hexens x GLIF

Security Review Report for GLIF

November 2025

Table of Contents

1. About Hexens
2. Executive summary
3. Security Review Details
    - Security Review Lead
    - Scope
    - Changelog
4. Severity Structure
    - Severity characteristics
    - Issue symbolic codes
5. Findings Summary
6. Weaknesses
    - Incorrect Calculation of stakingTotalYbt When Setting Window Root
    - Incorrect Use of _windowStart as _windowId in _binarySearchWindowId Call
    - The FIL reserves of the LpPlus contract may be insufficient to execute RWTFuture tokens
    - Full removal of YBT or RWT can be DoSed
    - Inconsistent DEFAULT_FUTURE_EXPIRATION_PERIOD
    - Missing address normalisation
    - Missing events for LpPlus configuration state changes
    - Redundant balance check in withdrawFilFunds
    - LpPlus mint emits unnormalised address parameter
    - Missing events for RWTFuture configuration state changes
    - Redundant balance check in _executeBatch
    - Redundant length checks in _claimBatch
    - Expired RWTFuture NFTs can never be burned

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. This review included the new LpPlus and RWTFuture contracts, that allow holders to stake and get beneficial strike prices on their RWT to FIL conversions.

Our security assessment was a full review of the code, spanning a total of 2 weeks.

During our review, we did identified 2 High severity vulnerabilities, which could have resulted in incorrect distribution of FIL among the token holders.

We 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:

First week:
https://github.com/glifio/plus/tree/9f7c95f3ab48c926a01087fecb9a3cf0505aa5c4

Second week:
https://github.com/glifio/plus/tree/da0b62f611deed8ccf0f035761a3ad3344eae6b1

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

https://github.com/glifio/plus/tree/84d63653688d6d47102913d0d76de813f8ad516a

Changelog

17 November 2025	Audit start
2 December 2025	Initial report
5 December 2025	Revision received
10 December 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.

