hexens × LIDO

# LIDO V2 ORACLE SECURITY REVIEW

# CONTENTS

# ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading web3 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Coinstats, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

# AUDIT LED BY


## HAYK ANDRIASYAN

Senior Security Researcher | Hexens


---

Audit Starting Date
24.04.2023

Audit Completion Date
01.05.2023

---

hexens × LIDO

# METHODOLOGY

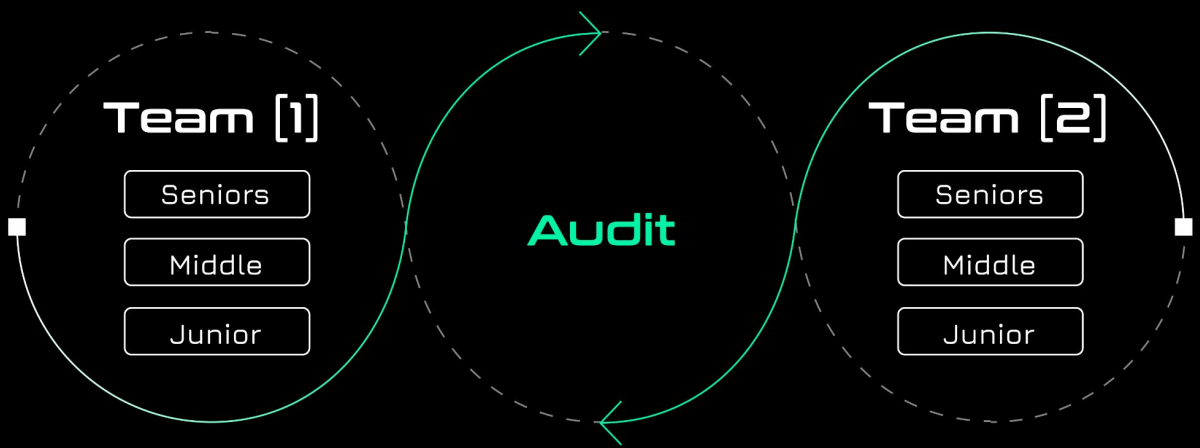
## COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

## HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



# SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

| IMPACT | PROBABILITY | | | |
|---|---|---|---|---|
| | Rare | Unlikely | Likely | Very Likely |
| Low / Info | Low / Info | Low / Info | Medium | Medium |
| Medium | Low / Info | Medium | Medium | High |
| High | Medium | Medium | High | Critical |
| Critical | Medium | High | Critical | Critical |

## SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

### CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

## HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

## MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

## LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

## INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of smart contracts. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

# EXECUTIVE SUMMARY

## OVERVIEW

This audit covered the latest release version of the off-chain part of Lido V.2.

Our security assessment was a Lido V.2 Oracle review aiming to validate the latest release commits. We have thoroughly reviewed each latest changes individually and the system as a whole.

All of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality has increased after the completion of our audit.

# SCOPE

The analyzed resources are located on:
https://github.com/lidofinance/lido-oracle/commit/e50088b0cc51d3ae8954f5651348fb1405bdf61f

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

The final version with all implemented fixes is located on:
https://github.com/lidofinance/lido-oracle/commit/44678954915b8291c949904c63de5e4e4983b427

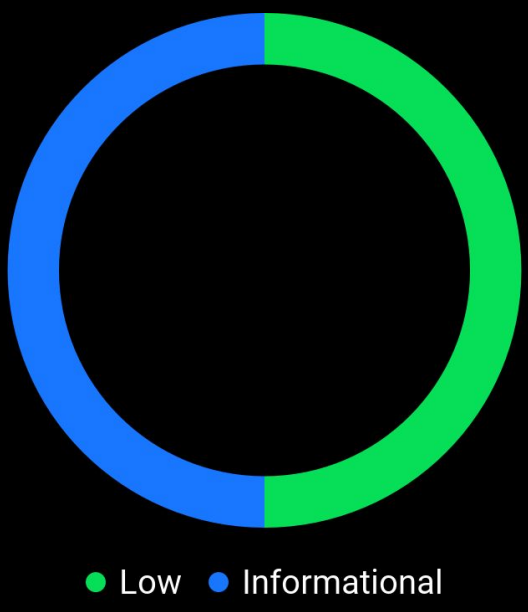
We confirm that the docker image located on:
https://hub.docker.com/layers/lidofinance/oracle/3.0.0/images/sha256-d2ee5ecc78f8b991fcd2327e1d1bc84b8015aa7b8fde73e5ec0e702e6bec6c86?context=explore was built on Lido backend's corresponding commit:
44678954915b8291c949904c63de5e4e4983b427

