verichains

SECURITY AUDIT OF

CAMLY SMART CONTRACTS

Public Report

January 13, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on January 13, 2023. We would like to thank the Camly for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Camly Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the application, along with some recommendations. Camly team has resolved and updated all issues following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Camly Smart Contracts

CamLy Ecosystem consists of seven platforms, is a great work of God the Father, assigned to Lightleaders and Lightworkers to act and carry out the mission of saving humanity, sentient beings, and helping Mother Earth ascend.

Camly Legal: A place to connect and share international law. Protect the truth, justice, human's right and love.

Camly Profile: A place to connect successful people in the world. The platform helps users create personal and company profiles, helping their business grow strong and professional.

Camly Trading: is a place to buy, sell, exchange and auction NFT technology products and other intellectual products of Gamers and (digital) real estate investors.

CamlyLife: is a social networking - entertainment - investment - business - education experience based on a 3D real - world platform that simulates the world.

Camly Investment Platform: is a place to share and connect global real estate business and investment opportunities.

Camly Charity Platform: The platform serves the needs of sharing and connecting the charity, philanthropist and volunteering.

Camly Academy: This platform helps to connect and share the knowledge of mankind. Vibrant visual interaction, making learners fascinated without taking their eyes off, from which they have more information and knowledge to succeed.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of Camly Smart Contracts.

It was conducted on commit `637a532c805e9b7e0164ef4765924867be60dde3` from git repository link: *https://github.com/CamlyEcosytem/CamlySmartContract*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`5e6f27a5e1366e625b434f4237db5ba248874d26d4fdb9f1cb584948fcb2c33d`	`interface/ICamlyMarketProxy.sol`
`44da06e907d75da89d2589187643cc27be985b767f3d1eb0e6c051c24c4dc047`	`CamlyMarket.sol`
`14334da3a09ddc06f23891e2158468adc6ae4ac72ddd2633d10eda605d15f6d3`	`Camly_token.sol`

d808e43cd726158628cf2a41591afcf2fe4a4ebf5334119c8b0dce91f2c53e7d	Camly.sol
8bac56756e56303689543263f0b9b2eda2d95baced4b9bd8d661ebbd839ec864	CamlyProxy.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.

SEVERITY LEVEL	DESCRIPTION
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Camly Smart Contracts was written in `Solidity` language, with the required version to be `0.8.16`. The source code was written based on OpenZeppelin's library and the ERC721A library developed by Chiru Labs.

2.1.1. Camly token contract

The Camly token contract is an ERC20-based contract. Besides the default ERC20 functions, the contract implements the external `burn` function that only allows the contract owner to call.

The Table below lists some properties of the audited `Camly_token` contract (as of the report writing time).

PROPERTY	VALUE
Name	CamlyDEV
Symbol	CLYD
Decimals	18
Total Supply	1,500,000,000 ($x10^{18}$) Note: the number of decimals is 18, so the total representation token will be 1,500,000,000 or 1.5 billion.

Table 2. The Camly token contract properties

2.1.2. NFT contract

Camly NFT contract is an ERC721A contract that extends `ERC721A`, `ERC2981`, `ReentrancyGuard`, and `AccessControl` contracts. The direct transfer function is disabled by default, the only whitelisted sender is the Camly Proxy contract. However, the direct transfer function can be enabled by the user with `ADMIN_ROLE`.

2.1.3. Marketplace contract

The marketplace contract is where users can exchange the NFT with each other. The buyer can offer a price for an NFT he wants and the seller can list an NFT item with a desired price.

2.1.4. Camly Proxy contract

The Camly Proxy contract is a mediator contract that implements Camly NFTs transfer function and is whitelisted by the Camly NFT contract. So that, the Camly NFTs donation feature can be implemented.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Camly Smart Contracts. Camly team fixed the code, according to Verichains's draft report.

#	Issue	Severity	Status
1	Missing authorization for `burn` function	CRITICAL	Fixed
2	Missing checks for buyer and seller in the `ExcuteOrder` function	CRITICAL	Fixed
3	User cannot cancel order after signing	HIGH	Acknowledged
4	The `DonateNFTs` function can be abused to bypass the direct transfer blocking of Camly NFTs	MEDIUM	Acknowledged
5	Missing order type verification in the `verifySign` function	LOW	Acknowledged

Audit team also suggested some possible enhancements.

#	Issue	Status
1	Typos	Acknowledged

2.2.1. Missing authorization for `burn` function CRITICAL

Affected files:

- Camly.sol

In the CamlyDEV NFT contract, the `burn` function can be called by anyone without restrictions. So, attackers can call this function to burn any token they don't own.

```
function burn(
        uint256 tokenId)
            external nonReentrant {
                _burn(tokenId);
```

```
                _resetTokenRoyalty(tokenId);
}
```

RECOMMENDATION

Check token ownership of the caller before burning.

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged and fixed by the Camly team.

