Security audit

Clopr

Security review

Date : November 16th, 2023

Summary

I. Introduction

1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

2. Methodology

a. Preparation

The project is an NFT based project where NFTs can interact with others to create stories. CloprBottle NFTs enhance the value of any NFT by enabling ownership of customized derivative assets across ecosystems.

A first technical meeting was held on 31/10/2023. It allows technical teams to explain the contract workflow, to define the exact scope, and start the audit process. Regarding the short deadline, Black Paper team was continuously interacting with the technical team to allow the best result in terms of security.

b. Review

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- Money-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead Of Transfer
- Costly Loop

- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report which contains for each vulnerability :

- Explanation
- Severity score
- How to fix it / Recommendation

Here are severity score definitions.

Critical	A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.
Major	A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.
Medium	A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.
Low	A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.

Informational	An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

The following smart contracts are considered in scope:

- CloprBottles.sol
- CloprStories.sol
- StoryPotion.sol
- ICloprBottles.sol
- ICloprStories.sol
- IPotionDelegatedFillContract.sol
- IStoryPotion.sol

Other libraries and smart contracts are considered safe.

II. Vulnerabilities

Critical

1 critical severity issue was found:

- CRIT-1 (RESOLVED) Front-running presale with SP consumption

Major

2 major severity issues were found:

- MAJ-1 (MONITORED) UNFT sale can be front run with SNFT burn
- MAJ-2 (RESOLVED) Unsafe external call

Medium

3 medium severity issue were found:

- MED-1 (MONITORED) Mitigating signature reuse vulnerability
- MED-2 (RESOLVED) ERC Interface support
- MED-3 (RESOLVED) ERC4906 security considerations

Low

7 low severity issues were found:

- LOW-1 (MONITORED) Missing protection against wrong transfer recipient
- LOW-2 (MONITORED) CloprBottles contract address should be immutable
- LOW-3 (RESOLVED) Missing input checks
- LOW-4 (RESOLVED) Enhancing delegatecash usage in burnAndGrowStory function
- LOW-5 (RESOLVED) Useless type conversion
- LOW-6 (RESOLVED) Preventing self-burn NFT in burnAndGrowStory function
- LOW-7 (RESOLVED) _beforeTokenTransfers override

Informational

6 informational severity issues were found:

- INF-1 (RESOLVED) Implicit visibility for MODIFY_FILL_PRICE_ROLE
- INF-2 (RESOLVED) Function order in contracts structure
- INF-3 (MONITORED) Architecture of StoryPotion contract
- INF-4 (RESOLVED) tokenURI function comment issue
- INF-5 (RESOLVED) Type consistency improvement in getFillPrice function
- INF-6 (RESOLVED) Variable optimization and readability

CRIT-1 (RESOLVED) Front-running presale with SP consumption

Impact: Critical

Description:

The technical team of the project has already identified a vulnerability where SP (Story Potion) is consumed just before selling the NFT, leading to potential malicious activities such as front-running and offers acceptance after emptying the bottle.

This vulnerability results in a mismatch between the buyer's payment and the actual value of the empty bottle. Letting the responsibility of buyers to be careful is not possible since they currently can't protect themselves from this vulnerability.

Note that this issue is also impacting the fill before selling. Because the value of the bottle is expected to increase in this specific case, we consider it less important.

Recommendation:

We recommend to implement a transfer function restriction disallowing transfers for a specific duration or number of blocks after emptying a bottle.

A technical solution could be to add a `lastEmptyBlock` variable in the `BottleInformation` struct, and update it each time the bottle is emptied. For each `transferFrom`, check that `block.number - lastEmptyBlock > x`, where x is the number of blocks where we assume it is the responsibility of users.

```
struct BottleInformation {
    uint16 potionId;
    uint8 potionFill;
    bool delegatedFillLevel;
    uint48 stakingTime;
    uint24 numberDrinks;
    uint256 lastEmptyBlock;
}
```

```
    function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public payable override(ERC721A, IERC721A) {
        if (msg.sender != from && bottles[tokenId].stakingTime > 0)
            revert CantTransferStakedBottle();

        require(block.number - bottles[tokenId].lastEmptyBlock > NUMBER_OF_BLOCKS_AFTER_EMPTY, "The
potion has been emptying during last blocks");

        ERC721A.transferFrom(from, to, tokenId);
    }
```

To prevent the case where the bottle is filled up, the `lastEmptyBlock` variable can be extended to `lastEmptyOrFillBlock`.

