Smart Contract Security Report

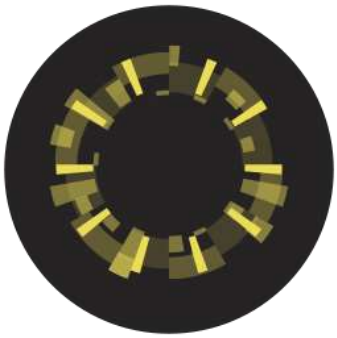
Yellow Blocks

2021-06-25

Content

Introduction

The process of tokenizing a film's frames and further managing it on a Dapp would open up new opportunities, at the same time it faces subsequent challenges to its adoption. The new tokenization method allows The Producer to open up innovative methods and promote widespread adoption of investments in the entertainment industry.



The Producer deploys and initiates the distribution of a new digital crypto token called CBK, which can be used by holders to enhance The Docuseries experience by interacting with The Yellow Dapp.

As requested by **Yellow Blocks** and as part of the vulnerability review and management process, **Red4Sec** has been asked to perform a security code audit to **evaluate the security of** its CBK token.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

Scope

The scope of this evaluation includes the following Yellow Blocks contracts:

- CBK- Token Smart Contract Audit:
  - https://github.com/Crossing-the-Yellow-Blocks/CBK-Token
- CBK.sol
  de2fb6f1d4fac8a9a2ce42cd44fee88f9133422e78d6203612efa4045e684281

Conclusions

The general conclusion resulting from the conducted audit, is that the CBK Token's smart contract **is secure and does not present any known vulnerabilities**. Nevertheless, Red4Sec has found a few potential improvements, these do not pose any risk by themselves, and we have classified such issues as informative only, but they will help **Crossing the Yellow Blocks** to continue to improve the security and quality of its developments.

- The **absence of the Unit Test** has been detected, during the security review, this is a highly recommended practice that has become mandatory in projects destined to manage large amounts of capital.
- The CBK contract includes its own implementation of the ERC-20 standard, which although it is correctly executed, it is a better practice to use highly tested libraries such as Open Zeppelin and to avoid developing own implementations.
- A **few low impact issues** were detected and classified only as informative, but they will continue to help *Crossing The Yellow Blocks* improve the security and quality of its developments.
- The *Crossing The Yellow Block* Project has been **developed using outdated versions**, and in some cases, versions marked as obsolete, both from the compiler and from third-party libraries. This is an absolutely discouraged practice and should be solved prior to the deployment.
- The overall impression about code quality and organization is not optimal. The developed code does not comply with code standards and lacks some good development practices. Some methods contain unnecessary modifiers, such as virtual, and the inheritances are not optimally applied.

Recommendations

Outdated Compiler version

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version.

We have detected that the audited contract uses the following version of Solidity pragma ^0.6.0:

```
pragma solidity ^0.6.0;
```

The 0.6.X version of Solc is affected by different known bugs that have already been fixed in later versions. It is always a good policy to use the most up to date version of the pragma.

Additionally, it is currently recommended to use at least the latest version of the branch 0.7.x or 0.8.x of the compiler.

Unsecure Ownership Transfer

The modification process of an owner is a delicate process, since the governance of our contract and therefore of the project may be at risk, for this reason it is recommended to adjust the owner's modification logic, to a logic that allows to verify that the new owner is in fact valid and does exist, for example, a missing zero address validation was detected.

In case this is the intended logic, an exclusive method should be implemented in case of wanting to renounce the ownership, which should include a call to said method when calling ***finishMinting***.

Following, we can see a standard logic of the owner's modification where a new owner is proposed first, the owner accepts the proposal and, in this way, we make sure that there are no errors when writing the address of the new owner.

```
function proposeOwner(address _proposedOwner) public onlyOwner
{
        require(msg.sender != _proposedOwner, ERROR_CALLER_ALREADY_OWNER);
        proposedOwner = _proposedOwner;
}

function claimOwnership() public
{
        require(msg.sender == proposedOwner, ERROR_NOT_PROPOSED_OWNER);
        emit OwnershipTransferred(_owner, proposedOwner);
        _owner = proposedOwner;
        proposedOwner = address(0);
}
```

Source reference

- CBK.sol

Use of Require statement without reason message

Throughout the audit, it was verified that the reason message is not specified in some *require* methods, in order to give the user more information, which consequently makes it more user friendly.

