hexens × yelay

MAR.25

SECURITY REVIEW REPORT FOR YELAY

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 Web 3.0 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, Lido, 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.

EXECUTIVE SUMMARY

OVERVIEW

This audit assesses the newly introduced Deposit Lock feature in the Yelay-Lite Protocol. Clients within specific ID ranges will be restricted from creating projects where user deposits must stay in the vault for a predefined period. Only after this period expires can the position be redeemed or migrated.

Our security review lasted one week and involved an in-depth analysis of three smart contracts.

During the audit, we identified a critical-severity vulnerability that could allow users to fraudulently mint shares via a reentrancy attack. Note that the critical finding was located in the protocol's core codebase, which was outside the main scope of the audit.

Additionally, we found one medium-severity issue and five informational findings.

All reported issues were either addressed or acknowledged by the development team and subsequently verified by us.

As a result, we can confidently state that the protocol's security and overall code quality have improved following our audit.

SCOPE

The analyzed resources are located on:
https://github.com/YieldLayer/yelay-lite/pull/3

The issues described in this report were fixed in the following pull request:
https://github.com/YieldLayer/yelay-lite/pull/7

AUDITING DETAILS

STARTED
03.03.2025

DELIVERED
11.03.2025

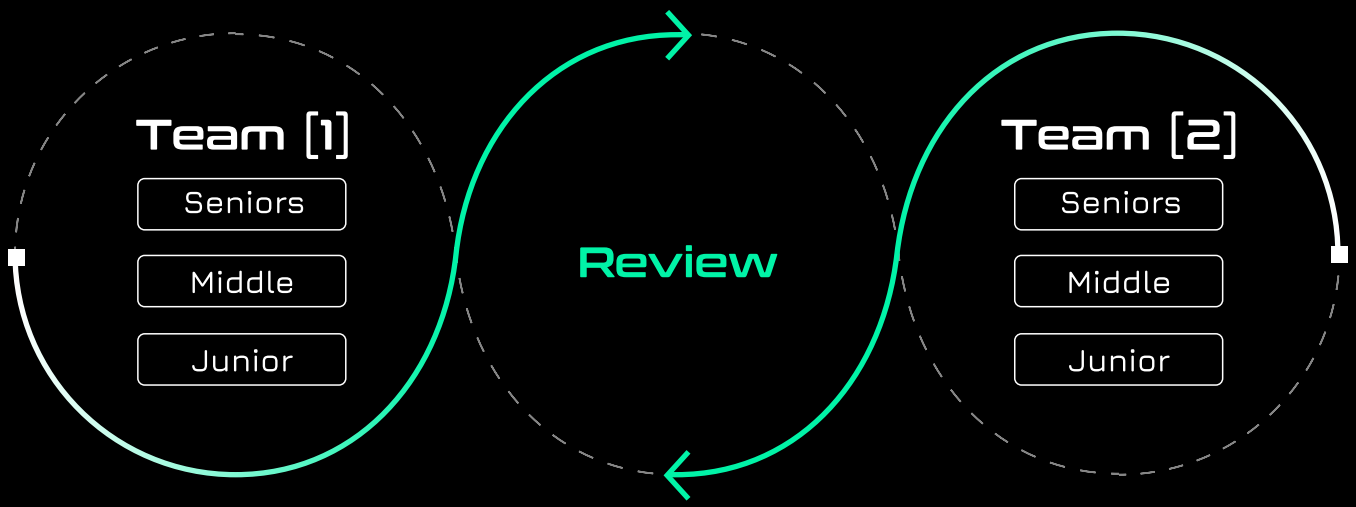
Review Led by

TRUNG DINH

Senior Security Researcher | Hexens

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

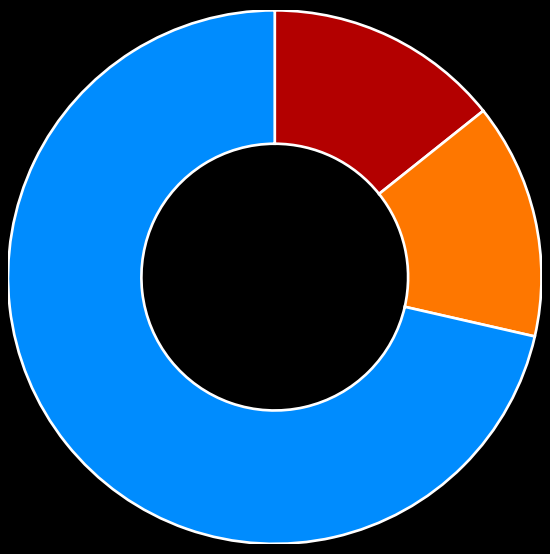
Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between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

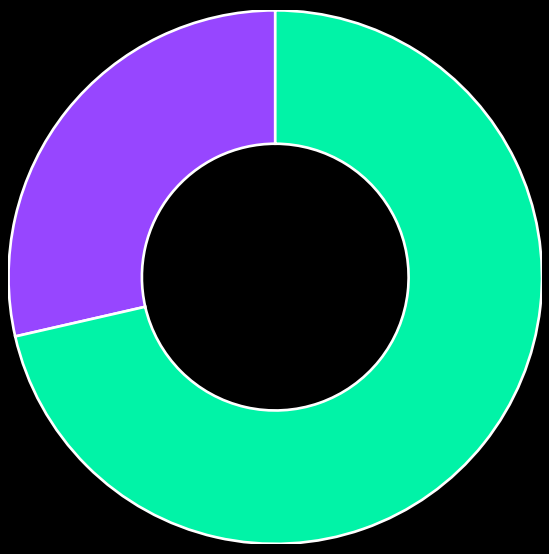
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	1
Low	0
Informational	5
	7





WEAKNESSES

This section contains the list of discovered weaknesses.