Status:

The fix is implemented, following recommendations.

MAJ-1 (MONITORED) UNFT sale can be front run with SNFT burn

Impact: Major

Description:

The technical team has already identified a vulnerability where the SNFT (Story NFT) is burned before selling the UNFT (underlying NFT). This loophole enables a user to retain the SNFT's value by transferring its pages to another story owned by a different UNFT and subsequently selling the UNFT at a premium, falsely implying that the SNFT is included in the purchase.

In scenarios where an NFT with an attached story is listed for sale on a marketplace, users naturally expect to receive the associated story upon purchase. However, the seller can exploit the system by front-running the buyer's transaction, burning the history attached to the NFT. Consequently, the buyer is left with only the NFT, devoid of any accompanying history.

Recommendation:

We recommend developing a custom marketplace with an Escrow contract allowing the buyer to specify whether the UNFT should come with a story or not. Upon execution, the escrow contract can verify the UNFT's state, as requested by the buyer, and revert the transaction if the conditions are not met.

Status:

The fix will be included in the future. For now, there is no escrow contract/marketplace to prevent front-running on sale.

MAJ-2 (RESOLVED) Unsafe external call

Impact: Major

Description:

In the current implementation, there is a potential vulnerability where a malicious user can send a SNFT to a smart contract (and his underlying NFT). For example, this smart contract can be an escrow contract for selling NFT. Even if the escrow smart contract becomes the owner, thye malicious user can still burn it and grow another story within the same transaction. In the case of an escrow contract or many other usages, this is a major source of vulnerability.

This behavior arises from the external calls in the `burnAndGrowStory` function, and the ownership check which is not performed just before the burn operation (ie using `_burn` instead of `burn`).

Here is an example scenario.

Initial Conditions

- The malicious user owns a Story NFT (SNFT) represented by burnedTokenId, we will call it “burnedToken”
- He deploys a malicious contract with a malicious ownerOf function. When the ownerOf function is called, it transfers the “burnedToken” to the escrow contract.

Transaction Execution

1. The malicious user starts the transaction, calling the burnAndGrowStory function with burnedToken as the first token to burn and the malicious contract as the extended token.

2. In the `burnAndGrowStory`, it first checks if the malicious user is the owner of the burnedToken.

```
        if (
         ownerOf(burnedTokenId) != requester || //<--- here
         ownerOf(extendedTokenId) != requester
    ) revert DontOwnStory();
```

3. It checks if the malicious user is the owner of the extended token, calling the malicious contract at the same time.

```
        if (
         ownerOf(burnedTokenId) != requester ||
         ownerOf(extendedTokenId) != requester //<--- here
      ) revert DontOwnStory();
```

4. burnedToken is transferred to the escrow contract.
5. The escrow contract sends tokens in exchange for the burnedToken.
6. burnedToken is burned.

```
        _burn(burnedTokenId);
```

Note that this malicious code is not in the burnedToken contract, but in the extendedToken. The escrow contract has no way to prevent it.

Recommendation:

To mitigate this issue, we recommend to use the `burn` instead of the `_burn` function since it is unsafe to assume the requester is the owner of the burnedToken during all the execution.

Status:

The technical team opted to swap `ownerOf(burnedTokenId)` and `ownerOf(extendedTokenId)`. This fix is also valid.

MED-1 (MONITORED) Mitigating signature reuse vulnerability

Impact: Medium

Description:

In CloprBottles contract, the `_isWhitelisted` function currently has a vulnerability in its content hashing mechanism, potentially allowing users to reuse signatures across different contracts.

```
function _isWhitelisted(
    uint256 mintPhaseIndex,
    bytes calldata signature
) internal view returns (bool) {
    bytes32 hash_ = ECDSA.toEthSignedMessageHash(
        keccak256(abi.encodePacked(msg.sender, mintPhaseIndex))
    );

    return hasRole(GRANT_WHITELIST_ROLE, ECDSA.recover(hash_, signature));
}
```

Recommendation:

Replace the existing content hashing mechanism with a more secure version, including the contract address in the hashed content.

```
function _isWhitelisted(
    uint256 mintPhaseIndex,
    bytes calldata signature
) internal view returns (bool) {
    bytes32 hash_ = ECDSA.toEthSignedMessageHash(
        keccak256(abi.encodePacked(msg.sender, mintPhaseIndex, address(this)))
    );

    return hasRole(GRANT_WHITELIST_ROLE, ECDSA.recover(hash_, signature));
}
```

Status:

This issue has chosen to be ignored in order to save gas during function execution. A lot of attention will be applied on all signatures by modifying wallets for each mint phase.

