hexens x GLIF

Security Review Report for GLIF

September 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 covers the security review for GLIF StICNT. This review covered the updates to the code in comparison to the scope of the previous engagement.

Our security assessment was a full review of the new code, spanning a total of 3 days.

During our review, we did not identify any major severity vulnerabilities.

We did identify several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team.

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

Jahyun Koo, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/glif-confidential/sticnt/tree/74ba119caa45540bed132f2d2d0b0b8d5561e8d7/src

The issues described in this report were fixed in the following commit:

🔗 https://github.com/glif-confidential/sticnt/tree/1b4fc70c6c80f41e97c5e89170647f044990a365/src

▪ Changelog

15 September 2025	Audit start
22 September 2025	Initial report
13 October 2025	Revision received
15 October 2025	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.

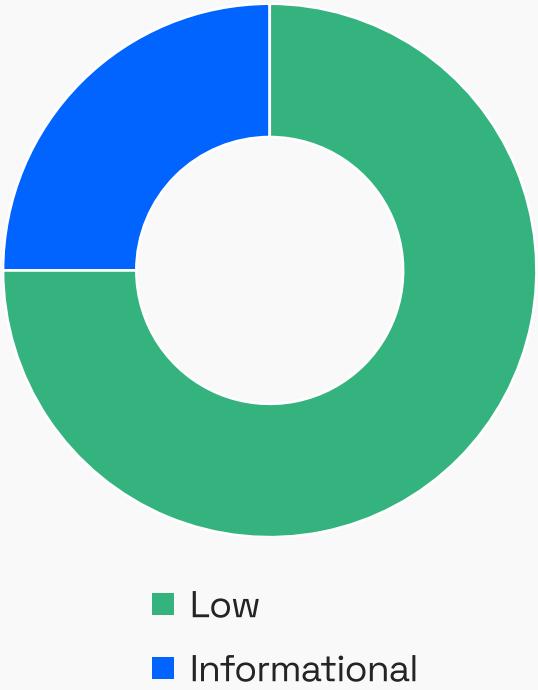
- **Issue Symbolic Codes**

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	3
Informational	1
Total:	**4**



6. Weaknesses

This section contains the list of discovered weaknesses.

GLIF4-1 | Incorrect Calculation for Delegatable Assets in shouldFlushDeposits

Fixed

Severity:	Low	Probability:	Rare	Impact:	Low

Path:

src/Periphery/Periphery.sol
src/Pool/PoolV2.sol

Description:

The **shouldFlushDeposits** function in **Periphery.sol** incorrectly calculates the amount of assets available for delegation. It compares the pool's total asset balance directly against **minPortfolioDelegationSize** without first subtracting the **exitReserve**. This could cause the function to return **true** even when the actual delegatable funds are insufficient, potentially misleading off-chain systems that rely on this view function to trigger transactions.

```
    function shouldFlushDeposits() external view virtual returns (bool) {
        PeripheryStorage storage $ = _getStorage();
        IPoolV2 pool_ = $.pool;

        uint256 balance = IERC20(pool_.asset()).balanceOf(address(pool_));
        uint256 exitReserve_ = pool_.exitReserve();

        // the balance should be above the exit reserve
        if (balance <= exitReserve_) {
            return false;
        }

        uint256 availableToDelegate = balance - exitReserve_;

        // check if enough time has passed since last delegation
        bool overTimeThrottle = block.timestamp >= pool_.lastTimeStampCheckpoint() +
pool_.delegationThrottleSeconds();
```

```
        // if the available amount is over the max delegation size, we can bypass the time throttle
        bool shouldByPassThrottle = availableToDelegate >= pool_.maxPortfolioDelegationSize();

        return availableToDelegate >= pool_.minPortfolioDelegationSize() && (overTimeThrottle ||
shouldByPassThrottle);
    }
```

```
    function flushDeposits() external virtual whenNotPaused {
        ...
        uint256 availableToDelegate = balance - exitReserve_;
        if (availableToDelegate >= $.minPortfolioDelegationSize) {
            uint256 maxPortfolioDelegationSize_ = $.maxPortfolioDelegationSize;
            // check if enough time has passed since last delegation
            bool overTimeThrottle = block.timestamp >= $._lastTimeStampCheckpoint +
$.delegationThrottleSeconds;
            // if the balance is over the max delegation size, we can bypass the time throttle
            bool shouldByPassThrottle = availableToDelegate >= maxPortfolioDelegationSize_;
            if (overTimeThrottle || shouldByPassThrottle) {
                $._lastTimeStampCheckpoint = block.timestamp;

                _delegatePortfolio($, availableToDelegate, maxPortfolioDelegationSize_);
            }
        }
```

Remediation:

Update the function to first calculate the **availableToDelegate** amount by subtracting **pool_.exitReserve()** from the total balance.

