Security Review Report for Chaos Labs

April 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covered the base contracts of Chaos Agents (also called Edge Agents) implementation by Chaos and BGD Labs. These agents work as middleware between Chaos's Risk Oracles and the target DeFi protocol.

Our security assessment was a full review of the smart contracts in scope, spanning a total of 1 week.

During our audit, we did not identify any major vulnerabilities.

We did identify several minor severity vulnerabilities and code optimisations.

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

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

3. Security Review Details

- **Review Led by**

Kasper Zwijsen, Head of Audits

- **Scope**

We initially reviewed an internal repository, but can confirm the final fixed version is located at:

🔗 https://github.com/ChaosLabsInc/chaos-agents/tree/

📌 **Commit:** `e7e566f6601d53b797e7de4bd8e2210da31584a0`

- **Changelog**

24 April 2025	Audit Start
05 May 2025	Initial Report
08 May 2025	Revision Received
12 May2025	Final Report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	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 that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.
High	Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

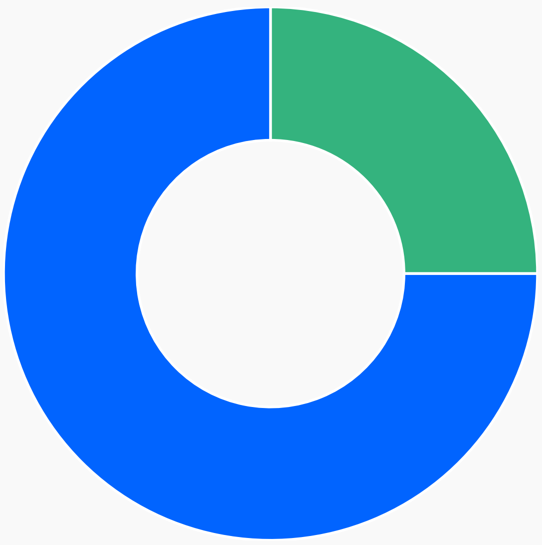
- ## Issue Symbolic Codes

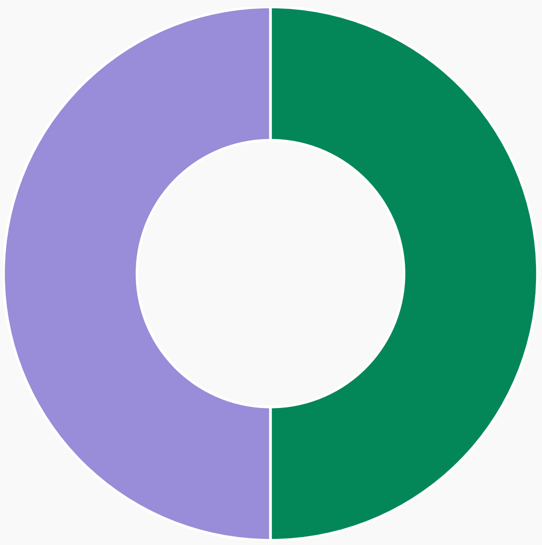
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
🟥 Critical	0
🟥 High	0
🟧 Medium	0
🟩 Low	1
🟦 Informational	3
Total:	**4**





6. Weaknesses

This section contains the list of discovered weaknesses.

BGDL1-2 | Unprotected Implementation Contract Initialization

Fixed ✓

Severity:	Low	Probability:	Rare	Impact:	Medium

Path:

`src/contracts/EdgeAgentHub.sol`

Description:

The **EdgeAgentHub** implementation contract can be directly initialized, allowing an attacker to become its owner, potentially allowing them to execute privileged functions on the implementation contract itself.

```
contract EdgeAgentHub is EdgeConfigurator, IEdgeAgentHub {
  using EnumerableSet for EnumerableSet.AddressSet;

  function initialize(address edgeOwner) external initializer {
    __EdgeConfigurator_init(edgeOwner);
  }
```

Remediation:

Implement initialization protection using one of these approaches

- Disable initializers in the constructor:

```
constructor() {
  _disableInitializers();
}
```

- Add a proxy-only check

```
function initialize(address edgeOwner) external initializer {
  require(address(this) != _getImplementation(), "Direct initialization not
allowed");
  __EdgeConfigurator_init(edgeOwner);
}
```

BGDL1-4 | Agent can DoS check function and automation using maximum batch limit

Acknowledged

Severity:	Informational	Probability:	Unlikely	Impact:	Low

Path:

`EdgeAgentHub.sol:check`

Description:

The EdgeAgentHub exposes a view function **check** for the automated off-chain component to calculate the **ActionData** from a list of agent IDs. This action data is later used to execute.

The function uses the global maxBatchSize to limit the amount of inject actions across all agents and the function will silently stop if the maximum has been reached. In such a case, it'll simply return the actions up to the maximum.

This mechanism can be used by a single agent to exploit others. Since the markets are fetched using **_getAgentMarkets**, which dynamically fetches the markets from the agent address, it can be arbitrarily increased and it can force the maximum batch size to be reached.

For example, if the number of markets is greater than the maximum batch size, the **check** function will never reach the next agent. The **check** function only accepts agent IDs as parameters and so it cannot skip to the middle of a market list of an agent where it left off.

For the hypothetical off-chain mechanism there are 2 cases if the agent has more markets than the limit:

- It thinks the agent ID was not fully processed because of the break, so it will include the same agent ID again, blocking it for all other agents.
- It thinks the agent ID was processed because it was included in the parameters, and so it skips over the rest of the markets. This can be abused by an agent, if the break is forced to happen during a victim agent ID, forcing a skip of those markets.