LOW-1 (MONITORED) Missing protection against wrong transfer recipient

Impact: Low

Description:

The contracts StoryPotion.sol, CloprStories.sol, and CloprBottles.sol currently use the internal function `_mint` of the ERC721 standard. However, this function assumes the receiver is either an Externally Owned Account (EOA) or a contract specifically designed to receive NFTs. In cases where the recipient is a contract not built to handle NFTs, the NFT faces permanent loss.

In CloprBottles.sol:

```
1426        _mint(to, quantity);

1554        _mint(msg.sender, quantity);
```

In CloprStories.sol:

```
1280        _mint(requester, tokenId);
```

In StoryPotion.sol:

```
156         _mint(msg.sender, 42);
```

Recommendation:

To prevent irreversible NFT loss and bolster security, it is recommended to replace the usage of `_mint` with `_safeMint` throughout the mentioned contracts.

Status:

This issue has chosen to be ignored in order to save gas during function execution.

LOW-2 (MONITORED) CloprBottles contract address should be immutable

Impact: Low

Description:

The variable `BOTTLES_CONTRACT`, representing the CloprBottles' smart contract address in both CloprStories.sol and StoryPotion.sol, is currently declared as a `constant`. However, being a constant necessitates manual changes before each deployment, posing a risk of deployment issues.

Recommendation:

To enhance deployment flexibility and avoid modifying code intervention, it is advised to replace the constant BOTTLES_CONTRACT with an immutable variable initialized in the constructor.
The new declaration should be:

```
ICloprBottles private immutable BOTTLES_CONTRACT;
```

Status:

To conserve gas during function execution, this matter has been deliberately disregarded, with careful attention planned for contract deployment.

LOW-3 (RESOLVED) Missing input checks

Impact: Low

Description:

The smart contract currently lacks comprehensive verification for 0 inputs, especially in scenarios involving the contract owner. Ensuring robust input validation is essential not only for preventing potential vulnerabilities but also for mitigating errors that may arise from caller actions.

On StoryPotion.sol:

- `baseUri_`

```
constructor(string memory baseUri_) ERC721("StoryPotion", "SP") {
```

- `newPrice`

```
function adminChangeFillPrice(
    uint64 newPrice
) external onlyRole(MODIFY_FILL_PRICE_ROLE) {
```

- `newBaseUri`

```
function changeDefaultBaseUri(string memory newBaseUri) external onlyOwner {
```

On CloprBottles.sol:

- `newPotionBaseUri`

```
function changePotionBaseUri(
    uint256 potionId,
    string memory newPotionBaseUri
) external onlyOwner {
```

- `potionFillContract`, `potionEmptyContract` and `potionBaseUri`

```
function addNewPotion(
    uint256 potionId,
    address potionFillContract,
    address potionEmptyContract,
    string memory potionBaseUri
) external onlyOwner {
```

- `price`, `startTimestamp`, `endTimestamp`, `maxMintPerWallet` and `phaseSupply`

```
function createNewMintPhase(
    uint256 phaseIndex,
    uint128 price,
    uint48 startTimestamp,
    uint48 endTimestamp,
    uint8 maxMintPerWallet,
    uint16 phaseSupply
) external onlyRole(MINT_PHASE_ROLE) {
```

On CloprStories.sol:

- `startDefaultBaseUri`

```
constructor(
    string memory startDefaultBaseUri
) ERC721("CloprStories", "CSTR") {
```

Recommendation:

We recommend verifying those values are not equal to the default value, depending on variable type.

Status:

The fix is implemented for values that can't be equal to zero.

LOW-4 (RESOLVED) Enhancing delegatecash usage in `burnAndGrowStory` function

Impact: Low

Description:

The `burnAndGrowStory` function currently checks if NFTs are delegated to the delegatecash smart contract. However, this check fails if `msg.sender` is the owner of one NFT and delegates the other.

```
if (vault != address(0)) {
    bool isDelegateValid = DC.checkDelegateForERC721(
        msg.sender,
        vault,
        address(burnedStory.unftContract),
        burnedStory.unftTokenId,
        ""
    ) &&
        DC.checkDelegateForERC721(
            msg.sender,
            vault,
            address(extendedStory.unftContract),
            extendedStory.unftTokenId,
            ""
        );
    if (!isDelegateValid) revert InvalidDelegateVaultPairing();
    requester = vault;
}
```

