Blaize.Security

July 19th, 2022 / V. 1.0

LIVE TREE
SMART CONTRACT AUDIT

TABLE OF CONTENTS

Audit rating 2

Technical summary 3

The graph of vulnerabilities distribution 4

Severity Definition 5

Auditing strategy and Techniques applied \ Procedure 6

Executive summary 7

Complete Analysis 8

Code coverage and test results for all files 36

Disclaimer 41

AUDIT RATING

LiveTree contract's source code was taken from the repository provided by the LiveTree team.

SCORE **9** /10

The scope of the project is **LiveTree** set of contracts:

1/ BuyInOfferData.sol

2/ HashtagNFTCollectiveERC721.sol

3/ HashtagNFTCollectiveFactoryProxy.sol

4/ HashtagNFTCollectiveGovernor.sol

5/ HashtagNFTCollectiveManager.sol

6/ HashtagNFTCollectiveResolver.sol

7/ HashtagNFTCollectiveTimelockController.sol

8/ HashtagNFTCollectiveTreasury.sol

9/ HashtagNFTCollectiveUtils.sol

10/ InitializedProxy.sol

11/ Settings.sol

Repository

https://github.com/livetreeShare/BlaizeAudit

Primary SHA256 (audited):

- 83be2f5a257bee042ffd9acc24a0ab4f297378321fd5e93281a62369e22ad195

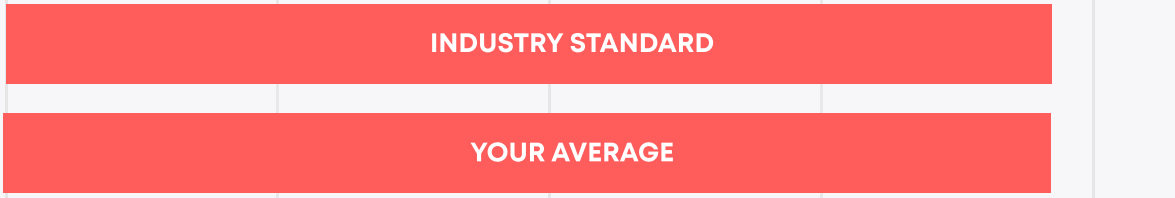
Final SHA256 (post-audit):

- 6401c30945bf241faa925a2f3f76b7fd773568f1dbf43431bf32deec182a5f1a

TECHNICAL SUMMARY

In this report, we consider the security of the contracts for LiveTree protocol. Our task is to find and describe security issues in the smart contracts of the platform. This report presents the findings of the security audit of **LiveTree** smart contracts conducted between **February 07th, 2022 - April 26th, 2022** (the first iteration) and **June 23rd, 2022 - July 15th, 2022** (the second iteration)

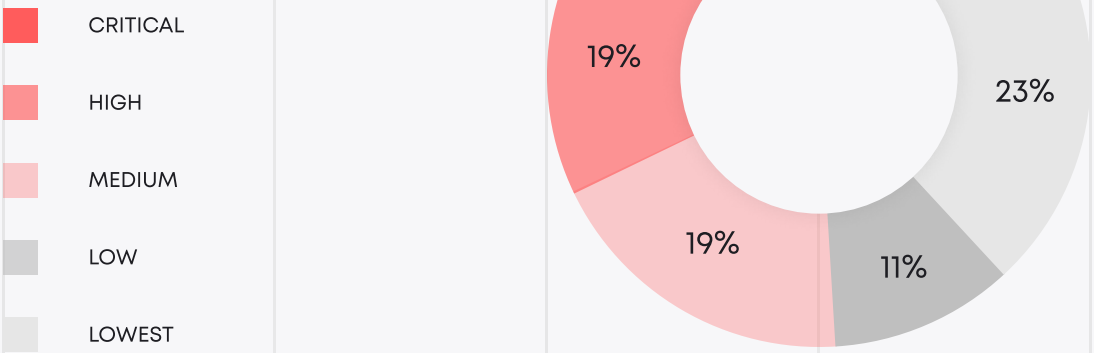
Testable code



The testable code has sufficient coverage, which corresponds to the industry standard of 95%.

