HashEx
BLOCKCHAIN SECURITY

Liquid Apes

smart contracts final audit report

January 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the ApeDAO team to perform an audit of their smart contract. The audit was conducted between 2023-01-08 and 2023-01-10.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @iotapes/liquid-apes Github repository after the d0563c6 commit.

Some of the contracts are designed to be deployed behind a proxy, meaning their implementations can be updated. Users must pay attention and do own research before using such contracts.

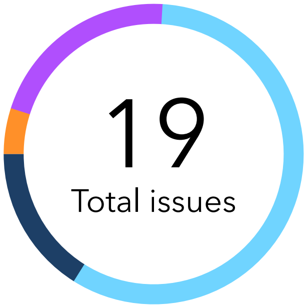
2.1 Summary

Project name	Liquid Apes
URL	https://apedao.finance
Platform	Shimmer Network
Language	Solidity

2.2 Contracts

Name	Address
LilApeNFT	
OGApeNFT	
ApeDAOVaultUpgradeable	
PausableUpgradeable	
IApeDAOVault	

3. Found issues



High	1 (5%)
Medium	4 (21%)
Low	11 (58%)
Info	3 (16%)

C1. LilApeNFT

ID	Severity	Title	Status
C1-01	Low	Default visibility of a variable	Resolved
C1-02	Low	Documentation inconsistency	Resolved

C2. OGApeNFT

ID	Severity	Title	Status
C2-01	Low	Default visibility of a variable	Resolved
C2-02	Low	Same base URI link in tokens	Resolved

C3. ApeDAOVaultUpgradeable

ID	Severity	Title	Status
C3-01	High	Deposited and redeemed assets may differ	Resolved
C3-02	Medium	Lack of checks of input parameters	Resolved
C3-03	Medium	Usage of pre-release library version	Resolved
C3-04	Medium	Insecure random	Acknowledged
C3-05	Low	Lack of events in governance functions	Resolved
C3-06	Low	Redundant unitialization	Resolved
C3-07	Low	Division before multiplication	Resolved
C3-08	Low	Excessive inheritance	Resolved
C3-09	Low	Variables with default visibility	Resolved
C3-10	Low	Gas optimizations	Resolved
C3-11	Low	Possible error message decoding fail	Resolved
C3-12	Info	Insufficient documentation of internal functions	Resolved
C3-13	Info	Usage of integers for enumerated values	Resolved
C3-14	Info	Tokenomics issues	Resolved

C5. IApeDAOVault

ID	Severity	Title	Status
C5-01	Medium	No collection is specified in events	Resolved

4. Contracts

C1. LilApeNFT

Overview

An implementation of the ERC-721 NFT token standard built with OpenZeppelin templates. LilApeNFT token has a fixed total supply, and supports enumeration and royalties.

Issues

C1-01 Default visibility of a variable — Low — Resolved

No explicit visibility is specified for the `maxSupply` variable. Default `internal` visibility is used. We recommend always specifying a visibility modifier even if it matches the default one.

C1-02 Documentation inconsistency — Low — Resolved

According to the NatSpec description, the token symbol should be LAPE, but LILAPE is used in the constructor section.

C2. OGApeNFT

Overview

An implementation of the ERC-721 NFT token standard built with OpenZeppelin templates. OGApeNFT token has a fixed total supply, and supports enumeration and royalties.

Issues

C2-01 Default visibility of a variable ● Low ✓ Resolved

No explicit visibility is specified for the `maxSupply` variable. Default `internal` visibility is used. We recommend always specifying a visibility modifier even if it matches the default one.

C2-02 Same base URI link in tokens ● Low ✓ Resolved

The base URI IPFS link for OGApeNFT is identical to the LILAPE token. Since both tokens share their enumeration from 1 to 1074 token IDs, collisions are inevitable.

C3. ApeDAOVaultUpgradeable

Overview

An implementation of the ERC-20 token standard built with OpenZeppelin templates. ApeDAOVaultUpgradeable can be minted in exchange for supported NFT tokens (LilApeNFT and OGApeNFT) with an updatable exchange rate. The contract is designed to be deployed behind a proxy, meaning its implementation can be updated. Users must pay attention and do their own research before using this contract.

Issues

C3-01 Deposited and redeemed assets may differ ● High ✓ Resolved

The `setOgApesAddress()` and `setLilApesAddress()` functions allow the owner to change the addresses of NFT assets, which may cause a situation of redeeming an unexpected token, not the one that was deposited. Moreover, setting an EOA as asset address will result in the possibility of unlimited minting.

