hexens x GLIF

Security Review Report for GLIF

August 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 Plus, an NFT that provides GLIF card holders with benefits, such as cashback on their FIL.

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

During our review, we identified 2 high severity vulnerabilities, which could have resulted in unauthorized changes and miscalculations in cashback amounts.

We have also identified 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

The analyzed resources are located on:

🔗 https://github.com/glifio/plus/tree/14ff30c78307861e47d107a600c640d1808f91b0

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

🔗 https://github.com/glifio/plus/

📌 Commit: `aae2b94eddb4a0b8539a3782cf638d1aea6b8017`

▪ Changelog

25 August 2025	Audit start
01 September 2025	Initial report
03 September 2025	Revision received
04 September 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.

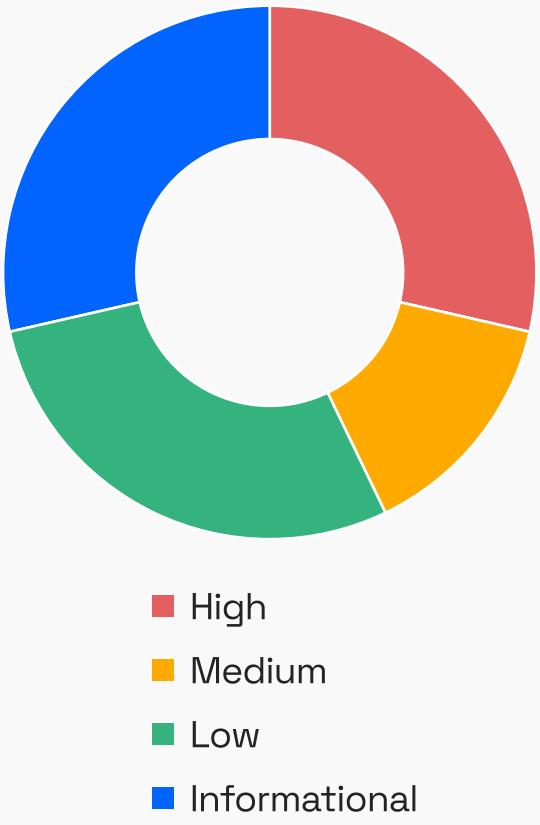
- **Issue Symbolic Codes**

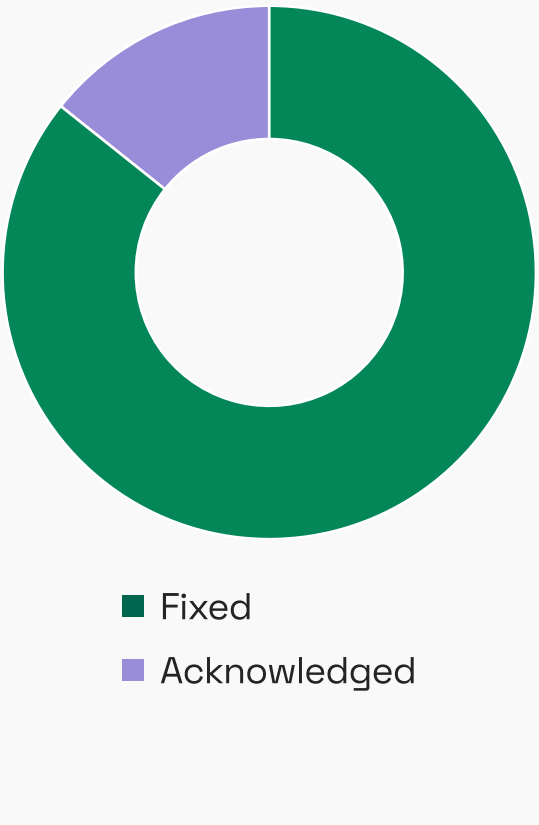
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	2
Medium	1
Low	2
Informational	2
Total:	**7**





6. Weaknesses

This section contains the list of discovered weaknesses.

GLIF3-1 | Personal cashback percentage can be changed by anyone

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/Plus.sol:fundGlfVault#L702-L733

Description:

Using the function **setPersonalCashBackPercent** only the token owner is allowed to change the cashback percentage of their token ID.

