verichains

SECURITY AUDIT OF

U2WIN SMART CONTRACT

U2Win
U2U HORSE RACING

Public Report

July 10, 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.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on July 10, 2023. We would like to thank the U2Win 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 U2Win Smart Contract. 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 2 vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About U2Win Smart Contract

U2Win is a secure, trustworthy, and decentralized horse racing game. It offers a seamless user experience and ensures high reliability through rigorous testing of all security protocols.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the U2Win Smart Contract.

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

SHA256 Sum	File
f9ccdb87d6ba541b1c4fe9bc47b03bd43c71fe541fac4a27b49b07158ad289cd	U2WinSale.sol
eeb8af8bc63791d40b71efc1ab99bc4280f06360731d7f346b3109732a5df055	U2Win.sol
ff9c8fc5c64294157f06aee117c35e111be1326b6238ec7a6c829928f69017d1	U2WinUtils.sol
a2d0012d0389805e9d240921dcfd8719f1346308392a85cb289b18f076fd5494	U2WinVault.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.
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 U2Win Smart Contract was written in `Solidity` language, with the required version to be `^0.8.6`. The source code was written based on OpenZeppelin's library.

2.1.1. U2Win.sol

The `U2Win` contract extends `ERC721Enumerable`, `Ownable` and `AccessControl` contracts. `AccessControl` allows the contract to implement role-based access control mechanisms. There are 2 roles: `DEFAULT_ADMIN_ROLE` and `MINTER_ROLE`. The contract deployer will be granted `DEFAULT_ADMIN_ROLE` and can assign any roles to anyone. Users with the `MINTER_ROLE` can mint any quantity of NFTs across any category. With `Ownable`, by default, the contract owner is contract deployer, but he can transfer ownership to another address at any time. The contract owner only has the ability to set the base URI.

2.1.2. U2WinSale.sol

The `U2WinSale` contract extends `Ownable` and `Pausable` contracts. With `Ownable`, by default, the contract owner is the contract deployer, but he can transfer ownership to another address at any time. The contract owner has the authority to enable trading for any category and can modify the sale metadata information at any time, including cap, maxCap, startTime, endTime, quoteToken, and price.

Additionally, the contract owner can cancel the sale using the `cancelSellNFT()` function. Users can only purchase new NFT items in the specified category when the contract is not paused and within the time range from startTime to endTime. Forty percent of the purchase amount will be sent to the `companyAddress`, while the remaining amount will be sent to the `u2winVault` address

2.1.3. U2WinVault.sol

The `U2WinVault` contract extends `Ownable`, `AccessControl` and `Pausable` contracts. `AccessControl` allows the contract to implement role-based access control mechanisms. There are 2 roles: `DEFAULT_ADMIN_ROLE` and `ADMIN_ROLE`. The contract deployer will be granted `DEFAULT_ADMIN_ROLE` and `ADMIN_ROLE`, and can assign any roles to anyone.

With `Ownable`, by default, the contract owner is contract deployer, but he can transfer ownership to another address at any time. The contract owner can add/remove the supported tokens and pause/unpause the contract. Users have to provide a signature signed by the `ADMIN_ROLE` to deposit and withdraw when the contract is not paused.

2.2. Findings

During the audit process, the audit team found 2 vulnerabilities in the given version of U2Win Smart Contract.

2.2.1. U2WinSale.sol - The `buy()` function does not mint the sufficient quantity for the buyer CRITICAL

When a user buys a number of NFT items in the same category using the `buy()` function, this function only mints a single NFT item instead of minting enough `_size` amount for the user.

```
function buy(
    uint256 _categoryId,
    uint256 _size
) external payable whenNotPaused withinTimeRange(_categoryId) {
    ...
    if (price > 0) {
        ...
        metadata.quoteToken.safeTransfer(
            address(u2winVault),
            price.sub(fee)
        );
    }
    // mint nft to msg.sender
    u2win.mint(msg.sender, _categoryId);

    emit Buy(msg.sender, _categoryId, _size, price, fee);
}
```

RECOMMENDATION

Add a for loop to mint the desired amount of NFT items (`_size`) that the user wants.

```
function buy(
    uint256 _categoryId,
    uint256 _size
) external payable whenNotPaused withinTimeRange(_categoryId) {
    ...
    if (price > 0) {
        ...
        metadata.quoteToken.safeTransfer(
            address(u2winVault),
            price.sub(fee)
        );
    }
    // mint nft to msg.sender
    for (uint256 i = 0; i < _size; i++) {
        u2win.mint(msg.sender, _categoryId);
    }
```

```
    emit Buy(msg.sender, _categoryId, _size, price, fee);
}
```

UPDATES

- *Jul 10, 2023*: This issue has been acknowledged and fixed by the U2Win team.

