SALUS SECURITY

JAN 2023

CODE SECURITY ASSESSMENT

LIFEFORM

Overview

Project Summary

- Name: LifeForm
- Version: Commit b1ea906
- Platform: BSC
- Language: Solidity
- Repository: https://github.com/halo-avatar/lifeform-protocol
- Audit Scope: See Appendix - 1

Project Dashboard

Application Summary

Name	LifeForm
Version	v3-rc0
Type	Solidity
Dates	Jan 29 2023
Logs	Oct 10 2022; Oct 24 2022; Jan 29 2023

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	4
Total Low-Severity issues	5
Total informational issues	2
Total	12

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	BaseMintRule. _burnAvatar721() does not update the state of _burnAssetList and _freeTimePoint if extraInfo.erc20Amount = 0	High	Business Logic	Unresolved
2	tokenId 0 cannot be withdrawn from AnswerFirst	Medium	Business Logic	Unresolved
3	Minters/IAMs can mint tokens for free	Medium	Centralization	Unresolved
4	User could mint token for free if the signer is compromised	Medium	Centralization	Unresolved
5	Centralization risk	Medium	Centralization	Unresolved
6	Unnecessary payable modifier	Low	Redundancy	Unresolved
7	Incorrect loop start index	Low	Business Logic	Unresolved
8	Unbounded loop in listMyNft()	Low	Business Logic	Unresolved
9	Project can be started via setUserStart() in HotBuyFactoryV2	Low	Business Logic	Unresolved
10	Difficult to query the airDropId for an activity	Low	Auditing and Logging	Unresolved
11	Mismatch between contract name and file name	Informational	Code Quality	Unresolved
12	Incorrect OpenZeppelin version in package.json	Informational	Configuration	Unresolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. BaseMintRule. _burnAvatar721() does not update the state of _burnAssetList and _freeTimePoint if extraInfo.erc20Amount = 0	
Severity: High	Category: Business Logic
Target: - LFT/AvatarMintRule/BaseMintRule.sol	

Description

The _burnAssetList state variable keeps track of the tokenIds that have been applied for burning by applyBurnAvatar721() but have not yet been burned by _burnAvatar721(). The _freeTimePoint state variable keeps track of the moment the tokenId was applied to burn.

After the burning of a tokenId by _burnAvatar721(), the information pertaining to that tokenId in _burnAssetList and _freeTimePoint should be reset.

However, the _burnAvatar721() function only updates _burnAssetList and _freeTimePoint if extraInfo.erc20Amount > 0.

LFT/AvatarMintRule/BaseMintRule.sol:L195-L252

```
function _burnAvatar721(uint256 tokenId) internal {
    _avatar721.burn(tokenId);

    IAvatar721.ExtraInfo memory extraInfo = _avatar721.getExtraInfo(tokenId);
    if (extraInfo.erc20Amount > 0) {
        require(extraInfo.erc20 != address(0x0), "invalid erc20 address");

        //fast burn fee split
        uint256 returnAmount = extraInfo.erc20Amount;
        if (_burnAssetList[msg.sender].contains(tokenId)) {
            uint256 maxFee = extraInfo.erc20Amount.mul(_fastBurnFeeRate).div(10000);
            uint256 punishFee = 0;
            uint256 finalFreeTime = _freeTimePoint[tokenId].add(_burnDuration);
            if (block.timestamp < finalFreeTime) {
                punishFee = maxFee.mul(finalFreeTime.sub(block.timestamp)).div(_burnDuration);
            }

            _burnAssetList[msg.sender].remove(tokenId);
            _freeTimePoint[tokenId] = 0;

            if (punishFee > 0) {
                (IERC20)(extraInfo.erc20).safeTransfer(_teamWallet, punishFee);
            }
            returnAmount = extraInfo.erc20Amount.sub(punishFee);
        }

        (IERC20)(extraInfo.erc20).safeTransfer(msg.sender, returnAmount);
    }

    ...
}
```

In other words, when extraInfo.erc20Amount = 0, the _burnAvatar721() function will not update _burnAssetList and _freeTimePoint.