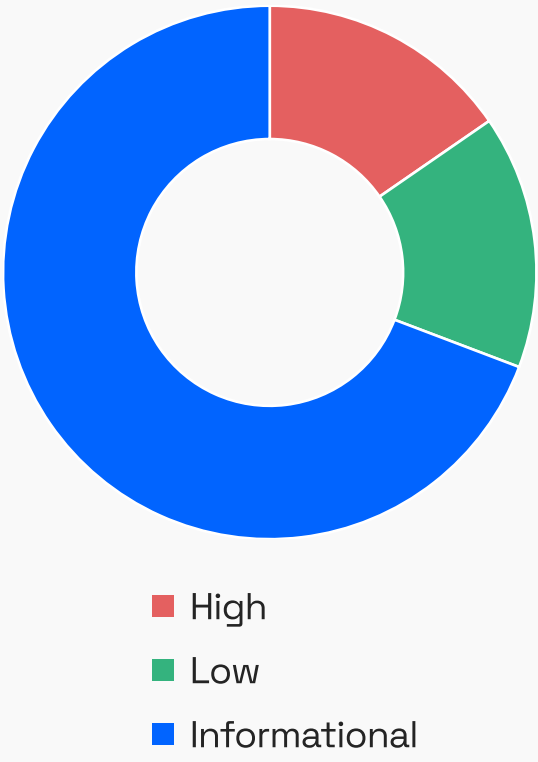
- **Issue Symbolic Codes**

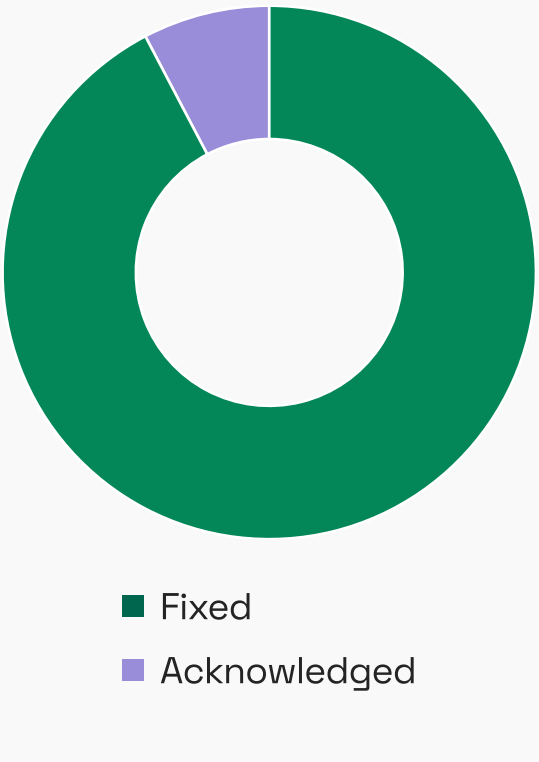
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	0
Low	2
Informational	9
Total:	**13**





6. Weaknesses

This section contains the list of discovered weaknesses.

GLIF5-2 | Incorrect Calculation of stakingTotalYbt When Setting Window Root

Fixed

Severity:	High	Probability:	Likely	Impact:	High

Path:

src/lp-plus/LpPlusMerkleHelper.sol#L336-L388

Description:

The Merkle root and the staking snapshot for a specific window are set using the flow **LpPlusMerkleHelper.setWindowRoot() -> LpPlus.commitWindow()**. The function **LpPlusMerkleHelper.setWindowRoot()** is responsible for computing the necessary data, while **LpPlus.commitWindow()** only stores that data on-chain.

Inside **LpPlusMerkleHelper.setWindowRoot()** the value **stakingTotalYbt** is computed by the internal function **_computeTwaStakingTotalYbt()**. This **stakingTotalYbt** is then stored in the window as stakingSnapshot.stakingYbtTwab and used as the denominator when allocating FIL to users. In other words, for a given window, if a user has a time-weighted average YBT balance **ybt**, their FIL allocation is:

$$\frac{\text{stakingSnapshot.totalFilToAllocate}}{\text{stakingSnapshot.stakingYbtTwab}} * \text{ybt}$$

FIL tokens.

There is a flaw in **_computeTwaStakingTotalYbt()** at line 317: **totalYbt** is initialized with **_lastVars.lastYbt**, which represents the **closingTotalYbt** of the previous staking snapshot. This is inconsistent with **_computeTwaValues()**, which computes each user's time-weighted average YBT balance and does not include the previous snapshot's **closingTotalYbt**. As a result, stakingYbtTwab can be overstated and an individual user's allocated FIL will be lower than expected.

Consider this example:

1. Assume there is only one card holder (**tokenIdGenerator = 2**).
2. Let the current **windowId = W > 1**. Define:
   - F = amount of FIL allocated for this window
   - T = **_windowEnd - _windowStart**
   - X = **windowIdToStakingSnapshot[W - 1].closingTotalYbt**
   - Y = **tokenIdToUserActions[1][length-1].ybtBalance**
3. Assume there are no new actions during this window. Following **_computeTwaStakingTotalYbt()**:
   - Line 317: **totalYbt = X**
   - Since there is only one card holder:
     - Line 354: **totalYbt = X + Y * T**
   - Line 358: **stakingTotalYbt = (X + Y * T) / T = X / T + Y**
     → **windowIdToStakingSnapshot[W].stakingYbtTwab = X / T + Y**
4. Now consider **_calculateUserSnapshot()** for the only holder. Because there is no snapshot for this window, the code block from lines 241–249 is executed:
   - Line 247: **allocatedFil = F * Y / stakingYbtTwab = F * Y / (X / T + Y) < F**
5. Therefore, despite being the sole card holder, the user does not receive the full FIL allocation for the window.

```
function _computeTwaStakingTotalYbt(
    LastTwaVars memory _lastVars,
    UserAction[][] memory _actionsPerToken,
    uint256 _windowStart,
    uint256 _windowEnd
) private pure returns (uint256 stakingTotalYbt) {
    require(_windowEnd > _windowStart, ZeroWindowDuration());


    uint256 totalYbt = _lastVars.lastYbt;


    for (uint256 i = 0; i < _actionsPerToken.length; ++i) {
        UserAction[] memory actions = _actionsPerToken[i];
        if (actions.length == 0) {
            continue;
        }


        // Determine last YBT/RWT before the window start
        LastTwaVars memory userLastVars;
        userLastVars.lastTime = _lastVars.lastTime;
```

```
        uint256 low = _binarySearchWindowId(_windowStart, actions, _compareExclusive);
        if (low > 0) {
            userLastVars.lastYbt = actions[low - 1].ybtBalance;
        }


        uint256 weightedYbt = 0;
        for (uint256 j = 0; j < actions.length; ++j) {
            UserAction memory action = actions[j];
            if (action.timestamp < _windowStart) {
                continue;
            } else if (action.timestamp > _windowEnd) {
                break;
            }


            uint256 dt = action.timestamp - userLastVars.lastTime;
            weightedYbt += userLastVars.lastYbt * dt;


            userLastVars.lastTime = action.timestamp;
            userLastVars.lastYbt = action.ybtBalance;
        }


        {
            uint256 dt = _windowEnd - userLastVars.lastTime;
            weightedYbt += userLastVars.lastYbt * dt;
        }


        totalYbt += weightedYbt;
    }


    uint256 windowDuration = _windowEnd - _windowStart;
    stakingTotalYbt = totalYbt / windowDuration;
}
```