2.2.2. U2Win.sol - No mechanism to check if the minting has reached the `MAX_SUPPLY` or not

LOW

Users with the `MINTER_ROLE` have the ability to mint an unlimited number of NFTs.

```
...
uint256[] public MAX_SUPPLY = [0, 20000, 20000, 35000, 20000, 5000];
...
constructor() ERC721("U2Win Horse", "U2Win") {
    _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _counters[1] = 0;
    _counters[2] = _counters[1] + MAX_SUPPLY[1];
    _counters[3] = _counters[2] + MAX_SUPPLY[2];
    _counters[4] = _counters[3] + MAX_SUPPLY[3];
    _counters[5] = _counters[4] + MAX_SUPPLY[4];
}
...
function mint(
    address account,
    uint256 categoryId
) external returns (uint256) {
    require(
        hasRole(MINTER_ROLE, _msgSender()),
        "U2WinSale::mint: only MINTER_ROLE"
    );
    require(
        categoryId > 0 && categoryId <= NUMBER_OF_CATEGORIES,
        "U2WinSale::mint:: invalid categoryId"
    );
    _counters[categoryId] += 1;
    uint256 tokenId = _counters[categoryId];
    _safeMint(account, tokenId);
    nftToCategory[tokenId] = categoryId;
    return tokenId;
}
```

RECOMMENDATION

Should add a mechanism to check each time minting reaches the `MAX_SUPPLY` limit for each category.

2.3. Additional notes and recommendations

2.3.1. U2WinVault.sol - The signature for the deposit can be reused INFORMATIVE

Not blacklisting used signatures by users can result in the reuse of those signatures, allowing users to make multiple deposits.

```
function deposit(address _token, uint256 _amount, uint256 _deadline, bytes memory
_signature) external whenNotPaused {
    require(_supportedTokens.contains(_token), "U2WinVault::deposit: unsupported token");
    require(_amount > 0, "U2WinVault::deposit: invalid amount");
    require(_deadline >= block.timestamp, "U2WinVault::deposit: signature expired");
    // verify signature
    bytes32 message = keccak256(abi.encodePacked(_token, msg.sender, _amount, _deadline));
    address signer = ECDSA.recover(message.toEthSignedMessageHash(), _signature);
    require(hasRole(ADMIN_ROLE, signer), "U2WinVault::deposit: invalid signature");
    // transfer token to vault
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);

    emit Deposit(_token, msg.sender, _amount, _deadline);
}
```

2.3.2. U2WinSale.sol - Redundant logic INFORMATIVE

Instead of transferring the token to the contract and then forwarding that amount to two other addresses, we can transfer the token directly to those two addresses.

```
function buy(
    uint256 _categoryId,
    uint256 _size
) external payable whenNotPaused withinTimeRange(_categoryId) {
    ...
    if (price > 0) {
        // transfer quote token to this contract
        metadata.quoteToken.safeTransferFrom(
            msg.sender,
            address(this),
            price
        ); // redundant logic
        // transfer fee to company address
        metadata.quoteToken.safeTransfer(companyAddress, fee);
        // transfer quote token to u2win vault to distribute marketing revenue
        metadata.quoteToken.safeTransfer(
            address(u2winVault),
            price.sub(fee)
        );
    }
}
```

RECOMMENDATION

Remove the redundant logic.

```
function buy(
    uint256 _categoryId,
    uint256 _size
) external payable whenNotPaused withinTimeRange(_categoryId) {
    ...
    uint256 price = metadata.price.mul(_size);
    uint256 fee = price.mul(COMPANY_BPS).div(10000);
    if (price > 0) {
        // transfer fee to company address
        metadata.quoteToken.safeTransferFrom(
            msg.sender,
            companyAddress,
            fee
        );
        // transfer quote token to u2win vault to distribute marketing revenue
        metadata.quoteToken.safeTransferFrom(
            msg.sender,
            address(u2winVault),
            price.sub(fee)
        );
    }
}
```

UPDATES

- *Jul 10, 2023*: This issue has been acknowledged and fixed by the U2Win team.

2.3.3. U2WinVault.sol - Users can bypass checking when withdrawing if the contract owner renounces ownership INFORMATIVE

If the contract owner renounces ownership and sets the owner address to zero, the condition `signer == owner()` can be bypassed with invalid signature.

```
function withdraw(
    address _token,
    uint256 _amount,
    string memory _txId,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external whenNotPaused {
    require(
        _supportedTokens.contains(_token),
        "U2WinVault::withdraw: unsupported token"
    );
```

```
    require(_amount > 0, "U2WinVault::withdraw: invalid amount");
    require(!txIds[_txId], "U2WinVault::withdraw: txId already used");
    require(
        _deadline >= block.timestamp,
        "U2WinVault::withdraw: signature expired"
    );
    txIds[_txId] = true;
    // verify signature
    bytes32 message = keccak256(
        abi.encodePacked(_token, msg.sender, _amount, _txId, _deadline)
    );
    address signer = _recoverAddress(message, _v, _r, _s);
    require(signer == owner(), "U2WinVault::deposit: invalid signature");
    // transfer token to user
    IERC20(_token).safeTransfer(msg.sender, _amount);

    emit Withdraw(_token, msg.sender, _amount, _txId, _deadline);
}

function _recoverAddress(
    bytes32 _message,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) internal pure returns (address) {
    return ecrecover(_message, _v, _r, _s);
}
```

RECOMMENDATION

Should use the ECDSA library from OpenZeppelin.

UPDATES

- *Jul 10, 2023*: This issue has been acknowledged and fixed by the U2Win team.

2.3.4. Redundancy of events INFORMATIVE

Affected files:

- U2WinSale.sol
- U2WinVault.sol

Event `Pause()` and `Unpause()` are already defined and emitted in the Openzeppelin library, so there is no need to define and emit again.

UPDATES

- *Jul 10, 2023*: This issue has been acknowledged and fixed by the U2Win team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jul 10, 2023*	Public Report	Verichains Lab

Table 2. Report versions history