However, in **fundGlfVault** it allows anyone to send tokens to any tokenId's vault. When doing this, they can also set the cashback percentage, even if they don't own that token.

For example, an attacker can send **1 wei** with **_cashBackPercent = X..%**, and this will overwrite the cashback setting for another token ID.

As a result, the token owner may be prevented from earning any FIL cashback on the next **onPaymentMade** call.

```
    function fundGlfVault(uint256 _tokenId, uint256 _amount, uint256
_cashBackPercent) public whenNotPaused {
        require(_amount != 0, ZeroAmount());


        PlusStorage storage $ = _getStorage();
        $.glfToken.transferFrom(msg.sender, address(this), _amount);


        $.tokenIdToGlfVaultBalance[_tokenId] += _amount;


        uint256 currentCashBackPercent =
$.tokenIdToPersonalCashBackPercent[_tokenId];
        uint256 vaultBalance = $.tokenIdToGlfVaultBalance[_tokenId];
```

```
        // used for event logging
        uint256 newCashBackPercent;
        if (_cashBackPercent == 0) {
            // if this is the first time funding the vault, set the cashback to
the default (max)
            // note we imply if this is the first time funding the vault, if
the vaultBalance equals the amount we just put in, and the current cashback is
0
            if (vaultBalance == _amount && currentCashBackPercent == 0) {
                newCashBackPercent = $.maxCashBackPercent;
                _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
            } else if (currentCashBackPercent > 0) {
                // otherwise this is not the first time setting the vault,
there's an existing cashback, don't override
                newCashBackPercent = 0;
                _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
            }
        } else {
            // otherwise just set the cashback to the provided value
            newCashBackPercent = _cashBackPercent;
            _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
        }


        emit GlfVaultFunded(_tokenId, msg.sender, _amount, newCashBackPercent);
    }
```

Remediation:

Add an ownership check inside **fundGlfVault**:

- If the caller is the token owner, allow **_cashBackPercent** changes
- If the caller is not the token owner, only allow funding (without changing cashback)

GLIF3-2 | Incorrect GLF/FIL price calculation in onPaymentMade

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/Plus.sol:onPaymentMade#L763-L815

Description:

Each tier card has a **cashBackPremium** value that determines the premium price for the card's owner. If the tier card is Gold, the **cashBackPremium** is set by default to 1.1, meaning a 10% premium for GLF. This allows the user to receive more FIL when cashing back with the same amount of GLF tokens.

According to the documentation, if the current price of FIL to GLF is 0.004, the Gold card owner receives a premium price of 0.0044 FIL per GLF. Therefore, when they cash back, they can exchange 1 GLF for 0.0044 FIL. This means fewer GLF tokens are needed to receive the same amount of FIL compared to the base rate.

However, in the **onPaymentMade** function, the **glfNeeded** value (in GLF tokens) is calculated by multiplying the **cashbackAmount** (in FIL) by the **conversionRateWithPremium** (the premium price of GLF).

```
uint256 cashbackAmount = (_interestAmount * cashbackPercent) /
BASIS_POINT_DENOMINATOR;

// First apply the tier bonus
uint256 filToGlf = $.baseConversionRateFILtoGLF;
uint256 conversionRateWithPremium =
filToGlf.rawMulWad(tierInfo.cashBackPremium);
// Then convert FIL to GLF using the computed rate, rounded up to favor the
protocol
uint256 glfNeeded = cashbackAmount.rawMulWadUp(conversionRateWithPremium);
```

This is incorrect because **baseConversionRateFILtoGLF** represents the price of a GLF token, so **conversionRateWithPremium** is the premium price of GLF. To calculate the amount of GLF tokens exchanged from FIL, **conversionRateWithPremium** should be divided by the amount of FIL tokens, not multiplied.