Consequently, if a tokenId is minted by mint() with mintData.stakeErc20Amount = 0, and the token owner calls applyBurnAvatar721(tokenId, true) to invoke _burnAvatar721() to force burn that tokenId, that tokenId will remain in _burnAssetList and _freeTimePoint.

Then, when the _burnDuration for that tokenId has ended, if the token owner calls claimBurnHeritage() to burn tokenIds in _burnAssetList and claim heritages, the message call will fail because the said tokenId has already been burned and cannot be burned again.

In summary, if a user burns a tokenId with extraInfo.erc20Amount = 0, then the user will not be able to successfully call claimBurnHeritage().

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

contract POC is Test {
    error OwnerQueryForNonexistentToken();

    BaseMintRule public baseMintRule;
    Avatar721 public avatar721;

    address public user;

    function setUp() public {
        vm.createSelectFork("https://rpc.ankr.com/bsc", 25173784);

        baseMintRule = BaseMintRule(0xa8fC6DAcCc5a9A3EAa35e71d06686aC0BcBeFb00);
        vm.label(address(baseMintRule), "BaseMintRule");

        avatar721 = Avatar721(baseMintRule._avatar721());
        vm.label(address(avatar721), "Avatar721");

        user = 0xeA425E505C84F961ccdB52637dc74acD725a7547;
        vm.label(user, "user");
    }

    function test_BurnedTokenIdInBurnAssetList() public {
        // assert that the info for burned and claimed token is not cleaned
        // in BaseMintRule._burnAssetList
        vm.prank(user);
        BurnList[] memory burnList = baseMintRule.getBurnList();
        assertGt(burnList.length, 1);
        uint256 tokenId = burnList[0].tokenId;
        assertEq(tokenId, 187669);
        bool tokenBurned = avatar721.getOwnershipOf(tokenId).burned;
        assertTrue(tokenBurned);
    }
```

```
    function test_ClaimBurnHeritageWillRevert() public {
        // assert that a subsequent call to claimBurnHeritage will revert
        vm.expectRevert(OwnerQueryForNonexistentToken.selector);
        vm.prank(user);
        baseMintRule.claimBurnHeritage();
    }
}

struct BurnList {
    uint256 tokenId;
    uint256 freePoint;
}

struct TokenOwnership {
    address addr;
    uint64 startTimestamp;
    bool burned;
}

interface BaseMintRule {
    function getBurnList() external view returns (BurnList[] memory);
    function _avatar721() external view returns (address);
    function claimBurnHeritage() external;
}

interface Avatar721 {
    function getOwnershipOf(uint256 tokenId) external view returns (TokenOwnership
memory);
}
```

To illustrate the issue, this proof of concept forked the BSC mainnet state at height 25173784, and selected a user (0xeA425E505C84F961ccdB52637dc74acD725a7547). It first asserted that this user had a burned tokenId (187669), then showed that a call to claimBurnHeritage() from the user would fail because the tokenId had already been burned.

Recommendation

Consider updating the tokenId info in _burnAssetList and _freeTimePoint regardless of extraInfo.erc20Amount value in _burnAvatar721().

2. tokenId 0 cannot be withdrawn from AnswerFirst	
Severity: Medium	Category: Business Logic
Target: - LFT/AnswerFirst.sol	

Description

LFT/AnswerFirst.sol:L233-L252