Recommendation:

To address this, we recommended modifying the function to accept two vault addresses (`burnedStoryVault` and `extendedStoryVault`) and use separate variables (`burnedStoryRequester` and `extendedStoryRequester`) to track the requesters for each NFT.

```
address burnedStoryRequester = msg.sender;
address extendedStoryRequester = msg.sender;

if (burnedStoryVault != address(0)) {
    if (!DC.checkDelegateForERC721(
        msg.sender,
        burnedStoryVault,
        address(burnedStory.unftContract),
        burnedStory.unftTokenId,
        ""
    )) revert InvalidDelegateVaultPairing();
    burnedStoryRequester = burnedStoryVault;
}

if (extendedStoryVault != address(0)) {
    if (!DC.checkDelegateForERC721(
        msg.sender,
        extendedStoryVault,
        address(extendedStory.unftContract),
        extendedStory.unftTokenId,
        ""
    )) revert InvalidDelegateVaultPairing();
    extendedStoryRequester = extendedStoryVault;
}
```

Status:

The fix is implemented, following recommendations.

LOW-5 (RESOLVED) Useless type conversion

Impact: Low

Description:

In `addNewPotion` function, there are few useless address-to-address conversions. This makes loss some gas for nothing.

```
function addNewPotion(
    uint256 potionId,
    address potionFillContract,
    address potionEmptyContract,
    string memory potionBaseUri
) external onlyOwner {
    if (potionUris[potionId].exists) revert PotionAlreadyExist();

    potionUris[potionId] = PotionBaseUri({
        baseUri: potionBaseUri,
        exists: true,
        isFrozen: false
    });

    fillAccess[potionId] = address(potionFillContract); //<-- here
    emptyAccess[potionId] = address(potionEmptyContract); //<-- here

    emit NewPotion(
        address(potionFillContract), //<-- here
        address(potionEmptyContract), //<-- here
        potionId,
        potionBaseUri
    );
}
```

Recommendation:

We recommend not using the `address` keyword in this case.

Status:

The fix is implemented, following recommendations.

LOW-6 (RESOLVED) Preventing self-burn NFT in `burnAndGrowStory` function

Impact: Low

Description:

If a user specifies the same story ID for both `burnedTokenId` and `extendedTokenId` in the `burnAndGrowStory` function, the story will be burned automatically without any protection.

Recommendation:

We recommend to add a condition that checks if the `burnedTokenId` and `extendedTokenId` are different to prevent unintentional story burning.

```
if (burnedTokenId == extendedTokenId) revert SameStory();
```

Status:

The fix is implemented, following recommendations.

INF-1 (RESOLVED) Implicit visibility for MODIFY_FILL_PRICE_ROLE

Impact: Informational

Description:

The variable `MODIFY_FILL_PRICE_ROLE` in the StoryPotion contract is implicitly of type internal as its visibility is not explicitly specified.

```
bytes32 constant MODIFY_FILL_PRICE_ROLE =
    keccak256("MODIFY_FILL_PRICE_ROLE");
```

Recommendation:

Specify the visibility of the variable to private.

```
bytes32 private constant MODIFY_FILL_PRICE_ROLE =
    keccak256("MODIFY_FILL_PRICE_ROLE");
```

Status:

The fix is implemented, following recommendations.

INF-2 (RESOLVED) Function order in contracts structure

Impact: Informational

Description:

All functions within the contract lack adherence to the recommended function order, as outlined in the Solidity style guide (https://docs.soliditylang.org/en/latest/style-guide.html). This order assists readers in identifying callable functions and locating the constructor, receive function (if present), and fallback function (if present) more efficiently.

Recommendation:

Reorganize the functions in the contract structure following the recommended order:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Status:

The fix is implemented, following recommendations.

INF-3 (MONITORED) Architecture of StoryPotion contract

Impact: Informational

Description:

The primary functionality of the StoryPotion.sol smart contract is to act as a potion tank, filling up potions and creating one associated NFT. However, the current design inherits from ERC721 libraries, combining NFT creation and tank behavior within a single contract. To enhance code readability and adhere to best practices, it is recommended to separate these concerns into distinct smart contracts.

Recommendation:

Create a new smart contract specifically dedicated to handling the NFT functionality. This separation will result in two distinct contracts—one for managing the potion tank behavior and another for NFT creation—improving code organization and readability.