Example from docs:
Total interest payment: 1500 FIL

- Portion of interest available for cash back (5% of payment): 75 FIL
- $GLF 30 Day TWAP: 0.004 $FIL per $GLF
- $GLF Protocol Pricing (10% premium): 0.0044 $FIL per $GLF
- Amount of $GLF spent for cash back: 17045.45 $GLF

Based on the calculation in the code:

- baseConversionRateFILtoGLF = 0.004
- conversionRateWithPremium = 0.004 * 1.1 = 0.0044
- glfNeeded = 75 * 0.0044 = 0.33 $GLIF

```
    function onPaymentMade(uint256 _agentId, uint256 _interestAmount) external
onlyPool {
        if (paused()) return;
        if (_interestAmount == 0) return;


        // Find the token associated with this agent
        PlusStorage storage $ = _getStorage();
        uint256 tokenId = $.agentIdToTokenId[_agentId];
        if (tokenId == 0) return; // No card associated with this agent


        if (!isTokenActive(tokenId)) return;


        uint256 cashbackPercent = $.tokenIdToPersonalCashBackPercent[tokenId];
        if (cashbackPercent == 0) return; // nothing to process


        Tier tier = $.tokenIdToTier[tokenId];
        TierInfo memory tierInfo = $.tierToTierInfo[tier];


        // Example:
        // p.s 10000 = 100%
        // Rounded down. Even though rounding up increases the amount of GLF
needed,
        // it would also mean paying more cash back.
        uint256 cashbackAmount = (_interestAmount * cashbackPercent) /
BASIS_POINT_DENOMINATOR;
```

```
        // First apply the tier bonus
        uint256 filToGlf = $.baseConversionRateFILtoGLF;
        uint256 conversionRateWithPremium =
filToGlf.rawMulWad(tierInfo.cashBackPremium);
        // Then convert FIL to GLF using the computed rate, rounded up to favor
the protocol
        uint256 glfNeeded =
cashbackAmount.rawMulWadUp(conversionRateWithPremium);


        uint256 glfVaultBalance = $.tokenIdToGlfVaultBalance[tokenId];
        if (glfVaultBalance < glfNeeded) {
            glfNeeded = glfVaultBalance;
            cashbackAmount = glfNeeded.rawDivWad(conversionRateWithPremium);
        }


        uint256 filBalance = $.totalFilCashbackVaultBalance;
        if (filBalance < cashbackAmount) {
            cashbackAmount = filBalance;
            glfNeeded = cashbackAmount.rawMulWadUp(conversionRateWithPremium);
        }


        if (glfNeeded == 0) return; // No cashback to process


        $.tokenIdToGlfVaultBalance[tokenId] -= glfNeeded;


        $.totalFilCashbackVaultBalance -= cashbackAmount;


        $.tokenIdToFilCashbackEarned[tokenId] += cashbackAmount;


        $.glfToken.transfer($.treasury, glfNeeded);


        emit PaymentProcessed(_agentId, tokenId, _interestAmount, glfNeeded,
cashbackAmount);
    }
```

Remediation:

Replace `cashbackAmount.rawMulWadUp` with `cashbackAmount.rawDivWadUp` to calculate `glfNeeded`.

GLIF3-7 | Activate/Upgrade does not check agent leverage position health

Acknowledged

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

src/Plus.sol:activate, upgrade

Description:

The system has default leverage limits (debtToLiquidationValue) per tier:

- Bronze = 75%
- Silver = 85%
- Gold = 88%

If a user is maxed out at 88% while in Gold Tier and tries to downgrade to Bronze, it fails because **validateAgentLeverage()** checks that their current leverage doesn't fit the lower Bronze limit.

A agent with 88% DTL could activate a bronze card (max 75% DTL), because it never checks if its in a healthy position with that **debtToLiquidationValue.**

Another example:

1. Suppose a user is on Bronze with 75% leverage (at the old max).
2. The admin changes the limits to Bronze = 50%, Silver = 60%, Gold = 70%.
3. the user can upgrade to Silver, even though Silver's new limit is only 60%.