The scope of the audit includes the unit test coverage, that bases on the smart contracts code, documentation and requirements presented by the LiveTree team. Coverage is calculated based on the set of Truffle framework tests and scripts from additional testing strategies. Though, in order to ensure a security of the contract Blaize.Security team recommends the LiveTree team put in place a bug bounty program to encourage further and active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of found issues and their severity. A total of 34 problems were found. All issues were fixed or verified by the LiveTree team.

	FOUND	FIXED/VERIFIED
Critical	10	10
High	7	7
Medium	7	7
Low	8	8
Lowest	4	4

SEVERITY DEFINITION

Critical

A system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Needs immediate improvements and further checking.

High

A system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge information or financial leak. Needs immediate improvements and further checking.

Medium

A system contains issues which may lead to mediumfinancial loss or users' private information leak. Needs immediate improvements and further checking.

Low

A system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Needs improvements.

Lowest

A system does not contain any issue critical to the secure work of the system, yet is relevant for best

AUDITING STRATEGYAND TECHNIQUES APPLIED \ PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

In our report we checked the contract with the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets best practices in efficient use of gas, code readability;

Automated analysis:

Scanning contract by several public available automated analysis tools such as Mythril, Solhint, Slither and Smartdec. Manual verification of all the issues found with tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. Checking smart contract logic and comparing it with the one described in the documentation.

EXECUTIVE SUMMARY

The contract contained several critical and high issues. Most of them were related to reentrancy attack, funds loss and access.

All other issues were connected to variables validation, gas optimization and Solidity version. Nevertheless, all issues were successfully fixed by Live Tree team and checked by Blaize Security Team.

The overall security is high enough though the code lacks of readability and documentation. Nevertheless, it performs all desired actions and has solid functionality.

	RATING
Security	9.3
Gas usage and logic optimization	9
Code quality	9
Test coverage	9
Total	9

COMPLETE ANALYSIS

CRITICAL ✔ **Resolved**

Reentrancy for the cashing out.

HashtagNFTCollectiveTreasury.sol, CashOutAndBurnERC20()
The function has no protection against call from the contract, thus, the ETH transfer may fall to the fallback function of the contract and the function can be called again.
The attack vector is next:
exploiter calls the function from the contract with part of tokens on it.
the CashOutAndBurnERC20() calculates the refund from the whole ercBalances[token] amount and sends it back;
Fallback function transfers more tokens from another account (approved before) to the exploit contract and calls CashOutAndBurnERC20() again
Refund is again calculated from the whole ercBalances[token] amount since it was not decreased yet
Exploiter can receive way more refunds than he should.

Recommendation:

Review function if it should be called from contracts, add reentrancy modifier from the ReentrancyGuard, move ETH transfer to the bottom of the function, so the whole storage is changed first.

CRITICAL ✔ Resolved

Pool can be withdrawn by anyone.

HashtagNFTCollectiveTreasury, BuyOutClaimRoyalty()
The function has no protection against call from the contract, thus, the ETH transfer may fall to the fallback function of the contract and the function can be called again. And since all storage changes and tokens transfer are in the wrong order at the end of the function, withdraw can be performed several times.

Recommendation:

Review function if it should be called from contracts, add reentrancy modifier from the ReentrancyGuard, move ETH transfer to the bottom of the function after the token transfer, so the whole storage is changed first.

CRITICAL ✔ Resolved

Commented modifier.

BuyInOffer.sol has all setters with commented out access modifiers, thus anyone can change the storage.

Recommendation:

Review the functionality and uncomment access modifier.

Post-audit.

Contract was deleted.

CRITICAL ✔ Resolved

Denial-of-Service attack.

HashtagNFTCollectiveERC721.sol, function wrapToken()
ID for token is generated by 'tokenIdsTracker' strictly in function wrapToken() and in function createCollectible() user can pass any id and mint token with such id.
Consider the next example:

1. Current index in 'tokenIdsTracker' is 2.
2. User creates an NFT with index 3 with function createCollectible().
3. Function wrapToken() is called, and will try to mint an NFT with the next index, returned by 'tokenIdsTracker' which would be equal 3.
4. Mint will fail because NFT with index 3 already exists and the function wrapToken() won't be executed.