```
function withdrawNFTs() public whenNotPaused nonReentrant {

    uint256[] memory ids = new uint256[](_activityId);
    uint32 count=0;
    for(uint256 k=0; k<_activityId; k++){
        uint256 tokenId = _registerInfo[k][msg.sender].tokenId;
        if(tokenId > 0){
            IERC721(address(_erc721)).safeTransferFrom(address(this), msg.sender, tokenId);
            _registerInfo[k][msg.sender].tokenId=0;

            ids[count]=tokenId;
            count = count+1;
        }
    }

    require(count>0, "nothing to be withdrawed! the pledge hasn't expired yet");

    emit eWithdraw(ids, msg.sender, block.timestamp, _activityId);

}
```

The withdrawNFTs() function withdraws the token only if tokenId > 0. However the tokenId of the underlying _erc721, Avatar721, starts from 0.

As a result, if the owner of tokenId 0 stakes this token in the AnswerFirst contract using register(), the owner will not be able to withdraw it using withdrawNFTs().

Recommendation

Consider using the isRegister() function instead of tokenId > 0 to check whether the tokenId is registered.

3. Minters/IAMs can mint tokens for free	
Severity: Medium	Category: Centralization
Target: - LFT/HotBuyFactoryV2.sol - LFT/Avatar721.sol - LFT/Adorn1155.sol - LFT/Adorn721.sol - LFT/LifeformToken.sol	

Description

1. IAMs can mint tokens for free in HotBuyFactoryV2

LFT/HotBuyFactoryV2.sol:L293-L349

```
function _mint721(address nftContract, address costErc20, uint256 mintCount, Condition calldata
condition, bytes memory dataSignature )  internal {

    ...

    bool exist = _IAMs[nftContract].contains(msg.sender);

    ...

    if(!exist){

        require(verify(condition, msg.sender, dataSignature), "this sign is not valid");

        uint256 count = historyCount + mintCount;
        require(count <= condition.limitCount,"sale count is max ");

        //once signCode
        require(isValidSignCode(nftContract,condition.signCode),"invalid signCode!");
    }

    ...

    IAdorn721(nftContract).mint(msg.sender,mintCount);

    ...
}
```

LFT/HotBuyFactoryV2.sol:L353-L411

```
function _mint1155(address nftContract, address costErc20, uint256 mintCount, Condition calldata
condition, bytes memory dataSignature )  internal {

    ...

    bool exist = _IAMs[nftContract].contains(msg.sender);

    ...

    if(!exist){

        require(verify(condition, msg.sender, dataSignature), "this sign is not valid");

        uint256 count = historyCount + mintCount;
        require(count <= condition.limitCount,"sale count is max ");

        //once signCode
        require(isValidSignCode(nftContract,condition.signCode),"invalid signCode!");
    }

    ...

    IAdorn1155(nftContract).mint(msg.sender,tokenId,mintCount,"");

    ...

}
```

The _mint721() and _mint1155() functions only check the dataSignature if the caller is not an IAM. The IAMs can bypass the signature check and mint the underlying token for free.

LFT/HotBuyFactoryV2.sol:L157-L163

```
function addIAM(address nftContract,address minter) public onlyOwner {
    _IAMs[nftContract].add(minter);
}

function removeIAM(address nftContract,address minter) public onlyOwner {
    _IAMs[nftContract].remove(minter);
}
```

The IAMs can be added and removed by the owner. There is no restriction on the minter address in addIAM(), which means that the IAM can be either an EOA address or a contract. And there is no event associated with addIAM() and removeIAM().

2. Minters can mint tokens for free in Avatar721

LFT/Avatar721.sol:L131-L47

```
function mint(address to, IAvatar721.ExtraInfo calldata info) external override  onlyMinter returns
(uint256 id)
{
    uint256 tokenId = _currentIndex;

    _extraInfo[_currentIndex].mintRule = info.mintRule;
    _extraInfo[_currentIndex].erc20 = info.erc20;
    _extraInfo[_currentIndex].erc20Amount = info.erc20Amount;
    _extraInfo[_currentIndex].erc721 = info.erc721;
    _extraInfo[_currentIndex].children721 = info.children721;
    _extraInfo[_currentIndex].erc1155 = info.erc1155;
    _extraInfo[_currentIndex].children1155 = info.children1155;
    _extraInfo[_currentIndex].amount1155 = info.amount1155;
    _extraInfo[_currentIndex].id = _currentIndex;
    _safeMint(to, 1, "");

    return tokenId;
}
```

Minters can mint Avatar721 tokens for free to any address using mint().

LFT/Avatar721.sol:L59-L70