So the system is letting users bypass the leverage restriction by upgrading or activating, even when their leverage is higher than the cap.

```
    function activate(address _beneficiary, uint256 _tokenId, Tier _tier)
        public
        whenNotPaused
        senderIsTokenOwner(_tokenId)
        tierIsActive(_tier)
    {
        address cardOwner = msg.sender;
        address beneficiary = _beneficiary.normalize();


        require(!isTokenActive(_tokenId), AlreadyActive(_tokenId));
```

```
        PlusStorage storage $ = _getStorage();
        uint256 requiredGlf = $.tierToTierInfo[_tier].tokenLockAmount;
        $.glfToken.transferFrom(cardOwner, address(this), requiredGlf);


        $.tokenIdToLastTierSwitchTimestamp[_tokenId] = block.timestamp;
        $.tokenIdToTier[_tokenId] = _tier;
        $.tokenIdToTierLockAmount[_tokenId] = requiredGlf;


        if (beneficiary == address(0)) {
            emit CardActivated(_tokenId, cardOwner, _tier);
        } else {
            uint256 agentId = agentIDByAddress(beneficiary);
            require(IAgent(beneficiary).owner() == cardOwner,
BeneficiaryOwnerIsNotCardOwner(beneficiary, cardOwner));
            uint256 agentTokenId = $.agentIdToTokenId[agentId];
            require(agentTokenId == 0, AgentAlreadyHasToken(agentId,
agentTokenId));


            $.agentIdToTokenId[agentId] = _tokenId;
            $.tokenIdToAgentId[_tokenId] = agentId;


            emit CardActivated(_tokenId, beneficiary, _tier);
        }
    }
```

Remediation:

Ensure that **validateAgentLeverage()** is used for all tier changes, including upgrade, and activation. This guarantees a user's current leverage never exceeds the target tier's cap.

Commentary from the client:

"The agent police still contains a DTL check, so if we run into issues where the Card holder is over leveraged even when upgrading, the leverage enforcement in the agent police (and in our oracle for that matter) will still stop the user from doing anything actually dangerous in the system."

GLIF3-3 | The fundGlfVault function should check whether the token ID is active

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/Plus.sol#L702-L733

Description:

There is no check to validate whether the card NFT **tokenId** is active in the **fundGlfVault()** function. Therefore, users can still fund an inactive card, but they will not be able to cash back in the **onPaymentMade()** function.

This puts the funds of the inactive card at risk of being frozen until the user activates it and receives the cashback.

```
function fundGlfVault(uint256 _tokenId, uint256 _amount, uint256
_cashBackPercent) public whenNotPaused {
    require(_amount != 0, ZeroAmount());

    PlusStorage storage $ = _getStorage();
    $.glfToken.transferFrom(msg.sender, address(this), _amount);

    $.tokenIdToGlfVaultBalance[_tokenId] += _amount;

    uint256 currentCashBackPercent =
$.tokenIdToPersonalCashBackPercent[_tokenId];
    uint256 vaultBalance = $.tokenIdToGlfVaultBalance[_tokenId];

    // used for event logging
    uint256 newCashBackPercent;
    if (_cashBackPercent == 0) {
        // if this is the first time funding the vault, set the cashback to the
default (max)
        // note we imply if this is the first time funding the vault, if the
vaultBalance equals the amount we just put in, and the current cashback is 0
        if (vaultBalance == _amount && currentCashBackPercent == 0) {
            newCashBackPercent = $.maxCashBackPercent;
            _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
        } else if (currentCashBackPercent > 0) {
```

```
            // otherwise this is not the first time setting the vault, there's
an existing cashback, don't override
            newCashBackPercent = 0;
            _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
        }
    } else {
        // otherwise just set the cashback to the provided value
        newCashBackPercent = _cashBackPercent;
        _setPersonalCashBackPercent(_tokenId, newCashBackPercent);
    }

    emit GlfVaultFunded(_tokenId, msg.sender, _amount, newCashBackPercent);
}
```

Remediation:

fundGlfVault() should include a check to ensure that only active cards can receive GLF funds.

GLIF3-4 | The mintAndActivate function only works when the sender is the recipient

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/Plus.sol:mintAndActivate#L259-L263

Description:

In **mintAndActivate()**, users can specify the **_to** address as the recipient of the minted card. However, if **_to** is different from **msg.sender**, the NFT card will not be minted to the caller. As a result, the **activate()** function will revert because it also triggers the **senderIsTokenOwner()** modifier.

Therefore, only **msg.sender** can be the recipient of the minted card token in the **mintAndActivate()** function.

