# Eden Security Audit

Report Version 1.0

November 23, 2024

Conducted by **Hunter Security**

# Table of Contents

# 1 About Hunter Security

Hunter Security is an industry-leading smart contract security auditing firm. Having conducted over 100 security audits protecting over $1B of TVL, our team always strives to deliver top-notch security services to the best DeFi protocols. For security audit inquiries, you can reach out on Telegram or Twitter at *@georgehntr*.

# 2 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 reveal the presence of vulnerabilities **but cannot guarantee their absence**.

# 3 Risk classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | High | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## 3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

## 3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

## 3.3 Actions required by severity level

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

# 4 Executive summary

Hunter Security was engaged by Eden to review their smart contract protocol. 2 separate teams of senior auditors conducted an audit aggregating all identified vulnerabilities below.

**Overview**

| | |
|---|---|
| Project Name | Eden |
| Repository | https://github.com/De-centraX/eden-contracts |
| Commit hash | c54dc1191f2e3ef74638737a4cb209bcb91c8e4e |
| Resolution | 015914fb2c58e64363273bc49bfe26a5cc61572c |
| Methods | Manual review & testing |

**Scope**

| |
|---|
| src/actions/SwapActions.sol |
| src/pools/EdenBloom.sol |
| src/Airdrop.sol |
| src/BuyAndBurn.sol |
| src/Eden.sol |
| src/Migrator.sol |
| src/Mining.sol |
| src/Staking.sol |

**Issues Found**

| | |
|---|---|
| High risk | 1 |
| Medium risk | 1 |
| Low risk | 4 |

# 5 Findings

## 5.1 High

### 5.1.1 Distribution logic could be distorted

**Severity:** High

**Context:** Staking.sol

**Description:** The *updateRewardsIfNecessary* function calls *_getCurrentDay* to calculate the time passed and distribute rewards:

```
function _getCurrentDay() internal view returns (uint32 currentDay) {
    currentDay = Time.dayGap(startTimestamp, Time.blockTs()) + 1;
}
```

The problem here arises if the *startTimestamp* (set in the constructor) is in the future:

- *_getCurrentDay* uses *Time.dayGap* which uses assembly, meaning overflow is possible in case *start* > *end*
- in case *Time.blockTs* < *startTimestamp*, *_getCurrentDay* would return a huge arbitrary value, which will get saved in *lastDistributedDay*

After *startTimestamp* has passed distribution will practically not happen, because *currentDay* will always be smaller than *lastDistributedDay*.

Currently, there is nothing preventing the user from calling *updateRewardsIfNecessary* before the *startTimestamp*.

**Recommendation:** Inside *updateRewardsIfNecessary*, validate that *startTimestamp* is not in the future. Also, consider implementing a check in *_getCurrentDay* that the *startTimestamp* is not > *Time.blockTs*.

**Resolution:** Resolved.

## 5.2 Medium

### 5.2.1 EdenBloom could be permanently DoS-ed affecting multiple other contracts

**Severity:** Medium

**Context:** EdenBloom.sol

**Description:** The logic of picking winners in the *EdenBloom* contract is the following:

1. *pickWinner* is called to request randomness from Chainlink, which also generates a new *WinnerRequest* with the *fulfilled* property set to *false*:

```
requests[requestId] = WinnerRequest({
    fulfilled: false,
    upForGrabs: upForGrabs
});
```

2. The *fulfilled* property is important since it is being used by the *noPendingRandomness* modifier to prevent new randomness requests, before the pending one has been resolved:

```
require(
    lastRequestId == 0 || _lastReq.fulfilled,
    RandomnessAlreadyRequested()
);
```

The problem is that in case the *fulfillRandomWords* call fails or does not happen (due to insufficient callback gas limit, lack of funds in the subscription account, etc.) the *_lastReq.fulfilled* will never get updated to *true* (since Chainlink invokes *fulfillRandomWords()* only once).