```
/**
 * @dev function to grant permission to a minter
 */
function addMinter(address minter) public onlyOwner {
    _minters[minter] = true;
}
/**
 * @dev function to remove permission to a minter
 */
function removeMinter(address minter) public onlyOwner {
    _minters[minter] = false;
}
```

Minters can be added and removed by the owner. There is no restriction on the minter address in addMinter(), which means that the minter can be either an EOA address or a contract. And there is no event associated with addMinter() and removeMinter().

3. Minters can mint tokens for free in Adorn1155

LFT/Adorn1155.sol:L114-L120

```
function mint(address account, uint256 tokenId, uint256 amount, bytes memory data) external override
onlyMinter
{
    _mint(account, tokenId, amount, data);
    if(!_mintedIds.contains(tokenId)){
        _mintedIds.add(tokenId);
    }
}
```

Minters can mint Adorn1155 tokens for free to any address using mint().

LFT/Adorn1155.sol:L68-L80

```
/**
 * @dev function to grant permission to a minter
 */
function addMinter(address minter) public onlyOwner {
    _minters[minter] = true;
}

/**
 * @dev function to remove permission to a minter
 */
function removeMinter(address minter) public onlyOwner {
    _minters[minter] = false;
}
```

Minters can be added and removed by the owner. There is no restriction on the minter address in addMinter(), which means that the minter can be either an EOA address or a contract. And there is no event associated with addMinter() and removeMinter().

4. Minters can mint tokens for free in LifeformToken

LFT/LifeformToken.sol:L143-L157

```
function mint(address account, uint256 amount) external returns (bool){

    require(account != address(0), "ERC20: mint to the zero address");
    require(_minters[msg.sender], "!minter");

    uint256 newMintSupply = _totalSupply.add(amount);
    require( newMintSupply <= maxSupply,"supply is max!");

    _totalSupply = newMintSupply;
    _balances[account] = _balances[account].add(amount);

    emit Transfer(address(0), account, amount);

    return true;
}
```

Minters can mint Lifeform tokens for free to any address using mint().

LFT/LifeformToken.sol:L160-L168

```
function addMinter(address minter) public onlyOwner
{
    _minters[minter] = true;
}