The same issue arises in the **mintActivateAndFund()** function.

```
function mintAndActivate(address _to, address _beneficiary, Tier _tier) public
returns (uint256 tokenId) {
    tokenId = mint(_to);
    activate(_beneficiary, tokenId, _tier);
    return tokenId;
}
```

Remediation:

The **_to** address parameter should be removed from the **mintAndActivate()** and **mintActivateAndFund()** functions. The NFT token should be minted to **msg.sender** instead.

GLIF3-5 | Missing events for penalties and withdraw

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Low

Path:

src/Plus.sol:downgrade#L343-L399

Description:

In the function **downgrade**, the user can downgrade their token to a lower tier. If the lower tier would result in a refund of GLF tokens for the user, a potential penalty could apply if the downgrade happened too quickly after a recent switch.

Currently the code does not emit events for the penalty and the resulting withdrawal amount, making off-chain tracking more difficult.

```
        uint256 currentGlf = $.tokenIdToTierLockAmount[_tokenId];
        uint256 desiredGlf = desiredTierInfo.tokenLockAmount;
        {
            uint256 timeSinceLastSwitch = block.timestamp -
$.tokenIdToLastTierSwitchTimestamp[_tokenId];
            IERC20 glf = $.glfToken;

            // The admin may change the price after the tokens are locked...
            // In the case of the lower tier requiring more tokens, we void the
fee
            if (currentGlf < desiredGlf) {
                uint256 additionalGlf = desiredGlf - currentGlf;
                glf.transferFrom(cardOwner, address(this), additionalGlf);
            } else if (currentGlf > desiredGlf) {
                uint256 withdrawGlf = currentGlf - desiredGlf;
                // check if penalty should be applied
                if (timeSinceLastSwitch < $.tierSwitchPenaltyWindow) {
                    // apply the penalty
                    uint256 penaltyAmount = withdrawGlf *
$.tierSwitchPenaltyFee / BASIS_POINT_DENOMINATOR;
                    glf.transfer($.treasury, penaltyAmount);
                    glf.transfer(cardOwner, withdrawGlf - penaltyAmount);
                } else {
                    glf.transfer(cardOwner, withdrawGlf);
                }
```

```
            }
        }
```

Remediation:

Consider adding an event to emit the penalty amount.

GLIF3-6 | Mixed use of mint and safe mint

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Low

Path:

src/Plus.sol:mint, changeOwnerForAgent#L193-L201, #L450-L464

Description:

Currently the **mint** function uses the internal **_mint** to mint the NFT to the recipient, this does not have a safety check. On the other hand, the **changeOwnerForAgent** function that transfers the NFT to the new owner uses the internal **_safeTransfer** function, which would have a safety check.

```
    function mint(address _to) public whenNotPaused returns (uint256 tokenId) {
        PlusStorage storage $ = _getStorage();
        address to = _to.normalize();
        $.glfToken.transferFrom(msg.sender, $.treasury, $.mintPrice);
        tokenId = $.tokenIdGenerator++;
        _mint(to, tokenId);
        emit CardMinted(tokenId, msg.sender, to);
        return tokenId;
    }
```

```
    function changeOwnerForAgent(address _agent) external whenNotPaused {
        address agent = _agent.normalize();
        require(agent != address(0), ZeroAddress());
        PlusStorage storage $ = _getStorage();
        require($.agentFactory.isAgent(agent), BeneficiaryIsNotAnAgent(agent));
        address newOwner = IAgent(agent).owner();
        require(newOwner != address(0), ZeroAddress());
        // Note: ID zero would fail with the check above (factory forbids ID
0).
        uint256 agentId = IAgent(agent).id();
        uint256 tokenId = $.agentIdToTokenId[agentId];
        // Note: OZ's ownerOf already checks that oldOwner != address(0).
        address oldOwner = ownerOf(tokenId);
        require(oldOwner != newOwner, SameOwner());
        _safeTransfer(oldOwner, newOwner, tokenId);
    }
```

Remediation:

Consider the user of mint and safe mint for both functions and whether a safety check is needed for both or neither. Since the CEI pattern wouldn't be violated in this contract, we recommend to use **_safeMint** in **mint** as well. Caution should be taken when having other contracts call Plus:mint.


hexens
x
GLIF