YELAY4-1

REENTRANCY IN DEPOSIT FUNCTION

SEVERITY: Critical

PATH:

src/facets/FundsFacet.sol#L147-L162

REMEDIATION:

Consider executing the _mint function after _updateLastTotalAssets function.

STATUS: Fixed

DESCRIPTION:

In the **FundFacets.deposit()** function, an ERC-1155 token is minted and assigned to the receiver on **line 147,** before the **STORAGE** variable **lastTotalAssets** is updated on **line 162.** This sequence of operations allows a reentrancy attack to occur.

```
shares = _convertToShares(assets, totalSupply(), newTotalAssets);
```

The number of shares issued is determined by multiplying **assets** by **totalSupply** and dividing by **totalAssets**.

Since **totalSupply** can be increased through the **_mint()** function while **totalAssets** remains unchanged during the minting process, an attacker can exploit a reentrancy vulnerability within the **ERC1155.onERC1155Received()** callback. By re-entering the **deposit()** function, the attacker can manipulate the share issuance mechanism,

taking advantage of the increased **totalSupply / totalAssets** ratio to mint additional shares fraudulently.

```
    _mint(receiver, projectId, shares, "");
    bool success;
    for (uint256 i; i < sM.depositQueue.length; i++) {
        (success,) =
sM.activeStrategies[sM.depositQueue[i]].adapter.delegatecall(
            abi.encodeWithSelector(
                IStrategyBase.deposit.selector, assets,
sM.activeStrategies[sM.depositQueue[i]].supplement
            )
        );
        if (success) {
            break;
        }
    }
    if (!success) {
        sF.underlyingBalance += SafeCast.toUint192(assets);
    }
    _updateLastTotalAssets(sF, newTotalAssets + assets);
```

USE SAFE TRANSFER FOR ERC20 TOKENS

SEVERITY: Medium

PATH:

src/plugins/DepositLockPlugin.sol#L143-L144

REMEDIATION:

Consider using OpenZeppelin's SafeERC20 versions with the safeTransfer and safeTransferFrom functions that handle the return value check as well as non-standard-compliant tokens.