And since there is no mechanism to unstuck such a request, the contract will be permanently DoS-ed, since the modifier is attached to the following functions:

- *pickWinner*
- *participate*
- *removeParticipant*

As result - no randomness requests can be made, but the more severe consequence is that *participate* and *removeParticipant* will revert meaning that *Staking.stake* and *Staking.unstake* will also get DoS-ed - locking user funds and disturbing normal operation of the *Staking* contract.

**Recommendation:** Consider implementing an additional mechanism, it could be permissioned or callable by the participants after some time delay that allows a request to be unstuck. For example an additional flag could be added to *WinnerRequest* to allow invalidating them.

**Resolution:** Resolved.

## 5.3 Low

### 5.3.1 Compounding staking rewards does not account for duration bonus

**Severity:** Low

**Context:** Staking.sol

**Description:** The function *compoundRewards* allows the user to add their rewards to their staking directly. However, it does not add a bonus amount, unlike the *stake* function, which calls *convertToEdenShares* that adds a bonus amount of shares using the *_Erank()* function.

Therefore, it will be more profitable for the user to withdraw their rewards and then stake again in order to get a better amount of shares.

**Recommendation:** Consider modifying the logic to account for the bonus amount of shares.

**Resolution:** Acknowledged. Intended behavior.

### 5.3.2 Staking & unstaking is temporarily DoS-ed during pending requests inside EdenBloom

**Severity:** Low

**Context:** Staking.sol

**Description:** Since the word size in the EVM is 32 bytes, using unsigned integers with a width less than 256 bits, in function parameters and body, can cause more gas consumption. Furthermore, down casting is not covered by solidity overflow protection, meaning that overflows may silently happen.

**Recommendation:** One potential solution might be to refactor the logic in *EdenBloom* so that the randomness request snapshots the participant list in the *WinnerSelected* struct and uses it later to pick the winner. This would allow removing the *noPendingRandomness* modifier from the participant functions

If blocking participants from unstaking is desired, then at least consider checking that the user trying to unstake is a participant.

**Resolution:** Acknowledged. Part of the design of the system. The proposed mitigation does not work as caching the participant list may cause OOG exception leading to permanent DoS.

### 5.3.3 Lack of slippage protection

**Severity:** Low

**Context:** Eden.sol, Mining.sol

**Description:** The *_createUniswapV3Pool* function of *Eden* uses the UniswapV3 quoter contract to deduce the price that would be set for the pool.

Using the quoter contract is susceptible to manipulation and should not be used on-chain (something Uniswap warns about as well - https://docs.uniswap.org/contracts/v3/reference/periphery/lens/Quoter).

Furthermore, the liquidity provided when minting the position in the *Mining* contract once *INITIAL_TITAN_X_FOR_LIQ* has been collected, is provided at a ratio calculated using the same values as when the pool was created with its initial price. However, the price may have changed significantly during the time gap between these 2 actions, both in the EDEN:VOLT pool and TITANX:VOLT pool.

**Recommendation:** Consider calculating the price off-chain and then providing it as a parameter directly to the *_createUniswapV3Pool* function. Also, consider the similar possible vector in *Mining.addLiquidityToVoltEdenPool*.

**Resolution:** Acknowledged.

### 5.3.4 Use of different unsigned integer width

**Severity:** Low

**Context:** src/*

**Description:** Since the word size in the EVM is 32 bytes, using unsigned integers with a width less than 256 bits, in function parameters and body, can cause more gas consumption. Furthermore, down casting is not covered by solidity overflow protection, meaning that overflows may silently happen.

**Recommendation:** We couldn't identify any severe issues arising from the use of smaller integer types. However, we strongly recommend using only *uint256* and *int256* to prevent the risk of silent overflow or exceptions, causing unexpected behavior and potential DoS.

