directly uses `address(this).balance` value, which can be influenced by any user donation. Under specific circumstances, the mechainc can be weaponized, so the resulting `etheriumToMint` rounds down to 0 and exploiter stealing victims underlying eth. An attack in style first depositor may be executed under the following conditions: 1. The minting period has ended (`block.timestamp > mintingEndTime`) 2. There is a small `totalSupply()` (ideally 1 token) - Everybody has redeemed and there are no funds in `FEES_POOL` & `LOT_POOL`

Recommendation: Consider implementing `minAmountOut` var, which may be set to a 5% below the value calculated by `eth_call` to `mint` from the FE

Resolution: Fixed in `f55a93e` by enforcing `etheriumToMint >= 100`

6.3 Low risk

6.3.1 Fees roll over will skip one day

Severity: *Low risk*

Context: Etherium.sol#L868-L869

Description: If there is no bidder for the auction for day X, the corresponding amount will be auctioned again in day X + 2, instead of X + 1.

That's the case because we first fetch the fees for X + 1 without those allocated for `X` and then we transfer back the amount from `LOT_POOL` to `FEES_POOL`:

1. `function _tryExecuteLotteryAndAuction()`
2. `uint256 feesToDistribute = balanceOf(FEES_POOL);` (without the missed day)
3. `_startAuction` (feesToDistribute - the fees for the missed day)
4. `_finalizeAuction`

```
            if (currentAuction.etheriumAmount > 0) {
                // Add unclaimed auction amount back to fees pool for next distribution
                _atomicUpdate(
                    LOT_POOL,
                    FEES_POOL,
                    currentAuction.etheriumAmount
                );
            }
            return;
```

Impact: Fees distributions is delayed

Recommendation: Consider overriding `feesToDistribute` in `_finalizeAuction`, if the should be rolled over

Resolution: Acknowledged

6.4 Informational risk

6.4.1 `_checkAndSetMaxSupply` in `mintFeeFree` can be removed

Severity: *Informational risk*

The snippet is useless because we enforce `block.timestamp <= mintingEndTime` The same for

```
require(
                totalSupply() + etheriumToMint <= maxSupplyEver,
                "Max supply reached"
            );
```

Resolution: Fixed

6.4.2 `block.prevrandao` is not a good source of randomness and block proposers can affect it.

Severity: *Informational risk*

As there is no impact in the current scope, it is a good to know info

---

Resolution: Acknowledged - “The proposer can skip a block if they don’t wish to publish prevrandao (last reveal attack) but that also forsakes their block fees (penalty). In addition they only affect 1 bit of information (skip current prevrandao or not). We also host daily prizes for half the daily fees, meaning the rewards should not grow too large.”