hexens × yelay

JAN.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 covered new Yelay protocol Governance contract, more specifically they are Reward pool contracts and Tier NFTs for boosted rewards.

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

During our audit, we have identified 1 high severity vulnerability, that would have lead to some major functionality never working.

We have also identified several minor severity vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed 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.

SCOPE

The analyzed resources are located on:

https://github.com/solidant/yelay-rewards/tree/393b8c64f90583d510e7c728bfcf4db0d63d8e3b

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

https://github.com/solidant/yelay-rewards/tree/7e88e202ad01df2e1c8eb3cba7117e85b9e4d0a8

AUDITING DETAILS

STARTED
15.01.2025

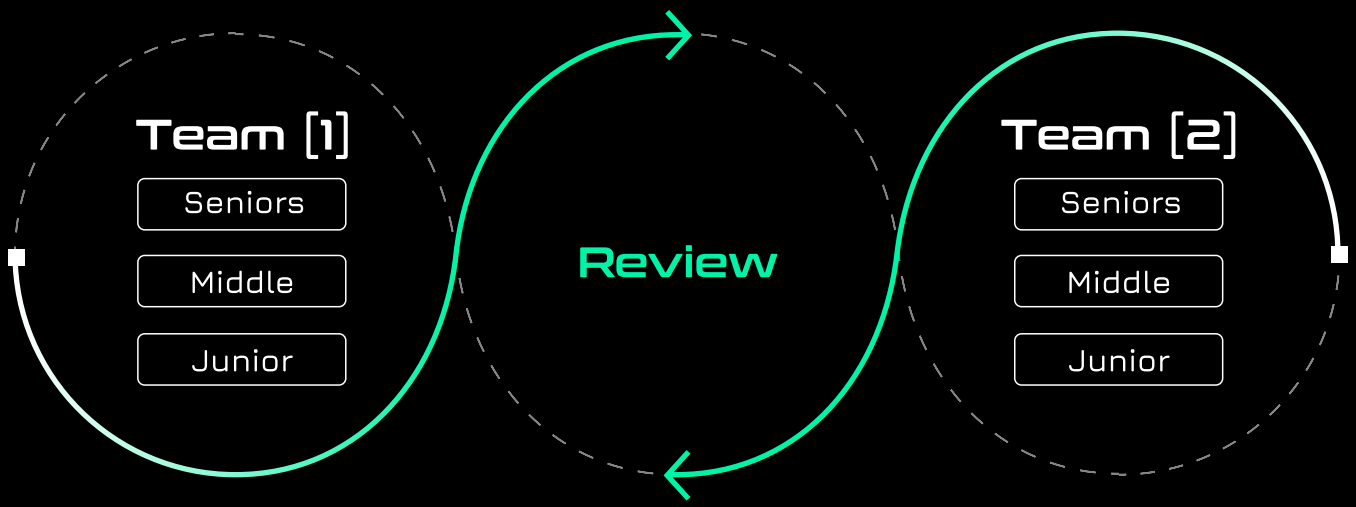
DELIVERED
23.01.2025

Review Led by

KASPER ZWIJSEN
Head of Audits | 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.