STATUS: Fixed

DESCRIPTION:

The protocol aims to support all ERC-20 tokens, but the function **depositLocked()** relies on the original transfer functions.

Certain tokens, such as USDT, do not fully adhere to the EIP-20 standard, as their **transfer** and **transferFrom** functions return **void** instead of a boolean indicating success. Attempting to call these functions using the standard EIP-20 function signatures will cause a revert.

```
IERC20(underlyingAsset).transferFrom(msg.sender, address(this), assets);
IERC20(underlyingAsset).approve(vault, assets);
```

EVENT CALCULATES AND RETURNS WRONG VALUE WITH A VAGUE ERROR NAME

SEVERITY: Informational

PATH:

src/plugins/DepositLockPlugin.sol#L289

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **_removeSharesVariable** function in **DepositLockPlugin.sol** removes shares by iterating through all matured deposits until the requested amount is fully satisfied.

It requires the **remaining** assets to be zero after the while-loop at line 289. If not, the function triggers an error: **NotEnoughShares(requested, remaining)**. However, the **remaining** parameter used in the error is incorrect. Instead of passing the actual remaining shares, the function uses **shares - remaining**, which represents the shares that have already been fulfilled. This contradicts the intended meaning of **remaining**, as described in the NatSpec comment, which states it should reflect the remaining shares from the requested amount.

```
/**
  * @dev The requested shares to remove is not available.
  * @param requested The requested shares to remove.
  * @param remaining Remaining shares from requested amount.
  */
error NotEnoughShares(uint256 requested, uint256 remaining);
```

```
function _removeSharesVariable(address vault, uint256 projectId, uint256
shares) internal {
    uint256 pointer = depositPointers[vault][projectId][msg.sender];
    Deposit[] storage deposits = lockedDeposits[vault][projectId]
[msg.sender];
    uint256 lockPeriod = projectLockPeriods[vault][projectId];
    uint256 remaining = shares;
    while (pointer < deposits.length && remaining > 0) {
        Deposit storage deposit = deposits[pointer];
        if (!_isMatured(deposit.lockTime, lockPeriod)) {
            break;
        }
        if (remaining < deposit.shares) {
            deposit.shares = uint192(deposit.shares - remaining);
            remaining = 0;
        } else {
            remaining -= deposit.shares;
            deposit.shares = 0;
            pointer++;
        }
    }
    require(remaining == 0, LibErrors.NotEnoughShares(shares, shares -
remaining)); // @audit wrong error message
    depositPointers[vault][projectId][msg.sender] = pointer;
}
```

It is recommended:

```
  function _removeSharesVariable(address vault, uint256 projectId, uint256
shares) internal {
.   -- snip
-    require(remaining == 0, LibErrors.NotEnoughShares(shares, shares -
remaining));
+    require(remaining == 0, LibErrors.NotEnoughShares(shares, remaining));
      depositPointers[vault][projectId][msg.sender] = pointer;
  }
```

MISSING CLIENTOWNER VALIDATION CAN CAUSE DOS ON UPDATELOCKPERIOD AND UPDATEGLOBALUNLOCKTIME FOR OLD PROJECT IDS

SEVERITY: Informational

PATH:

src/plugins/DepositLockPlugin.sol#L110

src/plugins/DepositLockPlugin.sol#L127

src/facets/ClientsFacet.sol#L17-L32

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

In the **ClientsFacet.createClient()** function, there is no validation for the **clientOwner** address, allowing the same **clientOwner** to call **ClientsFacet.createClient()** multiple times. As a result, a **clientOwner** can be assigned a new **minProjectId** and **maxProjectId** while already having an existing assignment (**old_minProjectId** and **old_maxProjectId**).

Potential Issue:

1. **ClientsFacet.createClient()** is called with **clientOwner = X**, setting **ownerToClientData[X].minProjectId = old_minProjectId** and **ownerToClientData[X].maxProjectId = old_maxProjectId**.