2.2.2. Missing checks for buyer and seller in the ExcuteOrder function CRITICAL

Affected files:

- CamlyMarket.sol

The `ExcuteOrder` function calls the `transferAsset` function to transfer NFT to the buyer and payment tokens to the seller. However, the `transferAsset` function doesn't check the buyer and seller correctly. There are two cases that can be exploited in this function as below:

- In case of BUYORDER, the value of `orderdetails.buyer` can be manipulated so that the attacker can force anyone to buy his NFTs (assuming that the attacker is the buyer). We need to ensure that the `buyer` is the `msg.sender` in this case.
- In case of SELLORDER, the value of `orderdetails.seller` can be manipulated so that the attacker can buy NFTs of any seller without the seller's permission (the price can be specified by the attacker). We need to ensure that the `seller` is the `msg.sender` in this case.

```
function ExcuteOrder(
        OrderDetails calldata orderdetails,
        Sign calldata sign
) external returns(bool){
    if(
        orderdetails.orderType ==
            OrderType.BUYORDER)
                {
                    verifySign(
                        orderdetails.seller,
                                orderdetails.nftAddress,
                                    orderdetails.erc20Address,
                                orderdetails.tokenId,
                            orderdetails.amount,
                        sign
                        );
        Fee memory fee = getFees(
                            orderdetails
                        );
        transferAsset( // MISSING `orderdetails.buyer` CHECK
```

```
            orderdetails,
                fee
        );
        // ...
    }

    if(
        orderdetails.orderType ==
            OrderType.SELLORDER)
                {
                    verifySign(
                            orderdetails.buyer,
                                    orderdetails.nftAddress,
                                        orderdetails.erc20Address,
                                    orderdetails.tokenId,
                                orderdetails.amount,
                            sign
                    );
        Fee memory fee = getFees(
                            orderdetails
                        );
        transferAsset( // MISSING `orderdetails.seller` CHECK
                orderdetails,
                    fee
        );
        // ...
    }

    return true;
}
```

RECOMMENDATION

The `ExcuteOrder` function can be fixed as below:

```
function ExcuteOrder(
        OrderDetails calldata orderdetails,
        Sign calldata sign
) external returns(bool){
    if(
        orderdetails.orderType ==
            OrderType.BUYORDER)
                {
                    verifySign(
                        orderdetails.seller,
                                orderdetails.nftAddress,
                                    orderdetails.erc20Address,
                                orderdetails.tokenId,
                            orderdetails.amount,
                        sign
                        );
```

```
                    require(orderdetails.buyer == msg.sender, "Buyer must be msg.sender");
// FIXED
        Fee memory fee = getFees(
                            orderdetails
                        );
        transferAsset(
            orderdetails,
                fee
        );
        // ...
    }

    if(
        orderdetails.orderType ==
            OrderType.SELLORDER)
                {
                    verifySign(
                            orderdetails.buyer,
                                    orderdetails.nftAddress,
                                        orderdetails.erc20Address,
                                    orderdetails.tokenId,
                                orderdetails.amount,
                            sign
                    );
                    require(orderdetails.seller == msg.sender, "Seller must be
msg.sender"); // FIXED
        Fee memory fee = getFees(
                            orderdetails
                        );
        transferAsset(
                orderdetails,
                    fee
        );
        // ...
    }

    return true;
}
```

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged and fixed by the Camly team.

2.2.3. User cannot cancel order after signing HIGH

Affected files:

- CamlyMarket.sol

In the `CamlyMarket` contract, there is no way to cancel an order after signing the signature.

RECOMMENDATION

A `cancelOrder` function should be added to this contract to allow signers to blacklist their unused signatures.

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged by the Camly team.

2.2.4. The `DonateNFTs` function can be abused to bypass the direct transfer blocking of Camly NFTs MEDIUM

Affected files:

- CamlyMarket.sol
- Camly.sol

The CamlyDEV NFT contract doesn't allow users to transfer their NFTs between each other by default since `isDirectTransferBlocked` is set to `true`. However, this restriction can be bypassed by calling the `DonateNFTs` function of the `CamlyExchangeMarket` contract.

```
contract CamlyExchangeMarket is AccessControl{
    // ...

    function DonateNFTs(
        address tokenAddress,
        address donationAddress,
        uint256 tokenId
    ) external returns(bool) {
        require
        (tokenAddress != address(0)
        && donationAddress != address(0)
            &&  tokenId != 0
        ,"Donation: Invalid values"
        );
        CamlyMarketProxy.erc721safeTransferFrom(
            IERC721(tokenAddress),
            _msgSender(),
            donationAddress,
            tokenId
        );
        return true;
    }
    // ...
}
```

RECOMMENDATION

The `DonateNFTs` function should only allow a list of valid `donationAddress`.

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged by the Camly team. It is a business decision to have it this way. Also, it does not violate any security and compliance rules.

2.2.5. Missing order type verification in the `verifySign` function LOW

Affected files:

- CamlyMarket.sol

In the `verifySign` function, the order type (BUY or SELL) is not checked in the signature. So, the signature for BUY order can be used for SELL order and vice versa.

```
function verifySign(
    address signer,
    address nftAddress,
    address paymentAssetAddress,
    uint256 tokenId,
    uint256 paymentAmount,
    Sign memory sign
) internal  {
    bytes32 hash = keccak256(
        abi.encodePacked(this, nftAddress, signer, paymentAssetAddress, paymentAmount,
tokenId,sign.nonce)
    );

    require(
        !isValidSign[hash],
        "Duplicate Sign"
    );

    isValidSign[hash] = true;

    require(
        signer ==
            ecrecover(
                keccak256(
                    abi.encodePacked(
                        "\x19Ethereum Signed Message:\n32",
                        hash
                    )
                ),
                sign.v,
                sign.r,
                sign.s
            ),
        "Signer sign verification failed"
```


```
    );
}
```

RECOMMENDATION

Order type should also be included in the signature along with other parameters.

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged by the Camly team.

2.3. Additional notes and recommendations

2.3.1. Typos INFORMATIVE

- `_changetTreasuryWallet` in `CamlyMarket.sol`
- `ExcuteOrder` in `CamlyMarket.sol`

UPDATES

- *Jan 13, 2023*: This issue has been acknowledged by the Camly team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jan 13, 2023*	Public Report	Verichains Lab

Table 3. Report versions history