An example of this issue can be found in: CBK.sol:189

```
function mint(address account, uint256 amount) onlyOwner public {
    require(account != address(0), "ERC20: mint to the zero address");
    require(!mintingFinished);
    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}
```

This functionality is available since version 0.4.22 and the contract's pragma indicates ^0.6.0, this will result in compatibility with this feature.

Outdated Third-Party Libraries

The smart contracts analyzed inherit functionalities from open-zeppelin contracts that have been labelled obsolete and/or outdated; this does not imply a vulnerability by itself, because their logic does not present them, but it does imply that an update is not carried out by third party packages or libraries.

Currently the latest version of OpenZeppelin contracts is 4.1.0, therefore it would be convenient to include it as a reference instead of including the sources, in this way we will keep the development environment updated; for example, the SafeMath library of the OpenZeppelin is outdated to its latest compatible version with the established contract's pragma (^0.6.0).

Additionally, these OpenZeppelin contracts are under the MIT license, which requires its license/copyright to be included within the code.

By using the original sources, in case the project resolves any vulnerability or bug in the code, you would obtain this update automatically. Consequently, avoiding inheriting known vulnerabilities.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.4/contracts/math/SafeMath.sol

Recommendations

- Include third-party codes by package manager.
- Include in the Crossing the Yellow Blocks project any references/copyright to OpenZeppelin code since it is under MIT license.

Lack of Event

Generally, events are used to inform the calling application about the current state of the contract. Events notify the applications about the change made to the contracts and applications which can be used to execute the dependent logic.

This can be very useful when making DApps or in the off-chain processing of the events in our contract, as it allows filtering by specific addresses, making it much easier for developers to query the results of invocations.

It would be convenient to review the **CBK** contract to ensure that all important actions emit an event to inform possible DApps and users. For example, it is recommended to issue an event for the ***finishMinting*** and ***transferOwnership*** methods.

Source reference

- https://github.com/Crossing-the-Yellow-Blocks/CBK-Token/blob/main/CBK.sol#L195-L197
- https://github.com/Crossing-the-Yellow-Blocks/CBK-Token/blob/main/CBK.sol#L212-L214

Recommendation

- Issue an event for important parameters changes.

GAS Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Executions Costs

The use of constants is recommended as long as the variables are never to be modified. In this case the variables "_name", "_symbol" and "_decimals" of the **CBK** contract should be declared as constants since they would not be necessary

to access the storage to read the content of these variables and therefore the execution cost is much lower.

```
constructor () public {
    _name = "CBK";
    _symbol = "CBK";
    _decimals = 18;
    _owner = msg.sender;
    _totalSupply = 21000000 ether;
    _balances[msg.sender] = _totalSupply;
}
```

Wrong Visibility

In order to simplify the contract for the users, it is recommended to turn "name", "symbol" and "decimals" to public variables.

When initializing variables as public the following methods will no longer be necessary, and can be removed:

```
function name() public view returns (string memory) {
    return _name;
}

function symbol() public view returns (string memory) {
    return _symbol;
}

function decimals() public view returns (uint8) {
    return _decimals;
}
```

Source References

- https://github.com/Crossing-the-Yellow-Blocks/CBK-Token/blob/main/CBK.sol#L115-L125

Use of Context Class

The CBK contract inherits a functionality of the Context class of OpenZeppelin, which is designed to be used with Ethereum Gas Station Network (GSN)[1], but it also contains references to *msg.sender*.

```
modifier onlyOwner() {
    require(msg.sender == _owner, "Only the owner is allowed to access this function.");
    _;
}

constructor () public {
    _name = "CBK";
    _symbol = "CBK";
    _decimals = 18;
    _owner = msg.sender;
    _totalSupply = 21000000 ether;
    _balances[msg.sender] = _totalSupply;
}
```

It is advisable to review that these functionalities are being used and to replace *msg.sender* with *_msgSender()* in every occurrence. If this is not the case, remove the *Context* functionality.

Source References

- https://github.com/Crossing-the-Yellow-Blocks/CBK-Token/blob/main/CBK.sol#L102-L112
- https://github.com/Crossing-the-Yellow-Blocks/CBK-Token/blob/main/CBK.sol#L177

[1] https://docs.opengsn.org/

Invest in Security, invest in your future