Remediation:

The allocated FIL should be correctly distributed among the total balance.

GLIF5-3 | Incorrect Use of _windowStart as _windowId in _binarySearchWindowId Call

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

lp-plus/LpPlusMerkleHelper.sol#L336-L388

Description:

Inside **LpPlusMerkleHelper._computeTwaStakingTotalYbt()**, the function **_binarySearchWindowId()** is invoked with **_windowStart** passed as the **_windowId** argument:

```
uint256 low = _binarySearchWindowId(
    _windowStart,   /// <== @audit this is a timestamp, not a windowId
    actions,
    _compareExclusive
);
```

This is incorrect because **_windowStart** represents the timestamp of the last staking snapshot, whereas **_binarySearchWindowId()** expects a window ID, which should be a much smaller index value - not a timestamp.

Due to this mismatch, **_binarySearchWindowId()** returns an invalid result, causing **_computeTwaStakingTotalYbt()** to compute **stakingTotalYbt** incorrectly. As a consequence, **setWindowRoot()** will produce an incorrect staking total for the window, directly impacting the FIL allocation for users.

```
function _computeTwaStakingTotalYbt(
    LastTwaVars memory _lastVars,
    UserAction[][] memory _actionsPerToken,
    uint256 _windowStart,
    uint256 _windowEnd
) private pure returns (uint256 stakingTotalYbt) {
    require(_windowEnd > _windowStart, ZeroWindowDuration());


    uint256 totalYbt = _lastVars.lastYbt;
```

```
for (uint256 i = 0; i < _actionsPerToken.length; ++i) {
    UserAction[] memory actions = _actionsPerToken[i];
    if (actions.length == 0) {
        continue;
    }


    // Determine last YBT/RWT before the window start
    LastTwaVars memory userLastVars;
    userLastVars.lastTime = _lastVars.lastTime;
    uint256 low = _binarySearchWindowId(_windowStart, actions, _compareExclusive);
    if (low > 0) {
        userLastVars.lastYbt = actions[low - 1].ybtBalance;
    }


    uint256 weightedYbt = 0;
    for (uint256 j = 0; j < actions.length; ++j) {
        UserAction memory action = actions[j];
        if (action.timestamp < _windowStart) {
            continue;
        } else if (action.timestamp > _windowEnd) {
            break;
        }


        uint256 dt = action.timestamp - userLastVars.lastTime;
        weightedYbt += userLastVars.lastYbt * dt;


        userLastVars.lastTime = action.timestamp;
        userLastVars.lastYbt = action.ybtBalance;
    }


    {
        uint256 dt = _windowEnd - userLastVars.lastTime;
        weightedYbt += userLastVars.lastYbt * dt;
    }


    totalYbt += weightedYbt;
}
```

```
    uint256 windowDuration = _windowEnd - _windowStart;
    stakingTotalYbt = totalYbt / windowDuration;
}
```

Remediation:

The function should use the **_windowId** instead of **_windowStart**.