# SUMMARY

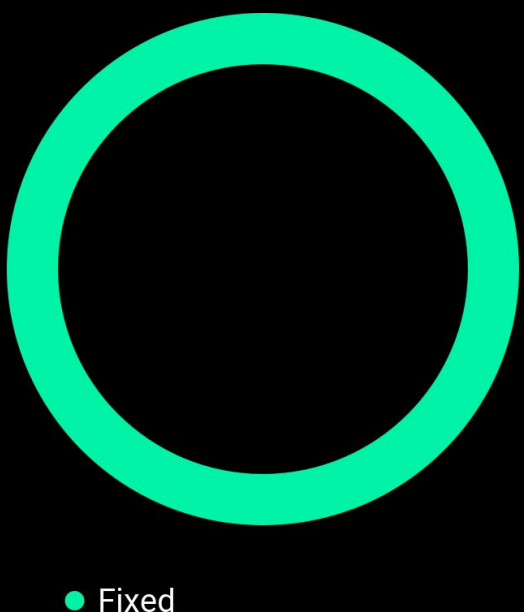
| SEVERITY | NUMBER OF FINDINGS |
| --- | --- |
| CRITICAL | 0 |
| HIGH | 0 |
| MEDIUM | 0 |
| LOW | 1 |
| INFORMATIONAL | 1 |

**TOTAL: 2**

## SEVERITY




## STATUS



# WEAKNESSES

This section contains the list of discovered weaknesses.

## LID-28. WRONG TEST

SEVERITY: Low

PATH: test_prediction.py

REMEDIATION: change test to expect an exception to be thrown

STATUS: fixed

DESCRIPTION:

In prediction **test test_get_rewards_no_matching_events** expects to get 0 wei reward. Test mocks 2 events with one transaction data in each event. Mocks are

```
web3.lido_contracts.lido.events.ETHDistributed.get_logs.return_value = [
  {'transactionHash': HexBytes('0x123'), 'args': {'name': 'first', 'reportTimestamp': 1675441508}},
]
web3.lido_contracts.lido.events.TokenRebased.get_logs.return_value = [
  {'transactionHash': HexBytes('0x456'), 'args': {'value': 2, 'reportTimestamp': 1675441508}},
]
```

and there is no matching transaction hash in this events.

**rewards = p.get_rewards_per_epoch(bp, cc)** this methods calls inside **events = self._group_events_by_transaction_hash(token_rebase_events, eth_distributed_events)** and then checks

```
if not events:
 return Wei(0)
```

As there is no matching transaction hash in mock events: **_group_events_by_transaction_hash** method will throw an exception so the test will fail as it expects to get 0.

# LID-27. CODE OPTIMISATION

SEVERITY: Informational

PATH: prediction.py

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In **_group_events_by_transaction_hash** method there is a check whether transactions hashes from the first event type exists in the second and then appends event's arguments in **result_event_data**. This check is done in the worst case O(n^2) complexity via nested loops. It can be optimised to worst case O(n) complexity.

```
for event_1 in event_type_1:
    for event_2 in event_type_2:
        if event_2['transactionHash'] == event_1['transactionHash']:
            result_event_data.append({
                **event_1['args'],
                **event_2['args'],
            })
            break
```

At first event_2's transaction hashes can be loaded in a dictionary. Then check event_1's transaction hashes exists in that dictionary or not.

```
cache = dict()

for event_2 in event_type_2:
  cache[event_2['transactionHash']] = event_2

for event_1 in event_type_1:
  if event_1['transactionHash'] in cache:
    result_event_data.append({
      **event_1['args'],
      **cache[event_1['transactionHash']]['args'],
    })
```