function removeMinter(address minter) public onlyOwner
{
    _minters[minter] = false;
}
```

Minters can be added and removed by the owner. There is no restriction on the minter address in addMinter(), which means that the minter can be either an EOA address or a contract. And there is no event associated with addMinter() and removeMinter().

The same issue also applies to Adorn721 and LAP.

In summary, the minters/IAMs have too much power with too little restriction and oversight.

Recommendation

Consider emitting events when adding and removing minters/IAMs to help index the minters/IAMs off-chain, or implementing a getMinters()/getIAMs() function to help query the current minters/IAMs on-chain.

Also consider adding appropriate restrictions when adding minters/IAMs. For example, disallow an EOA address to be the minter/IAMs.

4. User could mint token for free if the signer is compromised	
Severity: Medium	Category: Centralization
Target: - LFT/StoreFactoryV3.sol - LFT/HotBuyFactoryV2.sol	

Description

The mintAdorn() and mintAdornWithETH() functions in StoreFactoryV3 and HotBuyFactoryV2 allow the condition.price, a user input parameter, to be zero. In other words, the StoreFactoryV3 and HotBuyFactoryV2 allow the user to mint the token for free if the user can get the signature for that from the signer.

The potential risk of this implementation is that if an attacker gains access to the signer’s private key, the attacker can mint tokens for free in StoreFactoryV3 and HotBuyFactoryV2 contracts.

Recommendation

Consider taking good care of the signer's private key.

5. Centralization risk	
Severity: Medium	Category: Centralization
Target: - All	

Description

There are a number of privileged roles that exercise wide-ranging powers:

- Owner of Avatar721, Adorn721, Adorn1155, and LAP contract
  - can add or remove minter
  - can set baseURI
  - can set metaType
- Owner of LifeformToken contract
  - can add or remove minter
  - can enable or disable token transfer
- Owner of LBT contract
  - can airdrop token
  - can switch public mint on or off
  - can update signer
  - can add or remove SBTContract
  - can turn personality switch on or off
- Owner of AvatarFactoryV2 contract
  - can add or remove IAM
  - can add or remove mint rule address
  - can update signer
  - can start or stop user mint
- Owner of BaseMintRule contract
  - can update factory address
  - can update underlying NFT address
  - can withdraw ETH, ERC20, ERC721, ERC1155 to arbitrary address
- Owner of HotBuyFactoryV2 contract
  - can add or remove IAM
  - can withdraw ERC20 to arbitrary address
  - can update vault
  - can update signer

    - can start a project
    - can start or stop user mint
- Owner of StoreFactoryV3 contract
    - can withdraw ERC20 to arbitrary address
    - can add or remove IAM
    - can start or stop user mint
    - can update teamWallet
    - can update signer
- Owner of AnswerFirst contract
    - can withdraw underlying ERC721 to arbitrary address
    - can add or remove IAM
    - can set register fee and stake fee
    - can update startTime and endTime for the current activity
    - can update airDropId
    - can update underlying ERC20, ERC721, ERC1155 address
    - can pause the contract
- Owner of LuckyCheckIn contract
    - can start new activity
    - can add or remove SBTContract

If the privileged owner account is a plain EOA account, this can be worrisome and pose a risk to the users.

Recommendation

We recommend transferring the privileged owner roles to a community-governed DAO, or at least to multisig accounts. In addition, a timelock-based mechanism can be implemented to demonstrate the project owners' commitment to the ongoing health of this project.

6. Unnecessary payable modifier	
Severity: Low	Category: Redundancy
Target: - LFT/LifeformToken.sol	

Description

LFT/LifeformToken.sol:L222

```
fallback() external payable {}
```

There is a payable fallback function in the LifeformToken contract that allows the contract to receive funds from other addresses. However, there is no withdrawal logic in the LifeformToken contract.

As a result, if one accidentally sends BNB to the LifeformToken contract, the funds are locked in the contract and there is no way to move the funds out.

Recommendation

Consider removing the payable modifier. In addition, if there is no good reason to leave a fallback() function in the contract, consider removing the fallback() function as well.

7. Incorrect loop start index

Severity: Low	Category: Business Logic
Target: - LFT/AnswerFirst.sol	

Description

LFT/AnswerFirst.sol:L166-L174

```
function newActivity(uint256 airDropId) public onlyIAM {
    _activityId++;
    _activityInfo[_activityId].startTime = block.timestamp;
    _activityInfo[_activityId].endTime = block.timestamp + _deltaTime;

    _airDropId = airDropId;

    emit eNewActivity(_activityId,block.timestamp,block.timestamp + _deltaTime);
}
```

After starting the first activity in AnswerFirst via newActivity(), the _activityId is incremented to 1. When iterating over all the activities, the _activityId should start from 1.

LFT/AnswerFirst.sol:L188-195

```
function getAllRegisterInfo(address owner) public view returns(RegisterInfo[] memory ){

    RegisterInfo[] memory records = new RegisterInfo[](_activityId+1);
    for(uint256 i=0; i<=_activityId; i++){
       records[i] = _registerInfo[i][owner];
    }
    return records;
}
```

However, the for-loop in getAllRegisterInfo() starts from 0. As a result, the first element in the returned RegisterInfo[] is a non-existent RegisterInfo.

LFT/AnswerFirst.sol:L233-L252