GLIF5-1 | The FIL reserves of the LpPlus contract may be insufficient to execute RWTFuture tokens

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/lp-plus/LpPlus.sol#L882-L884

Description:

In the **_mintRWTFutures()** function, there is a check to ensure that the FIL reserves are sufficient for the allocations in this mint.

```
// Check if FIL reserves are sufficient for this allocation
require(
    totalAllocatedFil <= address(this).balance, FilReservesDepleted(totalAllocatedFil, address(this).balance)
);
```

However, **totalAllocatedFil** in the above code is only a memory variable used within the **_mintRWTFutures()** function, so it cannot ensure that the contract's FIL reserves are sufficient for all RWTFuture allocations that will be executed.

Therefore, after using **claim()** and receiving RWTFuture tokens, users may be unable to execute them before expiration due to insufficient FIL reserves, resulting in a loss of claims.

Additionally, claims made through **claimAndExecute()**, which does not trigger **_mintRWTFutures()**, are missing a FIL-reserve. This can also cause unexecuted claims from the **claim()** function to be unable to execute due to insufficient FIL tokens.

```
function _mintRWTFutures(
    address _to,
    uint256[] memory _strikePrices,
    uint256[] memory _allocatedFils,
    uint256[] memory _expirationDates,
    LpPlusStorage storage $
) internal returns (uint256[] memory tokenIds) {
    uint256 length = _strikePrices.length;
    uint256 totalAllocatedFil = 0;

    // Validate and accumulate how much to allocate
```

```
    for (uint256 i = 0; i < length; ++i) {
        uint256 strikePrice = _strikePrices[i];
        uint256 allocatedFil = _allocatedFils[i];
        uint256 expirationDate = _expirationDates[i];

        require(strikePrice > 0, ZeroAmount());
        require(allocatedFil > 0, ZeroAmount());
        require(block.timestamp <= expirationDate, RWTFutureExpired(0, expirationDate));

        totalAllocatedFil += allocatedFil;
    }

    // Check if FIL reserves are sufficient for this allocation
    require(
        totalAllocatedFil <= address(this).balance, FilReservesDepleted(totalAllocatedFil, address(this).balance)
    );

    tokenIds = $.RWTFuture.mint(_to, _strikePrices, _allocatedFils, _expirationDates);
}
```

Remediation:

The total allocation of unexecuted claims should be checked against the available FIL when creating new claims (for example, in the **claimBatch()** function) to ensure they are executable in time.

GLIF5-13 | Full removal of YBT or RWT can be DoSed

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/lp-plus/LpPlus.sol:removeYBT, removeRWT

Description:

The LpPlus NFT contains a balance of YBT and RWT and the contract has functions to respectively add and remove these.

The contract also implements a minimum balance for both YBT and RWT (initially set to **1e18**), so if adding or removing would lead to a non-zero balance of less than the minimum, it would revert:

```
uint256 newBalance = vaultBalance - _RWTAmount;
uint256 minimumRWTBalance_ = $.minimumRWTBalance;
require(
    newBalance >= minimumRWTBalance_ || newBalance == 0,
    InvalidBalance(_tokenId, newBalance, minimumRWTBalance_)
);
```

Only if the remove function is called with the exact amount that is the full balance, the resulting **newBalance** would be 0 and it would be allowed.

However, due to this check, it becomes possible for an attacker to DoS full removal of YBT or RWT by front-running the transaction and donating 1 wei of the corresponding token using the permissionless add function.