Recommendation:

Consider removing an ability for users to pass any id in function createCollectible() and generate id with 'tokenIdsTracker' instead.

Post-audit.

tokenId is passed in wrapToken() as well, so that function execution cannot be blocked.

CRITICAL ✔ Resolved

Potential loss of ETH funds.

HashtagNFTCollectiveFactoryProxy.sol, function RefundMyExpiredOffer()
HashtagNFTCollectiveManager.sol, function RefundMyExpiredOffer()
User always gets a refund in Ether, even in case an ERC20 token address was provided when he created a buyIn offer. Thus, user is able to use a made up token with no value, or a token which price is lower than Ether and then collect Ether, stored in contract.

Recommendation:

Return a currency which was originally provided by a user.

CRITICAL ✔ Resolved

Anyone can collect Ether from canceled buyouts.

HashtagNFTCollectiveGovernor.sol, function refundBuyoutFee(). Function has no restrictions thus anyone can call this function and collected Ether from canceled buyout.

Recommendation:

Restrict function to being called only by the user who initiated a specified proposal with 'proposalId'.

CRITICAL ✔ Resolved

Contract doesn't compile.

ERC2771.sol.

1. Contract ERC2771.sol inherits ERC2771ContextUpgradeable.sol from OpenZeppelin library. In function __ERC2771Config_init() a function with name "__ERC2771Context_init()" which is not present in ERC2771ContextUpgradeable.sol. Instead, this contract uses a constructor for initialization.

HashtagNFTCollectiveGovernor.sol

1. Identifier "ERC20VotesUpgradeable" is used in lines 43, 57, however such contract or interface is not imported.
2. The following contracts are missing constructor, thus the cannot be compiled and deployed: ERC2771Config, ERCX, ERCX721fier, ERCXEnumerable, ERCXMetadata, ERCXFull, ERCXMintable, NFT, HashtagNFTCollectiveERC721, HashtagNFTCollectiveGovernor, HashtagNFTCollectiveTimelockController, HashtagNFTCollectiveManager, HashtagNFTCollectiveFactoryProxy, HashtagNFTCollectiveResolver, HashtagNFTCollectiveTreasury.

Recommendation:

1. Use a constructor instead of this function.
2. Import "ERC20VotesUpgradeable" in HashtagNFTCollectiveGovernor.sol.
3. Implement constructor or make sure that contracts can be compiled and deployed.

CRITICAL ✔ Resolved

Initiator of buy in offer is able to refund accepted offers.

HashtagNFTCollectiveManager.sol: function RefundMyExpiredOffer().
HashtagNFTCollectiveFactoryProxy.sol: functionRefundMyExpiredOffer().
This function refunds an offer which has expired, however there is no check that the offer was accepted, which means that the initiator of the offer has received a reward and is not supposed to get refunded in this case.

Recommendation:

Add a validation, that offer state is PENDING. This should be enough to ensure that accepted offers are not refunded.

Post-audit.

Validation was added.

CRITICAL ✔ Resolved

Anyone is able to mint nft with buy in offer and collect payment.

HashtagNFTCollectiveFactoryProxy.sol: function mint(), _serviceBuyInOffer().
When creating a buy in offer(function MakeBuyInOffer()), a creator address is passed, which seems to be the creator of the collective and address which is supposed to collect payment from the offer. However, after an offer is created, anyone is able to call mint() with provided itemId and offerId and collect reward(Lines 444 and 451, payment is sent to msg.sender and there is no check that msg.sender is creator).

Recommendation:

Add a validation, that in case buy in offer exists, only creator can execute mint() and collect payment.

Post-audit.

Offer functionality was removed from HashtagNFTCollectiveFactoryProxy.sol

CRITICAL ✔ Resolved

Ether is not transferred during function call.

HashtagNFTCollectiveManager.sol: function BuyOut().
Line 469-470, contract perform a call of function DepositRoyalty() from contract HashtagNFTCollectiveTreasury.sol, which is payable and checks that required amount of Ether is sent. However, the transferring of Ether during function call is absent. This causes revert “failure: msg.value != total royalties”.