If we look at ChainlinkEdgeAgentHub.sol then it would probably be the 2nd case.

```
  function check(
    uint256[] memory agentIds
  ) public view virtual returns (bool, ActionData[] memory) {
    ActionData[] memory actionData = new ActionData[](agentIds.length);
    uint256 actionCount;

    EdgeHubStorage storage $ = _getStorage();
    uint256 maxBatchSize = $.maxBatchSize;
    uint256 batchSize; // total number of updates across all agents

    for (uint256 i = 0; i < agentIds.length; i++) {
      uint256 agentId = agentIds[i];

      AgentConfig storage config = $.config[agentId];
      BasicConfig memory basicConfig = config.basicConfig;

      if (!basicConfig.isAgentEnabled) continue;

      address[] memory markets = _getAgentMarkets(
        config,
        basicConfig.isMarketsFromAgentEnabled,
        basicConfig.agentAddress,
        agentId
      );
      if (markets.length == 0) continue;

      IRiskOracle riskOracle = IRiskOracle(basicConfig.riskOracle);
      string memory updateType = config.updateType;

      address[] memory marketsToUpdate = new address[](markets.length);
      uint256 marketsToUpdateCount;
      for (uint256 j = 0; j < markets.length; j++) {
        // The Risk Oracle is expected to revert if we query a non-existing
update.
        // In that case, we simply skip the market
        try riskOracle.getLatestUpdateByParameterAndMarket(updateType,
markets[j]) returns (
          IRiskOracle.RiskParameterUpdate memory updateRiskParams
        ) {
          if (_validateBasics(config, basicConfig, agentId, updateRiskParams))
{
            marketsToUpdate[marketsToUpdateCount++] = updateRiskParams.market;
            batchSize++;
          }
        } catch {}

        // stop collecting data if we reached max batch size, to protect
against gas overflow on execution
        if (maxBatchSize != 0 && batchSize == maxBatchSize) break;
      }

      if (marketsToUpdateCount != 0) {
        assembly {
          mstore(marketsToUpdate, marketsToUpdateCount)
        }
        actionData[actionCount].agentId = agentId;
        actionData[actionCount].markets = marketsToUpdate;
        actionCount++;
      }

      // stop collecting data if we reached max batch size, to protect against
gas overflow on execution
      if (maxBatchSize != 0 && batchSize == maxBatchSize) break;
    }

    assembly {
      mstore(actionData, actionCount)
    }

    return (actionCount != 0, actionData);
  }
```

Remediation:

The **check** function should allow for picking up where it left off with regard to a list of markets, although this might complicate automation further. An easier approach would be to break earlier, if **batchSize + markets.length > maxBatchSize** and not fill it all the way to **maxBatchSize**. That way you only process full agents and the malicious agent can be more easily left out by the off-chain component.

Commentary from the client:

" We assume that every Hub will be controlled by one entity, i.e., one hub for Aave and another for GMX. So, in this case, it's hard to consider the configuration of agents as a potential exploit."

BGDL1-5 | Optimisation of _getAgentMarkets by returning early

Fixed ✓

Severity:	Informational	Probability:	Unlikely	Impact:	Low

Path:

EdgeAgentHub.sol:_getAgentMarkets

Description:

The function **_getAgentMarkets** fetches the agent's list of markets and filter it against the **restrictedMarkets** list by copying everything into another array.

```
    function _getAgentMarkets(
      AgentConfig storage config,
      bool isMarketsFromAgentEnabled,
      address agentAddress,
      uint256 agentId
    ) internal view returns (address[] memory) {
      // Case we fetch only allowed markets configured on the Hub for this Agent
      if (!isMarketsFromAgentEnabled) {
        return config.allowedMarkets.values();
      }

      // Otherwise we fetch all markets from the Agent and apply the configured
  restricted markets
      address[] memory markets = IBaseAgent(agentAddress).getMarkets(agentId);

      uint256 validMarketCount;
      address[] memory validMarkets = new address[](markets.length);
      for (uint256 i = 0; i < markets.length; i++) {
        if (!config.restrictedMarkets.contains(markets[i])) {
          validMarkets[validMarketCount++] = markets[i];
        }
      }
      assembly {
        mstore(validMarkets, validMarketCount)
      }

      return validMarkets;
    }
```

Remediation:

We recommend to return the full list early if the **restrictedMarkets** list is empty and avoid creating an identical array.

For example:

```
if (config.restrictedMarkets.length() == 0) return markets;
```

BGDL1-6 | EdgeConfigurator does not validate uniqueness of agent address

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Medium

Path:

EdgeConfigurator.sol:registerAgent, setAgentAddress

Description:

The EdgeConfigurator allows for registering an agent with an arbitrary agent address. It also exposes a configuration function to modify the agent address of an existing agent. Both functions can only be called by the owner.

Nonetheless, there are no safeguards on whether the provided agent address is unique and not already in use by another agent. If a duplicate agent address were to be submitted (e.g. mistakenly or because of automation), it could result in one agent pivoting into another victim agent, as the agent trusts the call of the EdgeAgentHub. By only allowing the address to be used by one agent at a time, this would be prevented.

Moreover, the current Agent implementations do not validate the parameter **agentId** or **updateType** in **_processUpdate**, so it would not know if another agent with another ID has forced a call into it.

```
  function setAgentAddress(uint256 agentId, address agentAddress) public
onlyOwner {
    _getStorage().config[agentId].basicConfig.agentAddress = agentAddress;
    emit AgentAddressSet(agentId, agentAddress);
  }
```

Remediation:

We recommend to keep a mapping of **agentAddress => bool** and verify whether the address has been used already when registering or modifying an agent's address.

Commentary from the client:

" It is the design decision, we imagine cases when one agent contract can be "reused" by a different agentId, and may support multiple updateTypes."

hexens x CHAOS LABS x BORED GHOSTS DEVELOPING