For example:

- A user has an NFT with token ID 1 and 100 YBT and 100 RWT.
- The user calls **removeRWT(1, 100e18)**
- The attacker front-runs this call and calls **addRWT(1, 1)**, adding 1 wei to the NFT's YBT balance.
- The user's function call will now revert, because the calculated **newBalance** will be equal to 1 wei.

```
function removeYBT(uint256 _tokenId, uint256 _YBTAmount, address _receiver)
    external
    virtual
    whenNotPaused
    senderIsTokenOwner(_tokenId)
{
    LpPlusStorage storage $ = _getStorage();
    require(_YBTAmount > 0, ZeroAmount());

    address receiver = _receiver.normalize();
    require(receiver != address(0), ZeroAddress());

    // Check that we have enough balance to withdraw in the first place.
    uint256 vaultBalance = $.tokenIdToYBTBalance[_tokenId];
    require(_YBTAmount <= vaultBalance, InsufficientVaultBalance(_tokenId, _YBTAmount, vaultBalance));

    uint256 newBalance = vaultBalance - _YBTAmount;
    uint256 minimumYBTBalance_ = $.minimumYBTBalance;
    require(
        newBalance >= minimumYBTBalance_ || newBalance == 0,
        InvalidBalance(_tokenId, newBalance, minimumYBTBalance_)
    );
    $.tokenIdToYBTBalance[_tokenId] -= _YBTAmount;
    $.totalYbtStaked -= _YBTAmount;

    _pushUserAction(_tokenId, $);
    $.YBTToken.transfer(receiver, _YBTAmount);

    emit YBTVaultWithdrawn(_tokenId, msg.sender, _receiver, _YBTAmount, newBalance, $.totalYbtStaked);
}

function removeRWT(uint256 _tokenId, uint256 _RWTAmount, address _receiver)
    external
    virtual
    whenNotPaused
    senderIsTokenOwner(_tokenId)
{
    LpPlusStorage storage $ = _getStorage();
    require(_RWTAmount > 0, ZeroAmount());

    address receiver = _receiver.normalize();
    require(receiver != address(0), ZeroAddress());
```

```
        // Check that we have enough balance to withdraw in the first place.
        uint256 vaultBalance = $.tokenIdToRWTBalance[_tokenId];
        require(_RWTAmount <= vaultBalance, InsufficientVaultBalance(_tokenId, _RWTAmount, vaultBalance));

        uint256 newBalance = vaultBalance - _RWTAmount;
        uint256 minimumRWTBalance_ = $.minimumRWTBalance;
        require(
            newBalance >= minimumRWTBalance_ || newBalance == 0,
            InvalidBalance(_tokenId, newBalance, minimumRWTBalance_)
        );
        $.tokenIdToRWTBalance[_tokenId] = newBalance;
        $.totalRwtStaked -= _RWTAmount;

        _pushUserAction(_tokenId, $);
        $.RWTToken.transfer(receiver, _RWTAmount);

        emit RWTWithdrawn(_tokenId, msg.sender, _receiver, _RWTAmount, newBalance, $.totalRwtStaked);
    }
```

Remediation:

Consider adding the leftover balance to the amount to send when removing YBT or RWT.

GLIF5-4 | Inconsistent DEFAULT_FUTURE_EXPIRATION_PERIOD

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol#L40-L41

Description:

The LpPlus contract has a constant **DEFAULT_FUTURE_EXPIRATION_PERIOD** which is set to 180 days. However, the natspec comment states that it is 6 months, which is not the same and would be more accurately equal to 26 weeks or 182 days.

```
/// @dev The default future expiration period in seconds (6 months). RWT Futures lifespan.
uint256 private constant DEFAULT_FUTURE_EXPIRATION_PERIOD = 6 * 30 days;
```

Remediation:

We recommend to consider correcting either the comment or the constant's value.

GLIF5-5 | Missing address normalisation

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:mint#L168

Description:

In the function **mint** of the LpPlus contract, the function parameter **_to** is normalised into the stack variable **to** on line 173.

However, the raw **_to** address is still used as input for the modifier **validAddress(_to)** on line 168.