Recommendation:

Transfer required Ether during contract call. Example: IHashtagNFTCollectiveTreasury(ISettings(settings).getTreasury()).DepositRoyalty{value: msg.value}(tokens, balances)

HIGH	✔ Resolved

Reentrancy when accepting buy in offer.

HashtagNFTCollectiveManager.sol: function AcceptBuyInOffer().
HashtagNFTCollectiveFactoryProxy.sol: function _serviceBuyInOffer().
External calls are performed before the storage variable buyInOffers[offerId].state is updated, which can cause a reentrancy attack(due to call in line 303). Issue is not marked as critical because in order to withdraw all the Ether, malefactor has to provide a corresponding amount of tokens(Line 295), however the reentrancy issue still remains.
Same happens in HashtagNFTCollectiveFactoryProxy.sol (Line 440).

Recommendation:

Either add a NonReentrant modifier or update buyInOffers[offerId].state before external calls are performed.

Post-audit.

NonReentrant modifier was added

HIGH ✔ Resolved

Deprecated Eth transfer.

HashtagNFTCollectiveTreasury
CashOutAndBurnERC20(), WithdrawFromTreasuryToAddress(), BuyOutClaimRoyalty()
Due to the Istanbul update there were several changes provided to the EVM, which made .transfer() and .send() methods deprecated for the ETH transfer. Thus it is highly recommended to use .call() functionality with mandatory result check, or the built-in functionality of the Address contract from OpenZeppelin library.

Recommendation:

Correct ETH sending functionality.

Post-audit.

Should also be changed in:
HashtagNFTCollectiveGovernor.refundBuyoutFee().

Post-audit.

It was changed in
HashtagNFTCollectiveGovernor.refundBuyoutFee().

HIGH ✔ Resolved

Manager is able to burn any NFT.

HashtagNFTCollectiveERC721, function unWrapToken().
Manager is able to burn any NFT, which was created with function wrapToken() without an approval of the owner of NFT.

Recommendation:

Add a validation that the owner of NFT approved the manager to burn his token.

HIGH	✔ Resolved

Caller of function is able to provide any address with ERC20 interface.

HashtagNFTCollectiveFactoryProxy.sol, function MakeBuyInOffer()
HashtagNFTCollectiveManager,sol, MakeBuyInOffer()
Caller of function MakeBuyInOffer() is able to provide any address within parameter “erc20Token” including malicious contract which imitates interface of ERC20.
This can cause to unexpected behavior and attacks such as re-entrancy, since later a function transfer() is executed with this address(in function _serviceBuyInOffer(), line 420).

Recommendation:

Validate than only valid ERC20 tokens can be passed.

Post-audit.

Added nonReentrant modifier, but a caller is still able to provide any address with ERC20 interface.

Post-audit.

It was verified by the client, that due to the design of the protocol, the ability of passing any ERC20 tokens must remain. The nonReentrant modifier is enough to mitigate any possible threats.

HIGH	✔ Rresolved

Functions are not restricted.

HashtagNFTCollectiveERC721.sol: functions setWrappedTokenURI(), setCollectiveURI().
The following functions are not restricted from being called by anyone, meaning that anyone can set storage information such as URI, preventing protocol from correct operation.

Recommendation:

Add restriction to the following functions to be called only by authorized addresses.

Post-audit.

Validations were added to both functions.

HIGH	✔ Resolved

Contract is missing implemented calls of other contracts.

1. HashtagNFTCollectiveERC721.sol: functions unWrapToken(), setRoyalties().HashtagNFTCollectiveERC721.sol has functions with modifier onlyManager() which verified that caller is manager. Manager is supposed to be contract HashtagNFTCollectiveManager.sol. However, the following functions of HashtagNFTCollectiveERC721 are not called within HashtagNFTCollectiveManager.sol, so that these functions potentially cannot be called.
2. HashtagNFTCollectiveManager.sol: function SetAssetCount().HashtagNFTCollectiveFactoryProxy.sol is missing implementation of call if function SetAssetCount() in HashtagNFTCollectiveManager.sol;

Recommendation:

Implement the calls of functions unWrapToken(), setRoyalties() in HashtagNFTCollectiveManager.sol, SetAssetCount() in HashtagNFTCollectiveFactoryProxy.sol.