Status:

To maintain the same address for both the StoryPotion fill logic and the unique NFT, the decision has been made to overlook this issue.

INF-4 (RESOLVED) `tokenURI` function comment issue

Impact: Informational

Description:

The comment for the `tokenURI` function incorrectly refers to a "Clopr Story" instead of the correct term "Story Potion."

```
    /// @notice Get a Clopr Story's metadata URI
```

Recommendation:

Update the comment for the `tokenURI` function to accurately reflect its purpose.

```
    /// @notice Get a Story Potion's metadata URI
```

Status:

The fix is implemented, following recommendations.

INF-5 (RESOLVED) Type consistency improvement in `getFillPrice` function

Impact: Informational

Description:

In the StoryPotion.sol smart contract, the `getFillPrice` function currently has an unexplicit conversion, returning a `uint256` while the underlying variable `fillPrice` is a `uint64`. To enhance consistency and avoid unnecessary conversions, the function should explicitly return a `uint64`.

```
function getFillPrice() external view returns (uint256 fillPrice_) {
    fillPrice_ = fillPrice;
}
```

Recommendation:

Update the return type of the `getFillPrice` function to `uint64`.

```
function getFillPrice() external view returns (uint64 fillPrice_) {
    fillPrice_ = fillPrice;
}
```

Status:

The fix is implemented, following recommendations.

INF-6 (RESOLVED) Variable optimization and readability

Impact: Informational

Description:

In the `BottleInformation` struct of the smart contract logic, the `potionFill` variable is currently represented as a `uint8` with values of 100 for fill and 0 for empty. While it is designed to accommodate future usage where different fill levels may be employed, if the project intends to consistently use only 0 or 100, we recommend following our instructions for optimization.

To optimize gas efficiency and enhance readability, it is suggested to replace the `potionFill` variable with a boolean `isFilled`. This change not only improves gas efficiency and readability but also helps prevent potential undiscovered issues in the future related to uint calculations, especially if different StoryPotion contract implementations are introduced. This enhancement ensures a clearer and more robust representation of the potion's fill status.

```
struct BottleInformation {
    uint16 potionId;
    uint8 potionFill;
    bool delegatedFillLevel;
    uint48 stakingTime;
    uint24 numberDrinks;
    uint256 lastEmptyBlock;
}
```

Recommendation:

Update the `BottleInformation` struct by replacing `potionFill` with a boolean variable `isFilled`.

Status:

The fix is implemented, following recommendations.

III. Second review notes

MED-2 (RESOLVED) ERC Interface support

Description:

The supportsInterface function lacks support for ERC4906 and ERC5192. It is crucial to include these interfaces to ensure comprehensive compatibility.

Recommendation:

Include ERC4906 and ERC5192 in the supportsInterface function.

```
    function supportsInterface(
        bytes4 interfaceId
    )
        public
        pure
        override(ERC721A, AccessControl, IERC721A, ERC2981)
        returns (bool supports_)
    {
        supports_ =
            interfaceId == type(IERC721).interfaceId || // ERC165 interface ID for ERC721
            interfaceId == type(IERC721Metadata).interfaceId || // ERC165 interface ID for
ERC721Metadata
            interfaceId == type(IERC2981).interfaceId || // ERC165 interface ID for ERC2981
            interfaceId == type(IERC165).interfaceId || // ERC165 interface id for ERC165
            interfaceId == type(IAccessControl).interfaceId || // ERC165 interface id for AccessControl
            interfaceId == type(IERC4906).interfaceId || // ERC165 interface id for ERC4906
            interfaceId == type(IERC5192).interfaceId; // ERC165 interface id for ERC5192
    }
```

Status:

The status is now resolved as it has been fixed.

MED-3 (RESOLVED) ERC4906 security considerations

Description:

In light of ERC4906, a new function addressing security considerations related to off-chain modifications of metadata should be added. This ensures that the contract adheres to ERC4906 standards.

Recommendation:

We recommend to add a function in order to allow the owner to emit the `MetadataUpdate` event.

Status:

The status is now resolved as it has been fixed.

LOW-7 (RESOLVED) `_beforeTokenTransfers` override

Description:

The decision to override `_beforeTokenTransfers` is considered beneficial for code readability. However, it is essential to note a slight logic change in which any user must now wait for 20 blocks to burn their NFT after emptying it.

Recommendation:

We recommend taking no action if it aligns with the end user's logic and is deemed acceptable.

Status:

The technical team has reviewed and approved this modification, and the status is confirmed as satisfactory.