The modifier does not perform normalisation.

```
function mint(address _to, uint256 _referrerTokenId, uint256 _ybtToStake, uint256 _rwtToStake)
    external
    virtual
    whenNotPaused
    validAddress(_to)
    returns (uint256 tokenId)
{
    [..]
}

modifier validAddress(address _address) {
    require(_address != address(0), ZeroAddress());
    _;
}
```

Remediation:

We recommend normalising the address in the modifier:

```
modifier validAddress(address _address) {
--  require(_address != address(0), ZeroAddress());
++  require(_address.normalize() != address(0), ZeroAddress());
    _;
}
```

GLIF5-6 | Missing events for LpPlus configuration state changes

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol#L409-L492

Description:

In the contract LpPlus, there are numerous functions that allow the owner to change the configuration state variables. However, none of the functions emit events with the new values.

This makes off-chain tracking of the state cumbersome, as it is not always evident from the root transaction that a change was made (such as with Safe multi-sig wallets).

```
/// @inheritdoc ILpPlus
function setTreasury(address _treasury) external virtual onlyOwner {
    address treasuryNormalized = _treasury.normalize();
    require(treasuryNormalized != address(0), ZeroAddress());
    LpPlusStorage storage $ = _getStorage();
    $.treasury = treasuryNormalized;
}

/// @inheritdoc ILpPlus
function setWindowCommitter(address _windowCommitter) external virtual onlyOwner {
    address windowCommitterNormalized = _windowCommitter.normalize();
    require(windowCommitterNormalized != address(0), ZeroAddress());
    LpPlusStorage storage $ = _getStorage();
    $.windowCommitter = windowCommitterNormalized;
}

/// @dev Sets the RWT token address. Only callable by owner.
/// @param _RWTToken The new RWT token address
function setRWTToken(IERC20 _RWTToken) external virtual onlyOwner {
    address RWTTokenNormalized = address(_RWTToken).normalize();
    require(RWTTokenNormalized != address(0), ZeroAddress());
    LpPlusStorage storage $ = _getStorage();
    $.RWTToken = IERC20(RWTTokenNormalized);
}
```

```
/// @inheritdoc ILpPlus
function setYBTToken(IERC20 _YBTToken) external virtual onlyOwner {
    address YBTTokenNormalized = address(_YBTToken).normalize();
    require(YBTTokenNormalized != address(0), ZeroAddress());
    LpPlusStorage storage $ = _getStorage();
    $.YBTToken = IERC20(YBTTokenNormalized);
}

/// @inheritdoc ILpPlus
function setRWTFuture(IRWTFuture _RWTFuture) external virtual onlyOwner {
    require(address(_RWTFuture) != address(0), ZeroAddress());
    LpPlusStorage storage $ = _getStorage();
    $.RWTFuture = IRWTFuture(address(_RWTFuture).normalize());
}

/// @inheritdoc ILpPlus
function setFutureValidityDuration(uint256 _futureValidityDuration) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.futureValidityDuration = _futureValidityDuration;
}

/// @inheritdoc ILpPlus
function setMinimumYBTBalance(uint256 _minimumYBTBalance) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.minimumYBTBalance = _minimumYBTBalance;
}

/// @inheritdoc ILpPlus
function setMinimumRWTBalance(uint256 _minimumRWTBalance) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.minimumRWTBalance = _minimumRWTBalance;
}

/// @inheritdoc ILpPlus
function setMintBonus(RWTFutureBonus memory _mintBonus) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.mintBonus = _mintBonus;
}

/// @inheritdoc ILpPlus
function setReferrerBonus(RWTFutureBonus memory _referrerBonus) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.referrerBonus = _referrerBonus;
```

```
}

/// @inheritdoc ILpPlus
function setInterpolationParams(RwtInterpolation memory _interpolationParams) external virtual onlyOwner {
    // Note: Percentages allowed to exceed 100%.
    require(_interpolationParams.lowMultiplier < _interpolationParams.highMultiplier, InvalidPercentage());
    require(_interpolationParams.highRatio < _interpolationParams.lowRatio, InvalidPercentage());
    LpPlusStorage storage $ = _getStorage();
    $.interpolationParams = _interpolationParams;
}

/// @inheritdoc ILpPlus
function setMintPrice(uint256 _mintPrice) external virtual onlyOwner {
    LpPlusStorage storage $ = _getStorage();
    $.mintPrice = _mintPrice;
}
```

Remediation:

We recommend to add specific events for important state configuration changes that communicate the new value to off-chain indexers.