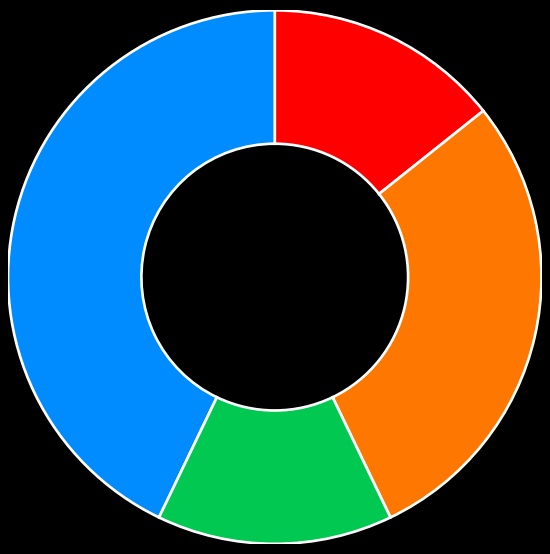
ISSUE SYMBOLIC CODES

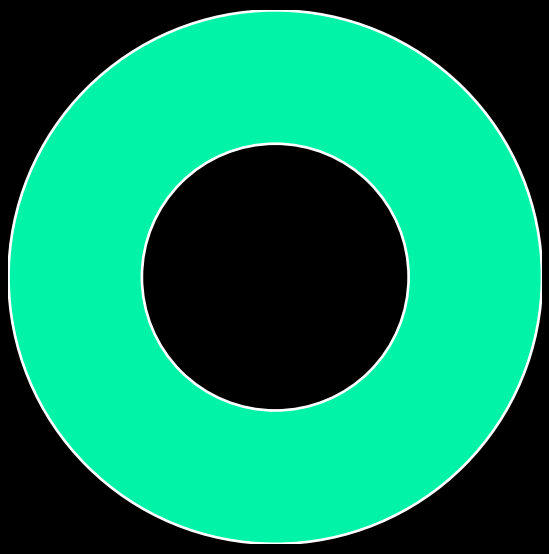
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	2
Low	1
Informational	3

Total: 7





WEAKNESSES

This section contains the list of discovered weaknesses.

YELAY3-1

PREMINT OF UNUSED ID WILL ALWAYS REVERT

SEVERITY: High

PATH:

src/Tier.sol#L181-L198

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **_premint()** function can be called from the **TierFactory** contract by the pre-minter role. Inside of the function, it first tries to distribute the NFTs from the unused IDs array.

However, these unused NFTs are being held by the **TierFactory** after they were returned in **returnNft()**. This means that the call to **_safeMint** for the **tokenId** will always lead to a revert, as that **tokenId** has already been minted.

```
//src/Tier.sol
...
    function _premint(address recipient) internal returns (uint256 tokenId)
{
          require(balanceOf(recipient) == 0, "Recipient can't hold multiple
tier NFTs");
        require(actualAmount < totalAmount, "All NFTs are sold out");

        actualAmount++;

        if (unusedIds.length > 0) {
            tokenId = unusedIds[unusedIds.length - 1];
            unusedIds.pop();
            _safeMint(recipient, tokenId);
        } else {
            tokenId = _tokenIdCounter;
            _tokenIdCounter++;
            _safeMint(recipient, tokenId);
        }

        return tokenId;
    }
...
```

In **Tier::_premint**, unused NFTs should be transferred to the receiver from **TierFactory** (the caller), similar to the function **buy()**:

```
//src/Tier.sol
...
    function _premint(address recipient) internal returns (uint256 tokenId)
{
        require(balanceOf(recipient) == 0, "Recipient can't hold multiple
tier NFTs");
        require(actualAmount < totalAmount, "All NFTs are sold out");

        actualAmount++;

        if (unusedIds.length > 0) {
            tokenId = unusedIds[unusedIds.length - 1];
            unusedIds.pop();
--          _safeMint(recipient, tokenId);
++          safeTransferFrom(msg.sender, recipient, tokenId);
        } else {
            tokenId = _tokenIdCounter;
            _tokenIdCounter++;
            _safeMint(recipient, tokenId);
        }

        return tokenId;
    }
...
```

A USER CAN HOLD MULTIPLE NFTS OF DIFFERENT TIERS

SEVERITY: Medium

PATH:

src/Tier.sol#L96-L115

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Each user can only own 1 NFT per address, which is enforced by the **_beforeTokenTransfer** function. However, this rule applies on a per-tier basis, meaning users can own different tiers of NFTs on the same address.

The **buyTier** function should have prevented a user from purchasing an NFT if they already own one, but if the user upgrades from Tier 1 to Tier 2, they can still buy another Tier 1 NFT. Normally the max “Boost” is 150%, however a user is now able to receive 155% through this owning a Tier5 and a Tier1.

It would be fair to only allow a user to hold one certain tier, because of the limited supplies of the higher tiers, they should be distributed to other players if they own one that is higher.