Post-audit.

1. Functions calls are now performed within HashtagNFTCollectiveNFTVault.sol
2. Function SetAssetCount() was moved to HashtagNFTCollectiveNFTVault.sol and its call is implemented in HashtagNFTCollectiveFactoryProxy.sol.

HIGH ✔ Resolved

Setter is not restricted.

HashtagNFTCollectiveFactoryProxy.sol: function setEscrowMgr(), setSeedCToken().
Function which sets a storage variable with essential information should be restricted from being called by anyone.

Recommendation:

Add a restriction.

Post-audit.

Function setEscrowMgr() was removed, setSeedCToken() is restricted with onlyOwner modifier.

MEDIUM ✔ Resolved

Major gas optimization.

HashtagNFTCollectiveUtils.sol, concatAll()
Function is never used for the concatenation of more than 2 strings at once, thus it can be simplified.

Recommendation:

Eliminate gas spending.

MEDIUM ✔ Resolved

Unverified commission percentage.

HashtagNFTCollectiveManager.sol, initialize()
Function mints commission for the owner based on the value from the settings contract and divides it by 100. Though, the fee value is not verified in the Settings contract to feat these accuracy. Thus the logic of mint can be broken.
The same applies to the livetreePercentage value used in this initialization.

Recommendation:

Add verification of the owner fee value and livetreePercentage value into the Settings contract - initializer and setter functions. Also add validation for the sum of both amounts to not exceed 100%.

MEDIUM ✔ Resolved

Unify Solidity version.

For now contracts set utilizes Solidity 0.8.x and 0.7.x - though mixing of Solidity versions is not recommended. Also it is recommended to use the last stable version in the line, which is 0.8.11 for now.

Recommendation:

Unify Solidity version over contracts and use the latest one.

MEDIUM ✔ Resolved

Variables lack validation

HashtagNFTCollectiveFactoryProxy.sol, function MakeBuyInOffer()
HashtagNFTCollectiveManager, function MakeBuyInOffer() (only 'expiry' variable)

1. Function parameter 'percentage' should be validated not be greater than 100 because of the subtraction in line 280.
2. Function parameter 'expiry' should be validated to be greater than block.timestamp. Also consider adding constants with minimum and maximum BuyInOffer duration and validated 'expiry' against these constants as sanity checks since currently user is able to set any 'expiry' date, which can be equal to a too large or too small value.

Recommendation:

Add validations for function parameters.

MEDIUM ✔ Resolved

Manager contract has no interface to execute TimelockController function.

Contract HashtagNFTCollectiveTimelockController.sol contains functions which can be called only by manager.
Manager is set in the initialize() and equals msg.sender which is HashtagNFTCollectiveManager contract(TimelockController is deployed within function mintGovernor()).
Contract HashtagNFTCollectiveManager.sol has no calls of functions setExecutors() and setAdmins() from TimelockController, thus these functions can never be executed.

Recommendation:

Implement call of all onlyManager functions in HashtagNFTCollectiveManager.sol.

MEDIUM ✔ Resolved

Use SafeERC20 library.

HashtagNFTCollectiveFactoryProxy.sol, function _serviceBuyInOffer(), line 420.
ERC20 tokens should be transferred with 'safeTransfer' and 'safeTransferFrom'. SafeERC20 performs all the checks that tokens were transferred, including ono-standard ERC20 implementation(like in USDT tokens).

Recommendation:

Use 'safeTransfer' and 'safeTransferFrom' instead.

Post-audit.

SafeERC20 was added.

MEDIUM ✔ Resolved

Incorrect interface check for IERC721Metadata.

HashtagNFTCollectiveERC721.sol: function wrapToken().
In line 188 contract checks that "from" contracts supports interface "this.tokenURI.selector" and after that, performs a call of function from IERC721Metadata to "from" contract. It is very unlikely, that another contract will register interface "this.tokenURI.selector". In order, to correctly check, that contract supports interface of IERC721Metadata.