```
function withdrawNFTs() public whenNotPaused nonReentrant {

    uint256[] memory ids = new uint256[](_activityId);
    uint32 count=0;
    for(uint256 k=0; k<_activityId; k++){
        uint256 tokenId = _registerInfo[k][msg.sender].tokenId;
        if(tokenId > 0){
            IERC721(address(_erc721)).safeTransferFrom(address(this), msg.sender, tokenId);
            _registerInfo[k][msg.sender].tokenId=0;

            ids[count]=tokenId;
            count = count+1;
```

```
        }
    }

    require(count>0, "nothing to be withdrawed! the pledge hasn't expired yet");

    emit eWithdraw(ids, msg.sender, block.timestamp, _activityId);

}
```

Similarly, the for-loop in withdrawNFTs() starts at 0 instead of 1, resulting in unnecessary gas costs.

Recommendation

Consider fixing the loop logic in getAllRegisterInfo() and withdrawNFTs().

8. Unbounded loop in listMyNft()

Severity: Low	Category: Business Logic

Target:
- LFT/Avatar721.sol
- LFT/Adorn721.sol
- LFT/LAP.sol

Description

LFT/Avatar721.sol:L166-L176

```
function listMyNFT(address owner) public view returns (uint256[] memory tokens) {
    uint256 owned = balanceOf(owner);
    tokens = new uint256[](owned);
    uint256 start = 0;
    for (uint i=0; i<_currentIndex; i++) {
        if(_ownerships[i].addr == owner && !_ownerships[i].burned){
            tokens[start] = i;
            start++;
        }
    }
}
```

The listMyNFT() function in Avatar721 iterates over all the minted tokens to find the tokens that belong to the owner. However, there is no upper limit to the total supply of the Avatar721 tokens. Therefore, the loop in listMyNFT() is an unbounded loop, and listMyNFT() may fail due to running out of gas.

The same issue also applies to Adorn721 and LAP.

Recommendation

Consider using off-chain indexing to list all the NFTs that belong to a user.

9. Project can be started via setUserStart() in HotBuyFactoryV2	
Severity: Low	Category: Business Logic
Target: - LFT/HotBuyFactoryV2.sol	

Description

LFT/HotBuyFactoryV2.sol:L145-L151

```
function setProject(address nftContract, address costErc20) public onlyOwner{

    if(!_projcetSaleAmount[nftContract].contains(costErc20)){
        _projcetSaleAmount[nftContract].set(costErc20,0);
    }
    _projcetSwitch[nftContract][costErc20] = true;
}
```

The standard way to start a project in HotBuyFactoryV2 is to use setProject().

LFT/HotBuyFactoryV2.sol:L153-L155

```
function setUserStart(address nftContract, address costErc20, bool start) public onlyOwner {
    _projcetSwitch[nftContract][costErc20] = start;
}
```

However, the project can also be started by using setUserStart(nftContract, costErc20, true). But, the setUserStart() does not initialize the costErc20 information in _projcetSaleAmount.

As a result, if a project is started using setUserStart() instead of setProject(), the user can start minting using mintAdornWithEth() or mintAdorn(), but this project is not included in the ProjectInfo[] returned by getProjectInfo().

Recommendation

Consider decoupling the functionality of setProject() and setUserStart(). A project can be started using setProject(). So, if the usage of setUserStart() is only to disable the project, then a disableProject() function should be implemented to replace setUserStart().

10. Difficult to query the airDropId for an activity	
Severity: Low	Category: Auditing and Logging
Target: - LFT/AnswerFirst.sol	

Description

Each activity in AnswerFist is associated with an airDropId. The airDropId is the tokenId of the underlying erc1155 to be airdropped in the activity.

LFT/AnswerFirst.sol:L77-L82