Recommendation

We strictly recommend denying NFT addresses changes in case of non-zero Vault balance. We also recommend verifying the new addresses by checking their interface (see EIP-165) or at least by checking their `extcode` size.

C3-02 Lack of checks of input parameters

Medium · Resolved

The functions `setManager()`, `setFeeReceiver()`, `setOgApesAddress()` , and `setLilApesAddress()` do not validate input parameters. In case a zero address is passed to one of the two latter functions an attacker can mint the ApeDAOVault tokens.

Recommendation

Add non-zero checks for functions' input parameters.

C3-03 Usage of pre-release library version

Medium · Resolved

The project uses a pre-release version of the library @openzeppelin/contracts-upgradeable (4.8.0-rc.1). The release version is available.

Recommendation

We recommend using only release versions of libraries. Upgrade to a stable version of the library.

C3-04 Insecure random

Medium · Acknowledged

A random number for redeeming NFT can be computed with high probability in the `getRandomTokenIdFromVault()` function. There's no reputable way of obtaining the randomness on-chain, so the industry standard is using VRF oracles or gaining the seed from the project owners if users trust them.

Recommendation

Use oracles like Chainlink VRF to get a secure random.

Team response

At the moment of writing this response Chainlink VRF is not available on the EVM network that we are planning to deploy our contracts to, but in case this changes we will consider updating this mechanism. Regardless of that, there is no direct benefit in knowing which NFT will be minted from the vault since our DAO is setup in a way that NFT rarity doesn't affect voting power.

C3-05 Lack of events in governance functions

Low Resolved

The functions `setManager()`, `setOgApesAddress()`, `setLilApesAddress()` don't emit events.

Functions that update system parameters should emit corresponding events to ease the off-chain tracking of changes history.

C3-06 Redundant unitialization

Low Resolved

Double initialization of the Ownable contract is performed once directly and the second time inside the Pausable contract initialization.

```
function __ApeDAOVault_init(
    string memory _name,
    string memory _symbol,
    address _ogApesSCAddress,
    address _lilApesSCAddress,
    address _apeDAOVaultFeeReceiver
) public virtual override initializer {
    __Ownable_init();
    __Pausable_init();
    ...
}
```

Recommendation

Remove the redundant `__Ownable_init();` call in the function.

C3-07 Division before multiplication

Low | Resolved

In several parts of the code division before multiplication is made (L281, L284, L319, L327). This increases calculation errors.

```
totalFee =
    (((baseOGApe * specificIds.length) / 100) * _targetRedeemFee) +
    (((baseOGApe * (amount - specificIds.length)) / 100) *
        _randomRedeemFee);
```

C3-08 Excessive inheritance

Low | Resolved

The contract inherits both from the OwnableUpgradeable and PausableUpgradeable contracts, but the PausableUpgradeable contract itself already inherits from the OwnableUpgradeable contract.

Recommendation

Remove inheritance from the OwnableUpgradeable contract.

C3-09 Variables with default visibility

Low | Resolved

No explicit visibility is specified for `burnedERC20Tokens`, `randNonce`, `baseOGApe`, `baseLilApe`, `maxSupply`, `vaultFees`, `ogApesHoldings`, `lilApesHoldings` variables. Default `internal` visibility is used. We recommend always specifying the visibility modifier even if it matches the default one.

C3-10 Gas optimizations

Low Resolved

1. The VaultFees structure could be packed into a single 256 bit by storing fees into smaller uint variables.

```
struct VaultFees {
    /**
     * @dev The fee for minting Ape NFTs into the Vault.
     */
    uint256 mintFee;
    /**
     * @dev The fee for redeeming Ape NFTs randomly.
     */
    uint256 randomRedeemFee;
    /**
     * @dev The fee for redeeming specific Ape NFTs.
     */
    uint256 targetRedeemFee;
}
```

2. Check of the `collectionId` parameter is duplicated in the `mint()` and `mintTo()` functions.

Recommendation

1. Pack variables in one slot:

```
struct VaultFees {
    uint64 mintFee;
    uint64 randomRedeemFee;
    uint64 targetRedeemFee;
}
```

2. Remove unnecessary checks.

C3-11 Possible error message decoding fail

Low   Resolved

Returning error messages may be failed in the `transferERC721()` and `transferFromERC721()` functions if they're blank or less than 64 bytes.

```
function transferFromERC721(
    address assetAddr,
    uint256 tokenId,
    uint256 collectionId
) internal virtual {
  ...
    (bool success, bytes memory resultData) = address(assetAddr).call(data);
    require(success, string(resultData));
}
```

Recommendation

Bubbling up error code example could be found in the @openzeppelin/contracts/utils/Address.sol library, which also supports bubbling custom errors. Another way to address the issue is by using interfaces to avoid low-level calls.