Recommendation:

Check, that contract supports interface of IERC721Metadata.
Example:
IERC165(from).supportsInterface(type(IERC721Metadata).interfaceId)

LOW	✔ Resolved

SafeMath usage can be omitted.

Contracts set utilizes Solidity 0.8.x, which has built in support of overflow and underflow warnings, thus SafeMath library can be omitted for gas savings and code simplification.

Recommendation:

Remove SafeMath library.

LOW	✔ Resolved

Unused storage.

Settings.sol. Since all storage variables are public, there is no need in getters, as all variables already can be accessed via autogenerated getters. Thus, they other getters just increase the contract size and add gas spendings during the call.

Recommendation:

Remove extra getters.

Post-audit.

Getters were not removed, however variables were made private.

LOW	✔ Resolved

Incorrect event info.

Settings.sol, setMaxCuratorFee() - incorrect storage parameter is sent vai event as a result of copy-paste mistake.

Recommendation:

Event should be emitted with maxCuratorFee.

Post-audit.

Function setMaxCuratorFee() was deleted.

LOW ✔ Resolved

Missing visibility identifier

HashtagNFTCollectiveTimelockController.sol missing visibility for manager variable.
HashtagNFTCollectiveResolver missing visibility for the factorProxy variable.
HashtagNFTCollectiveERC721 missing visibility for takenItems, licenseIsSet, governor variables.
HashtagNFTCollectiveFactoryProxy missing visibility for royaltyItemMgrSettings variable.
HashtagNFTCollectiveManager missing visibility for buyoutFee, assetCount variables.
Setting.sol, lines 58-78

Recommendation:

Set variable visibility.

Post-audit.

Also set visibility for storage variables in Settings.sol

LOW ✔ Resolved

Unused bool interface.

Settings.sol, setGovernorLogic(), setBuyInLogic(), setTreasury()
HashtagNFTCollectiveTreasury.sol, DepositRoyalty(), CashOutAndBurnERC20(), WithdrawFromTreasuryToAddress(), BuyOutClaimRoyalty()
HashtagNFTCollectiveResolver.sol, AddNftURIRecord()
HashtagNFTCollectiveManager.sol, timelockControllerLogic, buyoutFee, assetCount
Functions have a bool return interface, though the value is always true and it is actually not used within the contracts set. Remove it in order to decrease the contract size and gas spendings.

Recommendation:

Remove unused bool return interface.

LOW ✔ Resolved

Mapping values are never written.

HashtagNFTCollectiveERC721. Mapping takenItems.
Values from mapping are read, but never written. Values from mapping are read in function createCollectible to verify that NFT has not been minted, however after minting, mapping is not updated.
Also, ERC721 already performs checks that NFT doesn't exist during minting, thus additional validations are not necessary.

Recommendation:

Remove mapping or make sure to update its values.

Post-audit.

Updating of mapping was added.

LOW ✔ Resolved

Value "escrowManager" is never initialized

HashtagNFTCollectiveFactoryProxy.sol
Value "escrowManager" is never initialized, and there is not setter for this variable.

Recommendation:

Add setter for "escrowManager".

LOW	✔ Resolved

Unreachable branches.

HashtagNFTCollectiveFactoryProxy.sol

1. _validateMintParams(), line 269. Due to previous require(where expire is checked) this require cannot revert, since in case this statement reverts, previous statement also reverts.
2. _serviceBuyInOffer(), line 421. Can never revert because statement is checked before function is executed(Lines 409,833).
3. _serviceBuyInOffer(), line 428. Branch can never be entered because in case statement in "if" mush be false, it will trigger require in function _validateMintParams() (Line 265) which is executed before _serviceBuyInOffer() is executed.
4. _serviceBuyInOffer(), line 432. Probably unreachable, because if offer was served, it is not possible to mint token with same id and trigger _serviceBuyInOffer() and this require statement. In case, offer is expired, it will revert in function_validateMintParams().
5. _validateMintParams(), line 288. In case, this require has to revert, the previous one will also revert, thus making require in line 288 unreachable.
6. _calculateEscrowSupply(). Function is internal and is not called within other functions.

