BlockSec

Security Audit Report for Wrapped Tokens Contracts

Date: November 01, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.11.01 23:15:35 +08'00'

Contents

Report Manifest

Item	Description
Client	Crypto.com
Target	Wrapped Tokens Contracts

Version History

Version	Date	Description
1.0	November 01, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repositiory of Wrapped Tokens Contracts of Crypto.com. The Wrapped Tokens Contracts serve as ERC20 token contracts representing staked and generic assets.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The MD5 values of the files during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Version 1: File	md5
wrapped-tokens-os-main.zip	9f295e3251b819cbf8ea299bff769292
contracts/wrapped-tokens/MintUtil.sol	40c97260847ed5c094da04aa921e91b1
contracts/wrapped-tokens/MintForwarder.sol	c729f40e147f19273a29c75a59e5f955
contracts/wrapped-tokens/FiatTokenProxy.sol	f7ee7d926a4d31b39e154ac7ed5000cf
contracts/wrapped-tokens/RateLimit.sol	8ef7cad260fa577ea080d157e5ca9bdd
contracts/wrapped-tokens/staking/ExchangeRateUtil.sol	ec28a374455abc949ec05c5a2d4334d1
contracts/wrapped-tokens/staking/ExchangeRateUpdater.sol	c08e90496bafa47d86c3e4cd8759bca7
contracts/wrapped-tokens/staking/LiquidETHV1.sol	96f5ff1ef834bd57d14251f99ea61341

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

- ∗ Gas optimization
- ∗ Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we do not find potential issues. Besides, we also have **two** recommendations and **three** notes.

- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	-	Remove duplicate checks	Recommendation	Acknowledged
2	-	Prevent accidental native token transfers	Recommendation	Acknowledged
3	-	Potential centralization risk	Note	-
4	-	Ensure the proper initialization of `ExchangeRateUpdater` and `MintForwarder`	Note	-
5	-	Ensure the proper configuration of `maxAllowance`	Note	-

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Remove duplicate checks

Status Acknowledged

Introduced by `Version 1`

Description The check on Line 72-75 that verifies `newOwner` address is non-zero is redundant, as this validation is already performed in the `transferOwnership` function.

```
67   function initialize(address newOwner, address newTokenContract)
68       external
69       onlyOwner
70   {
71       require(!initialized, "MintForwarder: contract is already initialized");
72       require(
73           newOwner != address(0),
74           "MintForwarder: owner is the zero address"
75       );
76       require(
77           newTokenContract != address(0),
78           "MintForwarder: tokenContract is the zero address"
79       );
80       transferOwnership(newOwner);
81       tokenContract = newTokenContract;
82       initialized = true;
83   }
```

Listing 2.1: MintForwarder.sol

The same issue also exists in the `initialize` function of the `ExchangeRateUpdater` contract.

```
function initialize(address newOwner, address newTokenContract)
    external
    onlyOwner
{
    require(
        !initialized,
        "ExchangeRateUpdater: contract is already initialized"
    );
    require(
        newOwner != address(0),
        "ExchangeRateUpdater: owner is the zero address"
    );
    require(
        newTokenContract != address(0),
        "ExchangeRateUpdater: tokenContract is the zero address"
    );
    transferOwnership(newOwner);
    tokenContract = newTokenContract;
    initialized = true;
}
```

Listing 2.2: ExchangeRateUpdater.sol

Impact N/A

Suggestion Remove duplicate checks accordingly.

2.1.2 Prevent accidental native token transfers

Status Acknowledged

Introduced by `Version 1`

Description The `FiatTokenProxy` contract cannot process native token transfers for current design. However, the `fallback` function (in the inherited `Proxy` contract) only reverts when `msg.sender` is `admin`. This poses a risk where users accidentally send native tokens to the contract and have their funds locked. The locked assets can only be withdrawn by upgrading the contract, which brings extra costs.

```
function _fallback() internal {
    _willFallback();
    _delegate(_implementation());
}
```

Listing 2.3: Proxy.sol

```
function _willFallback() internal override {
    require(
        msg.sender != _admin(),
        "Cannot call fallback function from the proxy admin"
    );
    super._willFallback();
}
```

Listing 2.4: AdminUpgradeabilityProxy.sol

Impact The accidentally transferred assets are locked until an upgrade is performed.

Suggestion Revise the code logic accordingly.

2.2 Note

2.2.1 Potential centralization risk

Description The owner of the `FiatTokenProxy` contract currently has the authority to assign privileged roles, including masterMinter, pauser, blacklister, etc. Additionally, the owner-assigned rescuer can withdraw ERC20 tokens from the contract via the `rescueERC20` function. This concentration of privileges to the owner account raises centralization risks. If the private key of the owner is compromised, it could be abused to conduct misbehavior.

Feedback from the Project To mitigate risk of centralization, private key will be held by a MPC wallet of crypto.com and we will build a better on-chain contract monitoring.

2.2.2 Ensure the proper initialization of `ExchangeRateUpdater` and `MintForwarder`

Description In the `ExchangeRateUpdater` and `MintForwarder` contracts, the `initialize` function can only be called by the contract owner. If using the proxy pattern, the proxy contract must store the current owner in the same storage slot. Otherwise, this `initialize` function could never be executed to initialize the contracts.

2.2.3 Ensure the proper configuration of `maxAllowance`

Description The `configureCaller` function in the `RateLimit` contract allows the owner to add or update a caller. If the caller's `maxAllowance` is set excessively high, there is a potential overflow risk when the `_getReplenishAmount` function calculates the amount to replenish the caller's allowance (lines 201-202). This can cause the `_getReplenishAmount` function to revert, preventing the allowance from being replenished as expected. To prevent this, the contract owner should thoughtfully configure the `maxAllowance`.

```
function configureCaller(
    address caller,
    uint256 amount,
    uint256 interval
) external onlyOwner {
    require(caller != address(0), "RateLimit: caller is the zero address");
    require(amount > 0, "RateLimit: amount is zero");
    require(interval > 0, "RateLimit: interval is zero");
    callers[caller] = true;
    maxAllowances[caller] = allowances[caller] = amount;
    allowancesLastSet[caller] = block.timestamp;
    intervals[caller] = interval;
    emit CallerConfigured(caller, amount, interval);
}
```

Listing 2.5: RateLimit.sol

```
function _getReplenishAmount(address caller)
    internal
    view
    returns (uint256)
{
    uint256 secondsSinceAllowanceSet = block.timestamp -
        allowancesLastSet[caller];

    uint256 amountToReplenish = (secondsSinceAllowanceSet *
        maxAllowances[caller]) / intervals[caller];
    uint256 allowanceAfterReplenish = allowances[caller] +
        amountToReplenish;

    if (allowanceAfterReplenish > maxAllowances[caller]) {
        amountToReplenish = maxAllowances[caller] - allowances[caller];
    }
    return amountToReplenish;
}
```

Listing 2.6: RateLimit.sol