**Resolution:** Acknowledged.

## 5.4 Informational

### 5.4.1 Typographical, non-critical or centralisation issues

**Severity:** Informational

**Context:** src/*

**Description:** The contracts contain one or more typographical mistakes, non-critical issues or centralisation vulnerabilities. In an effort to keep the report size reasonable, we enumerate these below:

1. Consider using OpenZeppelin's Ownable2Step instead of Ownable.
2. Users would not be able to get claim their Eden tokens after the maturity timestamp of a miner has passed if the *mining* contract address was changed in Eden. Consider removing this setter.
3. Stakers in the EdenStaking contract would not receive any rewards if the *staking* contract address is changed in Eden. Consider removing this setter.
4. Potential Denial-of-Service if the owner of the Eden contract sets the *buyAndBurn* address to the null address or any interface-incompatible contract.
5. Consider enforcing an upper limit for the lookback period when set in *changeSlippageConfig* (i.e. 30 minutes).
6. Consider using UniswapV3's factory *getPool* method instead of computing it via the *PoolAddress* library.
7. When the *minSharesToBloom* value is updated in the Staking contract, the new value would not be effective immediately, but will only apply for new stakes and unstakes. This means that even if the minimum shares were increased or decreased, stakers having passed the old requirement would still be eligible for the EdenBloomPool prize.
8. There is no cut taken for the pool when rewards are added for distribution for a single pool via *distributeToSpecificPool*.
9. The default value of the *MinerStatus* enum is *CLAIMED* which would be returned even for non-existent miners. Consider adding a new element at index 0 such as *INVALID*.
10. Several state variables across all contracts do not have explicit visibility specified meaning it would default to *private*. Consider explicitly defining the visibility where missing.
11. No upper or lower limit in *changeLpSlippage* allowing the owner to set a slippage higher than 100% or equal to 0%.
12. Consider adding the following check to *startMiningLadder*:

```
require((ladderEnd - ladderStart) % ladderIntervals == 0);
```

13. *startMiningLadder* may revert due to OOG exception if the first loop does too many iterations. Consider limiting all iterations in total to 100, instead of just for the inner loop.
14. The calculations in *minerCost* would lead to overflow in ~10 years which could cause unexpected behaviour. Consider working only with *uint256* variables type and enforcing a maximum limit on the *timeOfCreation* variable.
15. In *_distribute*, consider skipping transfers when the fee amount is 0.
16. In *collectFees*, the 2 transfers may be omitted by simply specifying the *recipient* property as the *FEES_WALLET* instead of *address(this)*.
17. In *addLiquidityToVoltEdenPool*, if any VOLT amount was not taken by the pool to add as liquidity, the leftover would be locked forever in the Mining contract. Consider sending the remaining VOLT balance back to the owner if there's any leftover due to slippage.

18. The owner of the *EdenBloomPool* can change the coordinator address and provide malicious randomness so that they steal the funds.
19. The *admin* variable is never changed, hence it could be marked as *immutable* or a setter method should be implemented allowing to transfer ownership. Consider inheriting *Ownable2Step*.
20. The *startTimestamp* variable in *EdenBloomPool* is never used.
21. The *subscriptionId* cannot be changed. Consider whether implementing a setter method would be useful.
22. *changeRequestConfirmations* should not allow setting the parameter to a value lower than 3 as per Chainlink's documentation for Ethereum. An upper limit should also be enforced to prevent DoS.
23. As per Chainlink's documentation, *fulfillRandomWords* must never revert. However, in *EdenBloomPool* it could revert if *lastIntervalCall* > *block.timestamp* which would happens when the passed to the constructor *_startTimestamp* is in the future.
24. If no one has called *pickWinner* for more than 1 *INTERVAL_TIME*, the accumulated rewards for 2 intervals will be distributed to just 1 winner, and the winners from the other intervals would receive nothing.
25. Consider making the *callbackGasLimit* property passed to the Chainlink coordinator configurable to prevent potential DoS if the amount of gas has not been estimated properly.

**Recommendation:** Consider fixing the above typographical mistakes, non-critical issues and code-style suggestions.

**Resolution:** Partially resolved.