HashtagNFTCollectiveManager.sol

1. modifiers onlyEscrowMgr(), assetsNotSold(). Modifiers are never used.
2. Function SetAssetCount(). Function has modifier onlyProxyFactory which checks that caller is factory, however there is not interface for calling this function in HashtagNFTCollectiveFactoryProxy.sol.

HashtagNFTCollectiveGovernor.sol

1. _cancel(). Function is internal and is not called within other functions.
2. refundBuyoutFee(). Function can not be executed, since the contract does not have the functionality to assign the status of canceled to the proposal.

Recommendation:

Use the _cancel() function (HashtagNFTCollectiveGovernor.sol) to add the ability to assign the status of canceled to the proposal. It will also provide an opportunity to test refundBuyoutFee(). Remove other unreachable branches.

LOWEST	✔ Resolved

Unclear contract purpose.

InitializedProxy.sol
The contract is named as Proxy and aimed to serve as a proxy for another contract. Though it has an immutable implementation, thus it cannot be upgraded in any way. So, its necessity should be verified.

Recommendation:

Verify the functionality and remove unnecessary contract.

LOWEST	✔ Resolved

Constructor can be removed.

BuyInOffers.sol, HashtagNFTCollectiveTimelockController.sol, HashtagNFTCollectiveManager.sol
The contract is upgradeable, thus the constructor can be removed.

Recommendation:

Remove unused constructor.

LOWEST	✔ Resolved

Commented code.

There are a lot of commented code strings across multiple contracts. Pre-production code should not contain unimplemented or unfishied logic.

Recommendation:

Remove commented code or finish the logic.

LOWEST	✔ Resolved

Commented functions.

HashtagNFTCollectiveManager.sol
Functions detail(), SetExecutors(), SetAdmins() are commented out. Pre-production code should not contain commented parts of code.

Recommendation:

Either remove these functions or uncomment the.

Post-audit.

Commented functions were removed.

	BuyInOfferData.sol	HashtagNFTCollectiveERC721.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/ Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions / Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

	HashtagNFTCollectiveFactoryProxy.sol	HashtagNFTCollectiveGovernor.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/ Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions / Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

	HashtagNFTCollectiveManager.sol	HashtagNFTCollectiveResolver.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/ Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions / Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

	HashtagNFTCollectiveTimelockController.sol	HashtagNFTCollectiveTreasury.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/ Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions / Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

	HashtagNFTCollectiveUtils.sol	InitializedProxy.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/ Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions / Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

	Settings.sol
✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/ Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions / Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES BY BLAIZE.SECURITY TEAM

HashtagNFTCollectiveERC721

- ✓ prevents non-manager from setting of royalties (40ms)
- ✓ reverts when creation of a collectible if an uncorrect URL
- ✓ reverts when creation of a collectible if a used item ID (79ms)
- ✓ prevents someone who is not the proxy factory from setting of the manager
- ✓ prevents non-governor from setting of a license
- ✓ wraps a token when a contract storing an NFT supports ERC721 metadata interface
- ✓ wraps a token when a contract storing an NFT that does not support ERC721 metadata interface (132ms)
- ✓ sets a URI for a wrapped token (113ms)
- ✓ reverts when returning of royalty info for a non-existent token

Contract: HashtagNFTCollectiveFactoryProxy

- ✓ Should not mint if erc20 name not provided
- ✓ Should not mint if erc20 symbol not provided
- ✓ Should not mint if metadata url not provided
- ✓ Should revert getting royalty item manager settings if id > vault count
- ✓ Should pause and unpause
- ✓ Should set resolved
- ✓ Should mint for user and transfer collective (261ms)

HashtagNFTCollectiveGovernor

- ✓ returns the quorum number

- ✓ supports the interface of the ERC165Upgradeable contract
- ✓ returns a manager address
- ✓ does nothing when request of lend assets
- ✓ does nothing when request of a retrieve asset

Request of a buyout

- ✓ reverts when request of a buyout if a fee is not equals sent ether

Request of a buyout

- ✓ reverts when refund of a buyout fee if a wrong depositor
- ✓ reverts when refund of a buyout fee of a porposal which is not cancelled