```
struct ActivityInfo{
    uint256 registerCount;
    uint256 allPay;
    uint256 endTime;
    uint256 startTime;
}
```

However, the airDropId is not included in the ActivityInfo struct, therefore the getActivityInfo() will not return the airDropId for the activity.

LFT/AnswerFirst.sol:L47-L51

```
event eNewActivity(
    uint256 activityId,
    uint256 startTime,
    uint256 endTime
);
```

The airDropId is also not included in the eNewActivity event, so one cannot index the airDropIds for activities off-chain.

Recommendation

Consider adding airDropId to the ActivityInfo struct or the eNewActivity event so that people can query the airDropId for an activity.

2.3 Informational Findings

11. Mismatch between contract name and file name

Severity: Informational	Category: Code Quality
Target: - LFT/AvatarFactoryV2.sol	

Description

LFT/AvatarFactoryV2.sol:L42

```
contract AvatarFactory is Ownable, ReentrancyGuard {...}
```

The contract name is AvatarFactory while the file name is AvatarFactoryV2. It is best practice to name the contract with the same file name.

Recommendation

Consider changing the contract name from AvatarFactory to AvatarFactoryV2.

12. Incorrect OpenZeppelin version in package.json

Severity: Informational	Category: Configuration

Target:

- LFT/HotBuyFactoryV2.sol
- LFT/StoreFactoryV3.sol

Description

The EnumerableMap.UintToUintMap used in HotBuyFactoryV2 and StoreFactoryV3 is only available since OpenZeppelin v4.7.0. However, the specification is "@openzeppelin/contracts": "^4.6.0" in the package.json file.

Recommendation

Consider updating and freezing the version for @openzeppelin/contracts in package.json.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit b1ea906:

File	SHA-1 hash
LFT/Adorn1155.sol	a623ccf2ce9494c632564851ccf91e817f35cfea
LFT/Adorn721.sol	25be791c92acac427ee19db2d232607e624cb9b3
LFT/AnswerFirst.sol	cccb98d1585c0e4051f0debbf3bee3a4bcf50e71
LFT/Avatar721.sol	9d87bdf69fa604f108a33737d31ef40ae897682e
LFT/AvatarFactoryV2.sol	cc8bd704c11fe5319906dc055203042ad228907d
LFT/AvatarMintRule/BaseMintRule.sol	d6c2d333b914a35d01a0f22d6e9f825bac74f1a2
LFT/Exchange/LifeFormExchange.sol	fe1ec71826042ae24d026e7e6892a4b67dae53f0
LFT/Exchange/LifeFormRegistry.sol	35f10c7c16d23757d361cb7638ed78c342bc9d32
LFT/Exchange/LifeFormTransferProxy.sol	4d3329689165eb8d44b4f823b2f1709ef3b46f0f
LFT/HotBuyFactoryV2.sol	be5f6a8ae388705364663b149c431a1b43c29ced
LFT/Interface/IAdorn1155.sol	ba04173c034058aefc58e771765e631395b75f8c
LFT/Interface/IAdorn721.sol	d64beb7c84f4ff038495bb01b25a23cb9a0d23fc
LFT/Interface/IAvatar721.sol	689584a1579e49c7ec44cd86aeb764654c25a490
LFT/Interface/IAvatarMintRule.sol	2f4961b9014eea7e27007b01a0267e7e7f8faf97
LFT/LAP.sol	32425463b8003ff7b84752af01a7f2ec6d25626a
LFT/LBT.sol	20b590e749a27f17637b0cc82772f7cd76f14312
LFT/LifeformToken.sol	a3f4ca04c1e82a210a84490b1d0dc232769d4cf4
LFT/LuckyCkeckIn.sol	53ff554d299bc5026a88ecce9c49726f21fb106e
LFT/StoreFactoryV3.sol	f7d9e65757feec94d793681314a5c693afba4b74