C3-12 Insufficient documentation of internal functions

Info   Resolved

It should be explicitly documented that internal functions `receiveNFTs()` and `withdrawNFTsTo()` do not validate the collection ID. A wrong collection ID results in a successful call to `address(0)` in the `receiveNFTs()` and `withdrawNFTsTo()` functions. The safety check requirement must be mentioned explicitly in the NatSpec description.

C3-13 Usage of integers for enumerated values

Info   Resolved

To determine which collection should be used an integer of value 0 for OgApe or 1 for LilApe is used. The code readability can be improved by using enums instead of an integer.

Recommendation

We recommend using an enum for collection identifier to reduce possibility of errors in future updates

C3-14 Tokenomics issues

Info Resolved

1. If ApeDAOVault tokens are burnt with the `burnERC20Tokens()` function, it effectively means that some NFTs sent to the contract would be locked as there won't be enough ApeDAOVault tokens to withdraw all tokens.

2. A user that mints tokens by sending NFTs to the contract won't get enough tokens to redeem them. For example, the user mints 500 ApeDAOVault tokens by sending an OG Ape NFT to the contract upon calling the `mint()` function. The mint commission of 10% is taken, so the user receives 450 ApeDAOVault tokens to his balance. To redeem an OG Ape token he needs at least 500 tokens, so he needs to obtain them somewhere.

Team response

The reason why we are using burnERC20Tokens() function is because of the way we are deploying the Lil' Apes collection. It is important to note that this function can only be called once, and this will be during SC deployment (that's why we use the burnedERC20Tokens boolean, so that it can only be called once).

Essentially the idea is that initially, once the Vault is deployed, it will contain all Lil' Apes in it by design, so that these can be redeemed/bought by users. Because of that, once we deploy the ApeDAO Vault, we'll need to mint the Lil' Apes into the Vault, and this makes the vault minting $APEin tokens (ERC20 token from the Vault) as per design. So, the idea is that, once all Lil' Apes are minted into the Vault, our deployment script will run that burnERC20Tokens() function to burn all created $APEin tokens. This way, the Vault will be holding all Lil' Apes, and the number of minted $APEin tokens is 0.

As mentioned, this function can only be called once. After deployment, no one, not even the owner of the Vault will be able to call it again. Also, note that, no one will be able to redeem/

buy any NFT from the Vault until we finish the deployment, as the only way to redeem an NFT is via $APEin token, which can only be minted and received by minting/selling an OG NFT into the Vault. We will be distributing the OG NFT to the holders after everything has been deployed, so no one will be able to mint/sell their NFTs into the Vault until we complete deployment (i.e. after calling the burnERC20Tokens() function).

The mint commission of 10% goes to the DAO treasury so the tokens are not lost and $APEin token will have liquidity on a DEX/AMM for users to obtain them. The reasoning for doing it this way is to expand the number of ApeDAO NFTs while preserving the initial voting power of OG Ape holders - detailed explanation is available in our project's documentation.

C4. PausableUpgradeable

Overview

A custom contract that inherits the Ownable authorization model from the OpenZeppelin library and allows setting a mapping of boolean flags for different types of pause statuses.

C5. IApeDAOVault

Overview

Interface for the ApeDAOVaultUpgradeable contract.

Issues

C5-01 No collection is specified in events — Medium — Resolved

The events `Minted` and `Redeemed` do not specify the collection id. The same events will be emitted when tokens with the same id are deposited or withdrawn from the contract.

```
event Minted(uint256[] nftIds, address to);
```

```
    event Redeemed(uint256[] nftIds, uint256[] specificIds, address to);
```

Recommendation

Add collection id parameter to these events.

5. Conclusion

1 high, 4 medium, 11 low severity issues were found during the audit. 1 high, 3 medium, 11 low issues were resolved in the update.

The ApeDAOVaultUpgradeable contract is highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx

BLOCKCHAIN SECURITY