GLIF5-7 | Redundant balance check in withdrawFilFunds

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:withdrawFilFunds#L502-L512

Description:

In the function **withdrawFilFunds**, the owner can withdraw FIL from the contract. The function does some validation in the form of a balance check against the specified amount.

However, this check is redundant, as the same check is already performed in the **sendValue** function that is subsequently used to send the FIL:

```
function sendValue(address payable _recipient, uint _amount) internal {
    if (address(this).balance < _amount) revert InsufficientFunds();

    (bool success, ) = _recipient.call{value: _amount}("");
    if (!success) revert CallFailed();
}
```

Source: fevmate/contracts/utils/FilAddress.sol at 6a80e989847fd563df21360176cbfd5d08aaabbb · wadealexc/fevmate

```
function withdrawFilFunds(address _to, uint256 _amount) external virtual onlyOwner {
    address to = _to.normalize();
    require(to != address(0), ZeroAddress());
    require(_amount > 0, ZeroAmount());

    uint256 availableBalance = address(this).balance;

    require(_amount <= availableBalance, InsufficientVaultBalance(0, _amount, availableBalance));

    payable(to).sendValue(_amount);
}
```

Remediation:

We recommend to remove the redundant check in favour of gas savings and redundancy.

GLIF5-8 | LpPlus mint emits unnormalised address parameter

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:mint#L164-L223

Description:

In the function mint of LpPlus, the **_to** parameter is normalised into the local **to** variable. The latter is then used in subsequent actions, function calls and events.

However, only on line 180 there is an event that still emits the unnormalised version using **_to**.

```
function mint(address _to, uint256 _referrerTokenId, uint256 _ybtToStake, uint256 _rwtToStake)
    external
    virtual
    whenNotPaused
    validAddress(_to)
    returns (uint256 tokenId)
{
    [..]

    emit CardMinted(tokenId, msg.sender, _to, _referrerTokenId);

    [..]

}
```

Remediation:

We recommend using to in the event instead.

GLIF5-9 | Missing events for RWTFuture configuration state changes

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/RWTFuture.sol:setLpPlus#L151-L155

Description:

In the RWTFuture contact, there is an owner function **setLpPlus** to change the configuration state of the contract.

However, this function does not emit an event with the new value.

This makes off-chain tracking of the state cumbersome, as it is not always evident from the root transaction that a change was made (such as with Safe multi-sig wallets).

```
function setLpPlus(ILpPlus _lpPlus) external virtual onlyOwner {
    require(address(_lpPlus) != address(0), ZeroAddress());
    RWTFutureStorage storage $ = _getStorage();
    $.lpPlus = ILpPlus(address(_lpPlus).normalize());
}
```

Remediation:

We recommend to add a corresponding event to the function.

GLIF5-10 | Redundant balance check in _executeBatch

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:_executeBatch#L828-L862

Description:

In the function **_executeBatch**, the RWTFuture holder can execute their the strike price and purchase FIL from the contract. The function does some validation in the form of a balance check against the specified amount.

However, this check is redundant, as the same check is already performed in the **sendValue** function that is subsequently used to send the FIL:

```
function setLpPlus(ILpPlus _lpPlus) external virtual onlyOwner {
    require(address(_lpPlus) != address(0), ZeroAddress());
    RWTFutureStorage storage $ = _getStorage();
    $.lpPlus = ILpPlus(address(_lpPlus).normalize());
}
```

Source: fevmate/contracts/utils/FilAddress.sol at 6a80e989847fd563df21360176cbfd5d08aaabbb · wadealexc/fevmate

```
function _executeBatch(
    uint256[] memory _rwtFutureTokenIds,
    uint256[] memory _strikePrices,
    uint256[] memory _allocatedFils,
    address _receiver,
    LpPlusStorage storage $
) internal {
    [..]

    require(address(this).balance >= allocatedFil, InsufficientFILBalance(allocatedFil, address(this).balance));

    $.RWTToken.transferFrom(msg.sender, $.treasury, requiredRwt);

    payable(_receiver).sendValue(allocatedFil);
}
```

Remediation:

We recommend to remove the redundant check in favour of gas savings and redundancy.