Additionally, users can use a second account to gain more advantage. If a second account buys a tier, upgrades it to the one the user doesn't hold yet, transfers it to the main account for each tier. This would allow the user to boost their total “boost” by 280% (combining multiple tiers 50% + 75% + 35% + 15% + 5%), rather than being limited to the max tier 150%. (This would require a unpaused Tier NFT).

```
//src/Tier.sol
...
    function buy(address buyer) external onlyOwner returns (uint256) {
        require(actualAmount < totalAmount, "All NFT tiers are sold out");

        uint256 tokenId;
        actualAmount++;

        // If there's an unused ID, reuse it
        if (unusedIds.length > 0) {
            tokenId = unusedIds[unusedIds.length - 1];
            unusedIds.pop();
            safeTransferFrom(msg.sender, buyer, tokenId);
        } else {
            // Else mint a fresh ID
            tokenId = _tokenIdCounter;
            _tokenIdCounter++;
            _safeMint(buyer, tokenId);
        }


        return tokenId;
    }
...
```

Add a check to the **buyTierNft** function so it reverts the transaction if the user already owns an NFT, by for example using the existing function **hasTier()** which returns a boolean without reverting. Also, look into the issue of transferring a tier to another user who already owns a different tier.

```
function buyTierNft(uint256 maxPrice) public {
        _checkAllowance(maxPrice);
++      require(hasTier(msg.sender) == false);
        uint256 newTierIndex = 0;
        bytes16 newTierId = _getNewTierId(maxPrice, 0);
        _buyTier(newTierIndex, newTierId);
}
```

Proof of concept (forge):
Add to the file **Tierfactory.t.sol**, and run with **forge test --mt testBuySecondTierNftPOC**.

```
...
    function testBuySecondTierNft() public {
        _fundAndApprove(addr1, 1000);

        vm.startPrank(addr1);
        paymentToken.approve(address(tierFactory), 1000);

        // Buy Tier1
        tierFactory.buyTierNft(10);
        Tier tier1 = _getTier(tier1Id);

        // Get NFT ID
        uint256 nftId = tier1.tokenOfOwnerByIndex(addr1, 0);

        // Approve TierFactory to move it
        tier1.approve(address(tierFactory), nftId);

        // Upgrade to Tier2
        tierFactory.upgradeTierNft(20, tier1Id, nftId);

        // buy another Tier1.
        tierFactory.buyTierNft(10);

        tier1.approve(address(tierFactory), nftId);
        tierFactory.upgradeTierNft(40, tier2Id, 0);

        // Now the user should have 1 Tier1 NFT
        assertEq(tier1.balanceOf(addr1), 1);
        // Now the user should have 1 Tier3 NFT
        Tier tier3 = _getTier(tier3Id);
        assertEq(tier3.balanceOf(addr1), 1);
    }
...
```

TIER BATCH PREMINT CAN BE DOSED

SEVERITY: Medium

PATH:

src/Tier.sol#L194

REMEDIATION:

Consider implementing a recipient validation check using the isContract function from the Address library.

STATUS: Fixed

DESCRIPTION:

The **TierFactory::remintTierBatch** allows a **ROLE_PREMINT** wallet to mint multiple NFTs in a single call. However, the premintBatch function does not verify whether the recipient address is a contract while using **_safeMint**. This oversight allows a malicious actor to deploy a contract with a custom onERC721Received implementation that forces the minting process to revert.

Practically, this may be exploited when attackers specify pre-calculated CREATE2 addresses as recipients, and front-running calls to **TierFactory::remintTierBatch** with their own call to deploy the malicious contract to the pre-calculated address.

```
//src/Tier.sol
...
    function _premint(address recipient) internal returns (uint256 tokenId) {
        ...
        else {
            tokenId = _tokenIdCounter;
            _tokenIdCounter++;
            _safeMint(recipient, tokenId);
        }
      ...
    }
...
```