2. **clientOwner = X** sets a high **projectGlobalUnlockTime** for a project ID **pid** within this range (**old_minProjectId < pid < old_maxProjectId**). This may be intentional to prevent early unstaking, with plans to update the unlock time later when certain conditions are met.
3. The **createClient()** function is called again with **clientOwner = X**, assigning a new project ID range (**new_minProjectId** to **new_maxProjectId**). As a result, **pid** now falls outside the valid range for **X**.
4. When **clientOwner = X** attempts to call **DepositLockPlugin.updateGlobalUnlockTime(pid)**, the transaction reverts (at line 127 - contract **DepositLockPlugin**) because **pid** is no longer associated with **X**.

Impact: This permanently locks funds in **pid**, as **clientOwner = X** can no longer update the unlock time.

A similar issue also occurs in the function **DepositLockPlugin.updateLockPeriod()**.

```
function createClient(address clientOwner, uint128 reservedProjects, bytes32
clientName) external {
    LibOwner.onlyOwner();
    LibClients.ClientsStorage storage clientStorage =
LibClients._getClientsStorage();
    require(reservedProjects > 0, LibErrors.ReservedProjectsIsZero());
    require(clientName != bytes32(0), LibErrors.ClientNameEmpty());
    require(clientStorage.isClientNameTaken[clientName] == false,
LibErrors.ClientNameTaken());
    uint128 minProjectId = SafeCast.toUint128(clientStorage.lastProjectId +
1);
    uint128 maxProjectId = minProjectId + reservedProjects - 1;
    clientStorage.ownerToClientData[clientOwner] =
        ClientData({minProjectId: minProjectId, maxProjectId: maxProjectId,
clientName: clientName});
    clientStorage.lastProjectId = maxProjectId;
    clientStorage.isClientNameTaken[clientName] = true;
    emit LibEvents.NewProjectIds(clientOwner, minProjectId, maxProjectId);
}
```

```
function updateLockPeriod(address vault, uint256 projectId, uint256
lockPeriod) external {
    require(_isProjectOwner(vault, projectId),
LibErrors.NotProjectOwner(vault, projectId, msg.sender));
    require(lockPeriod <= MAX_LOCK_PERIOD,
LibErrors.LockPeriodExceedsMaximum(lockPeriod));
    _setOrValidateLockMode(vault, projectId, LockMode.Variable);


    projectLockPeriods[vault][projectId] = lockPeriod;
    emit LibEvents.LockPeriodUpdated(vault, projectId, lockPeriod);
}
```

```
function updateGlobalUnlockTime(address vault, uint256 projectId, uint256
unlockTime) external {
    require(_isProjectOwner(vault, projectId),
LibErrors.NotProjectOwner(vault, projectId, msg.sender));
    _setOrValidateLockMode(vault, projectId, LockMode.Global);


    projectGlobalUnlockTime[vault][projectId] = unlockTime;
    emit LibEvents.GlobalUnlockTimeUpdated(vault, projectId, unlockTime);
}
```

Consider verifying if the clientOwner exists through the minProjectId value.

```
    function createClient(address clientOwner, uint128 reservedProjects,
bytes32 clientName) external {
        LibOwner.onlyOwner();
        LibClients.ClientsStorage storage clientStorage =
LibClients._getClientsStorage();
        require(reservedProjects > 0, LibErrors.ReservedProjectsIsZero());
        require(clientName != bytes32(0), LibErrors.ClientNameEmpty());
        require(clientStorage.isClientNameTaken[clientName] == false,
LibErrors.ClientNameTaken());
+       require(clientStorage.ownerToClientData[clientOwner].minProjectId ==
0, "clientOwnerTaken");
        uint128 minProjectId =
SafeCast.toUint128(clientStorage.lastProjectId + 1);
        uint128 maxProjectId = minProjectId + reservedProjects - 1;
        clientStorage.ownerToClientData[clientOwner] =
            ClientData({minProjectId: minProjectId, maxProjectId:
maxProjectId, clientName: clientName});
        clientStorage.lastProjectId = maxProjectId;
        clientStorage.isClientNameTaken[clientName] = true;
        emit LibEvents.NewProjectIds(clientOwner, minProjectId,
maxProjectId);
    }
```