GLIF5-11 | Redundant length checks in _claimBatch

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:_claimBatch#L769-L824

Description:

In the function **_claimBatch**, a holder of the LpPlus NFT can claim an RWTFuture using Merkle proofs. The function takes 3 different arrays and checks the length of each:

```
require(length > 0, ZeroAmount());
require(_userSnapshots.length > 0, ZeroAmount());
require(_proofs.length > 0, ZeroAmount());

require(_userSnapshots.length == length, InvalidLengths(_userSnapshots.length, length));
require(_proofs.length == length, InvalidLengths(_proofs.length, length));
```

However, the non-zero checks on the **_userSnapshots** and **_proofs** is redundant, as they are required to be equal to **length** and **length** is greater than 0.

```
    function _claimBatch(
        uint256[] calldata _windowIds,
        LpPlusMerkleHelper.UserSnapshot[] calldata _userSnapshots,
        bytes32[][] calldata _proofs,
        LpPlusStorage storage $
    )
        internal
        returns (uint256[] memory strikePrices, uint256[] memory allocatedFils, uint256[] memory
expirationDates)
    {
        uint256 length = _windowIds.length;

        // We don't strictly need to check the length of the other arrays,
        // but we do it so that the errors are consistent.
        require(length > 0, ZeroAmount());
        require(_userSnapshots.length > 0, ZeroAmount());
        require(_proofs.length > 0, ZeroAmount());
```

```
        require(_userSnapshots.length == length, InvalidLengths(_userSnapshots.length, length));
        require(_proofs.length == length, InvalidLengths(_proofs.length, length));

        [..]

    }
```

Remediation:

We recommend to remove the redundant check in favour of gas savings and redundancy.

GLIF5-12 | Expired RWTFuture NFTs can never be burned

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/lp-plus/LpPlus.sol:execute#L690-L715

Description:

In the protocol, the LpPlus contract is the only contract that is allowed to burn RWTFuture NFTs and it is only done in the **execute** function.

One requirement in the function, is that the RWTFuture has not expired for it to be executed:

```
require(
    block.timestamp <= position.expirationDate, RWTFutureExpired(rwtFutureTokenId, position.expirationDate)
);
```

Even though this is correct, it does make burning expired (and thus useless) RWTFuture NFTs impossible.

```
    function execute(uint256[] calldata _rwtFutureTokenIds, address _receiver) external virtual whenNotPaused
{
        LpPlusStorage storage $ = _getStorage();

        uint256 length = _rwtFutureTokenIds.length;
        require(length > 0, ZeroAmount());

        uint256[] memory strikePrices = new uint256[](length);
        uint256[] memory allocatedFils = new uint256[](length);

        for (uint256 i = 0; i < length; ++i) {
            uint256 rwtFutureTokenId = _rwtFutureTokenIds[i];
            require($.RWTFuture.ownerOf(rwtFutureTokenId) == msg.sender, NotTokenOwner(rwtFutureTokenId,
msg.sender));
```

```
            Position memory position = $.RWTFuture.tokenIdToPosition(rwtFutureTokenId);

            require(
                block.timestamp <= position.expirationDate, RWTFutureExpired(rwtFutureTokenId,
position.expirationDate)
            );

            strikePrices[i] = position.strikePrice;
            allocatedFils[i] = position.allocatedFil;
        }

        $.RWTFuture.burn(_rwtFutureTokenIds);
        _executeBatch(_rwtFutureTokenIds, strikePrices, allocatedFils, _receiver, $);
    }
```

Remediation:

Consider allowing the burning of expired RWTFuture NFTs, which would not need a subsequent call to **_executeBatch**.


hexens
x
GLIF