Add to **Tier.t.sol**, and run with **forge test --match-test testPremitBatchDOS**.

```
//test/Tier.t.sol
contract TierTest is Test {
...
    function testPremitBatchDOS() public {
        vm.prank(addr1);
        attacker = new Attacker();

        address[] memory recipients = new address[](4);
        recipients[0] = makeAddr("recipient1");
        recipients[1] = makeAddr("recipient2");
        recipients[2] = address(attacker);
        recipients[3] = makeAddr("recipient3");
        vm.prank(owner);
        vm.expectRevert();
        tier.premintBatch(recipients);
    }
    ...
}
contract Attacker is IERC721Receiver{
    uint256 counter = 0;
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4){
        assert(1 != 1);
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

MISSING TIERID VALIDATION IN THE CREATETIER FUNCTION

SEVERITY: Low

PATH:

src/TierFactory.sol#L72-L89

REMEDIATION:

Consider enforcing a uniqueness check for the tierId argument to prevent accidental overwrites in the idToTier mapping.

STATUS: Fixed

DESCRIPTION:

According to the documentation, in the **TierFactory::createTier function**, the **tierId** must be a unique value. However, there are currently no validation checks to enforce this requirement. This can lead to incorrect usage of **tierId**, potentially causing incorrect assignments in the **idToTier[tierId]** mapping. In turn, the **getTierId** function may misbehave and fail to return the expected value.

```
//src/TierFactory.sol
...
    /**
     * @notice Deploy a new Tier as a BeaconProxy pointing to
`address(this)` as the beacon.
     * @dev Can be called only by the account with the ROLE_ADMIN
     ...
     * @param tierId      Unique tier ID
     ...
     */
    function createTier(
        string memory name,
        string memory symbol,
        bytes16 tierId,
        uint256 basePrice,
        uint256 multiplier,
        uint256 totalAmount,
        uint256 boost
    ) external onlyRole(ROLE_ADMIN, msg.sender) returns (Tier) {
        string memory initSig =
"initialize(address,string,string,string,bytes16,uint256,uint256,uint256,uin
t256)";
        // We encode the initialization calldata for Tier.initialize(...)
        bytes memory initCalldata = abi.encodeWithSignature(
            initSig, _accessControl, name, symbol, nftStorageURI, tierId,
basePrice, multiplier, totalAmount, boost
        );

        address tierAddress = address(new BeaconProxy(address(tierBeacon),
initCalldata));

        idToTier[tierId] = tierAddress;
        tiers.push(tierAddress);

        emit TierCreated(tierId, tierAddress);

        return Tier(payable(tierAddress));
    }
...
    function getTierId(address _address) public view returns (bytes16) {
        uint256 tierIndex = getTierIndex(_address);
        return Tier(tiers[tierIndex]).tierId();
    }
...
```

UNREACHABLE TIER CHECK DUE TO ARRAY INDEX OUT OF BOUNDS ERROR

SEVERITY: Informational

PATH:

src/TierFactory.sol#L250

REMEDIATION:

Move the highest tier check before calling _getNewTierId.

STATUS: Fixed

DESCRIPTION:

In the **upgradeTierNft** function of the TierFactory contract, there is a logical sequence issue where the highest tier check is performed after a potential array index out of bounds error. The function calls **_getNewTierId** with **newTierIndex** before checking if the current tier is the highest tier.

When a user with the highest tier attempts to upgrade, the function will revert with an array index out of bounds error, instead of the intended "You already have the highest tier" message.

```
//src/TierFactory.sol
...
    function upgradeTierNft(uint256 maxPrice, bytes16 currentTierId, uint256
currentNftId) public {
        _checkAllowance(maxPrice);
        require(getTierId(msg.sender) == currentTierId, "You are not the
owner of this tier");
        uint256 currentTierIndex = getTierIndex(msg.sender);
        require(msg.sender ==
Tier(tiers[currentTierIndex]).ownerOf(currentNftId), "You are not the owner
of this NFT");
        uint256 newTierIndex = currentTierIndex + 1;
        bytes16 newTierId = _getNewTierId(maxPrice, newTierIndex);
        require(currentTierIndex < getHighestTierIndex(), "You already have
the highest tier");
        Tier(tiers[currentTierIndex]).returnNft(msg.sender);
        _buyTier(newTierIndex, newTierId);
    }
...
    function _getNewTierId(uint256 maxPrice, uint256 newTierIndex) private
view returns (bytes16) {
        bytes16 newTierId = Tier(tiers[newTierIndex]).tierId();
        require(
            Tier(tiers[newTierIndex]).totalAmount() -
Tier(tiers[newTierIndex]).actualAmount() > 0,
            "All tokens are already minted"
        );
        require(maxPrice >= getCurrentPrice(newTierId), "Current price is
higher");
        return newTierId;
    }
...
```

REDUNDANT VALIDATION OF THE RECIPIENT'S BALANCE IN THE _PREMINT FUNCTION

SEVERITY: Informational

PATH:

src/Tier.sol#L182

REMEDIATION:

Consider removing the redundant validation of the recipient's balance in the _premint function.

STATUS: Fixed

DESCRIPTION:

The **_premint()** function of the **Tier** contract includes a validation for the recipient's balance as follows:

```
function _premint(address recipient) internal returns (uint256 tokenId) {
  require(balanceOf(recipient) == 0, "Recipient can't hold multiple tier
NFTs");
  [...]
}
```

However, the above check is redundant and unnecessary because there is already a validation restricting only one Tier NFT per address in **Tier::_beforeTokenTransfer**, except for the **TierFactory** owner.

Additionally, that validation will also prevent the owner from receiving airdropped Tier NFTs, even though the owner is intended to hold multiple Tier NFTs.

```
//src/Tier.sol
...
    function _beforeTokenTransfer(address from, address to, uint256 tokenId,
uint256 batchSize)
        internal
        override(ERC721Upgradeable, ERC721EnumerableUpgradeable)
    {
        require(balanceOf(to) == 0 || to == owner(), "Tier: Only one NFT per
address permitted");
        [...]
    }
...
    function _premint(address recipient) internal returns (uint256 tokenId)
{
      require(balanceOf(recipient) == 0, "Recipient can't hold multiple tier
NFTs");
      require(actualAmount < totalAmount, "All NFTs are sold out");

      actualAmount++;

      if (unusedIds.length > 0) {
          tokenId = unusedIds[unusedIds.length - 1];
          unusedIds.pop();
          _safeMint(recipient, tokenId);
      } else {
          tokenId = _tokenIdCounter;
          _tokenIdCounter++;
          _safeMint(recipient, tokenId);
      }

      return tokenId;
    }
...
```

REDUNDANT VALIDATION IN UPGRADETIERNFT

SEVERITY: Informational

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **upgradeTierNft** function, the user provides two parameters: **currentTierId** and **currentNftId**. However, the **currentTierId** and **currentNftId** are not used, but instead, they are only validated against the user's actual tier and NFT ownership in the system. Since it always upgrades the highest tier, it could be using the system values directly.

Redundant Input:

1. **Tier ID Validation**: The function doesn't require the user to explicitly provide their tier ID because it can be derived from the contract by calling **getTierIndex(msg.sender)**, which retrieves the user's current tier index.
2. **NFT Ownership Validation**: The check **require(msg.sender == Tier(tiers[currentTierIndex]).ownerOf(currentNftId), "You are not the owner of this NFT");** is redundant.
   The **getTierIndex(msg.sender)** already ensures the user owns an NFT in the given tier (it checks if the balance is greater than 0). Therefore, manually verifying ownership is unnecessary.

```
function upgradeTierNft(uint256 maxPrice,
--          bytes16 currentTierId,
--          uint256 currentNftId)
public {
        _checkAllowance(maxPrice);
--      require(getTierId(msg.sender) == currentTierId, "You are not the
owner of this tier");
        uint256 currentTierIndex = getTierIndex(msg.sender);
--      require(msg.sender ==
Tier(tiers[currentTierIndex]).ownerOf(currentNftId), "You are not the owner
of this NFT");
        uint256 newTierIndex = currentTierIndex + 1;

        bytes16 newTierId = _getNewTierId(maxPrice, newTierIndex);
        require(currentTierIndex < getHighestTierIndex(), "You already have
the highest tier");
        Tier(tiers[currentTierIndex]).returnNft(msg.sender);

        _buyTier(newTierIndex, newTierId);
}
```


hexens
×
yelay