Contract: HashtagNFTCollectiveManager

- ✓ Should revert mintTo if supply is 0 (213ms)
- ✓ Should revert mintTo is caller is not proxy factory
- ✓ Should revert make buy in offer if expiry < block.timestamp
- ✓ Should revert make buy in offer if amount != msg.value
- ✓ Should revert make buy in offer if offer is not new (47ms)
- ✓ Should approve burn
- ✓ Should not accept buy in offer if it doesn't exist
- ✓ Should revert accepting if offer expired (52ms)
- ✓ Should accept buy in offer (89ms)
- ✓ Should not refund if offer doesn't exist
- ✓ Should not refund if item id and offer id mismatch (59ms)
- ✓ Should not refund if offer not expired (67ms)
- ✓ Should not refund offer more than once (88ms)
- ✓ Should return detai
- ✓ Should revert mint governor if caller is not proxy factory

- ✓ Should let governor set buyout (64ms)
- ✓ Should get settings proxy address
- ✓ Should process buyout
- ✓ Should revert treasury burn if caller is not treasury
- ✓ Should set proposers
- ✓ Should set executers (40ms)
- ✓ Should set admins
- ✓ Should not transfer ownership if called not by admin
- ✓ Should not make buy in offer in self token
- ✓ Should not transfer stake if called not by treasury
- ✓ Should not approve burn if balance of msg.sender != totalSupply (39ms)

HashtagNFTCollectiveTreasury

- ✓ initializes
- ✓ reverts when cashing if an account is not token holder (60ms)

Withdrawal from a treasury to an account

- ✓ withdraws
- ✓ reverts when withdrawal if a false governor
- ✓ reverts when withdrawal if an uncorrect amount

Contract: InitializedProxy

- ✓ Should revert constructor if initialize call failed (88ms)
- ✓ Should set new logic (176ms)
- ✓ Should not let not admin set new logic (179ms)

Settings

- ✓ should revert when summary is greater than 100 percents
- ✓ should return correct branch buyer username
- ✓ should return correct buyer
- ✓ should return correct escrow manager
- ✓ Someone who is not factory should not set buy in logic
- ✓ Factory should set and get correct buy in logic
- ✓ should return correct livetree

HashtagNFTCollectiveGovernor

- ✓ should allow SetWithdrawalFromTreasuryToAddress (199ms)
- ✓ should allow SetBuyOutAmount (156ms)
- ✓ should allow SetLicense (139ms)
- ✓ should allow Buyout (151ms)

HashtagNFTCollectiveManager BuyIn

- ✓ should register buyIn offers (239ms)
- ✓ should accept existing buyIn offers (256ms)
- ✓ should fail on attempt to accept accepted offer (239ms)
- ✓ should release funds to buyer when offer expired (243ms)
- ✓ should throw error on attempt to release un-expired offer (230ms)

HashtagNFTCollectiveTreasury

- ✓ should deposit royalties
- ✓ should check that royalties value equals sent msg.value
- ✓ should get treasury balance
- ✓ should return royalties for ERC20 burnt (122ms)
- ✓ should return royalties for ERC20 bought-out (103ms)

HashtagNFTCollectiveFactoryProxy

- ✓ should have correct Settings
- ✓ should create RoyaltyItemManager with configured settings (163ms
- ✓ should mint NFTs (228ms)
- ✓ should fail NFT mint if token symbol,name pair exists (174ms)
- ✓ should mint RoyaltyItemManager (226ms)
- ✓ should register buyIn offers
- ✓ should accept existing buyIn offers (274ms)
- ✓ should refund expired buyIn offers
- ✓ should import existing NFT's (419ms)

- ✓ should check if Collective exists using the HashtagNFTCollectiveResolver
- ✓ should check if Collective exists using the HashtagNFTCollectiveResolver before minting Collective (185ms)

90 passing (26s)

The total coverage represented by native unit-tests and testset prepared by auditors is sufficient for the security qualification. Due to the size of the contracts (which exceeds 24kB without the optimization) the coverage cannot be measured by the Hardhat or Truffle tools expicitely. Though it can be measured indirectly, the fact verified by the auidtors team.

DISCLAIMER

The information presented in this report is an intellectual property of the customer including all presented documentation, code databases, labels, titles, ways of usage as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else requirements and be fully secure, complete, accurate and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.