UPDATEGLOBALUNLOCKTIME COULD LOCK USERS PERMANENTLY

SEVERITY: Informational

PATH:

src/plugins/DepositLockPlugin.sol#L126-L132

REMEDIATION:

Consider enforcing a MAX_LOCK_PERIOD for the global lock type. However, this remediation alone won't fully resolve the issue, as the owner could update the lock each time it reaches MAX_LOCK_PERIOD - 1, effectively extending it for another MAX_LOCK_PERIOD. Therefore, a cooldown period is also necessary, during which the lock period cannot be extended until the cooldown has elapsed, ensuring that users have the opportunity to withdraw their funds.

STATUS: Acknowledged

DESCRIPTION:

The **updateGlobalUnlockTime** function does not enforce a maximum lock period, which could hold users permanently locked. For example, if an owner updates the lock duration to 100 years from now, users would never be able to redeem their shares.

```
    function updateGlobalUnlockTime(address vault, uint256 projectId,
uint256 unlockTime) external {
        require(_isProjectOwner(vault, projectId),
LibErrors.NotProjectOwner(vault, projectId, msg.sender));
        _setOrValidateLockMode(vault, projectId, LockMode.Global);

        projectGlobalUnlockTime[vault][projectId] = unlockTime;
        emit LibEvents.GlobalUnlockTimeUpdated(vault, projectId,
unlockTime);
    }
```

USE UNCHECKED WHEN IT IS SAFE

SEVERITY: Informational

PATH:

src/plugins/DepositLockPlugin.sol::_removeSharesVariable()

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In Solidity (**^0.8.0**), adding the **unchecked** keyword around arithmetic operations can reduce gas usage where underflow or overflow is impossible due to preceding checks. In the **_removeSharesVariable()** function of the **DepositLockPlugin** contract, the arithmetic operation after the require statement **require(current >= shares, LibErrors.NotEnoughShares(shares, current))**; could be considered safe because shares is always less of the same as current.

```
    function _removeSharesGlobal(address vault, uint256 projectId, uint256
shares) internal {
        require(
            block.timestamp >= projectGlobalUnlockTime[vault][projectId],

LibErrors.GlobalUnlockTimeNotReached(projectGlobalUnlockTime[vault]
[projectId])
        );
        uint256 current = globalLockedShares[vault][projectId][msg.sender];
        require(current >= shares, LibErrors.NotEnoughShares(shares,
current));
        globalLockedShares[vault][projectId][msg.sender] = current - shares;
    }
```

Using unchecked in the arithmetic operation improves gas efficiency since the greater-than condition ensures that no underflow can occur.

```
    function _removeSharesGlobal(address vault, uint256 projectId, uint256
shares) internal {
        require(
            block.timestamp >= projectGlobalUnlockTime[vault][projectId],

LibErrors.GlobalUnlockTimeNotReached(projectGlobalUnlockTime[vault]
[projectId])
        );
        uint256 current = globalLockedShares[vault][projectId][msg.sender];
        require(current >= shares, LibErrors.NotEnoughShares(shares,
current));
+       unchecked {
          globalLockedShares[vault][projectId][msg.sender] = current -
shares;
+       }
    }
```

USERS CAN STILL DEPOSIT INTO A GLOBAL LOCK PROJECT AFTER A FINISHED DEADLINE.

SEVERITY: Informational

REMEDIATION:

Perhaps don't allow deposits after the unlock time has been reached for a global lock.

STATUS: Fixed

DESCRIPTION:

When the **projectGlobalUnlockTime** has been reached, all users are able to withdraw their shares from that project. However the user can still call **depositLocked** to lock assets after the unlock time has been reached with the opportunity to immediately withdraw.


hexens
×
yelay