GLIF4-4 | getScalarNodeMap() uses wrong array size for nodeInfos

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/Periphery/Periphery.sol#L254-L272

Description:

The **getScalarNodeMap()** function creates the **nodeInfos** array using the last element of **nodeIDs** (nodeIDs[nodeIDs.length - 1]) as the length. If the node IDs are non-sequential or have gaps (e.g., [4,5,8,10]), this results in uninitialized or incorrect entries and wrong poolDelegated values. A safe approach is to allocate the array with **nodeIDs.length** and map each nodeId to its correct index.

Remediation:

Instead of using **nodeIDs[nodeIDs.length - 1]** to set the length of nodeInfos, allocate the array with **nodeIDs.length** to ensure it exactly matches the number of nodes:

```
-- NodeInfo[] memory nodeInfos = new NodeInfo[](nodeIDs[nodeIDs.length - 1]);

++ NodeInfo[] memory nodeInfos = new NodeInfo[](nodeIDs.length);
```

Replace any indexing like NodeInfo memory **nodeInfo = nodeInfos[nodeId - 1];** with a different implementation, for example a loop that searches for the matching nodeId:

```
uint256 nodeId = delegations[i].nodeDelegations[j].nodeId;
uint256 amount = delegations[i].nodeDelegations[j].amount;

 for (uint256 k = 0; k < nodeInfos.length; k++) {
    if (nodeInfos[k].nodeId == nodeId) {
       nodeInfos[k].poolDelegated += amount;
    }
 }
```

GLIF4-3 | Incorrect loop termination in getActiveNodeListFromICN

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/Periphery/Periphery.sol#L300-L329

Description:

The function **getActiveNodeListFromICN** increments **nodeId** using **activeCount + 1**.
Since **activeCount** only increases for active nodes, the loop misses valid node IDs if gaps exist in the sequence. Additionally, it uses **break** when encountering an inactive node, which prevents discovery of higher node IDs.

For example, if node statuses are:

- 1 → Active
- 2 → Active
- 3 → Inactive
- 4 → Active

The function would stop at node **3** and fail to include node **4** or higher.

Remediation:

Use an independent **nodeId** counter and only **break** when encountering an **InvalidScalerNode** error. Increment **nodeId** on every iteration, not just for active nodes.

```
    function getActiveNodeListFromICN() external view virtual returns (uint256[] memory) {
        PeripheryStorage storage $ = _getStorage();
        IICNProtocolMini icnp_ = $.icnp;


        uint256[] memory tempList = new uint256[]($.maxNodeIdCount); // Temporary array with reasonable max
size
        uint256 activeCount = 0;
++      uint256 nodeId = 1;

        while (true) {
--          uint256 nodeId = activeCount + 1;
            try icnp_.getProtocolScalerNodeStatus(nodeId) returns (IICNProtocolMini.ProtocolScalerNodeStatus
status) {
```

```
            if (status == IICNProtocolMini.ProtocolScalerNodeStatus.Active) {
                tempList[activeCount] = nodeId;
                activeCount++;
--              } else {
--                  break;
--              }


++.             nodeId++;

        } catch (bytes memory lowLevelData) {
            // Check if it's the InvalidScalerNode error
            if (lowLevelData.length == 4 && bytes4(lowLevelData) ==
IICNProtocolMini.InvalidScalerNode.selector) {
                break; // Stop after finding an invalid node
            } else {
                // If it's a different error, rethrow it
                assembly {
                    revert(add(lowLevelData, 32), mload(lowLevelData))
                }
            }
        }
    }

    // Create properly sized array and copy values
    uint256[] memory activeNodeList = new uint256[](activeCount);
    for (uint256 i = 0; i < activeCount; i++) {
        activeNodeList[i] = tempList[i];
    }

    return activeNodeList;
  }
```

GLIF4-2 | Gas Optimization: In-place Array Resizing

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/Periphery/Periphery.sol#L300-L329

Description:

The **getActiveNodeListFromICN** function used a temporary, fixed-size array to store results, then created a new, correctly-sized array and copied the elements. This process of allocating a second array and looping to copy data resulted in unnecessary gas consumption.

```
function getActiveNodeListFromICN() external view virtual returns (uint256[] memory) {
    ...
    // Create properly sized array and copy values
    uint256[] memory activeNodeList = new uint256[](activeCount);
    for (uint256 i = 0; i < activeCount; i++) {
        activeNodeList[i] = tempList[i];
    }

    return activeNodeList;
}
```

Remediation:

Modify to resize the temporary array in-place using an assembly block.

```
// Resizing the array in place to save gas
assembly {
    mstore(tempList, activeCount)
}
return tempList;
```


hexens
x
GLIF