Smart contract security audit

Wintoken

v.1.2

Table of Contents

1.0 Introduction

1.1 Project engagement

During December of 2022, Wintoken engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. Wintoken provided CTDSec with access to their code repository and whitepaper.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the network's fast-paced and rapidly changing environment, we at CTDSec recommend that Wintoken team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the Wintoken contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

**FlattenedContracts_win.zip [SHA256] -
000936127e93c720c03026ccce090de04f783182d04d909fed1d541e22bbb8b1**

Fix version: FlattenedFiles.zip [SHA256] -
e3d31ee0290a6dcb3a43ac32ed1085b99387baf8a9a88eb91eb0f8e4a10dd9ae

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.	Checking status
1	Compiler warnings.	FIXED
2	Race conditions and Reentrancy. Cross-function race conditions.	PASSED
3	Possible delays in data delivery.	PASSED
4	Oracle calls.	PASSED
5	Front running.	PASSED
6	Timestamp dependence.	PASSED
7	Integer Overflow and Underflow.	PASSED
8	DoS with Revert.	PASSED
9	DoS with block gas limit.	PASSED
10	Methods execution permissions.	FIXED
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.	PASSED
12	The impact of the exchange rate on the logic.	PASSED
13	Private user data leaks.	PASSED
14	Malicious Event log.	PASSED
15	Scoping and Declarations.	PASSED
16	Uninitialized storage pointers.	PASSED

17	Arithmetic accuracy.	PASSED
18	Design Logic.	FIXED
19	Cross-function race conditions.	PASSED
20	Safe Zeppelin module.	PASSED
21	Fallback function security.	PASSED
22	Overpowered functions / Owner privileges	PASSED

3.0 Security Issues

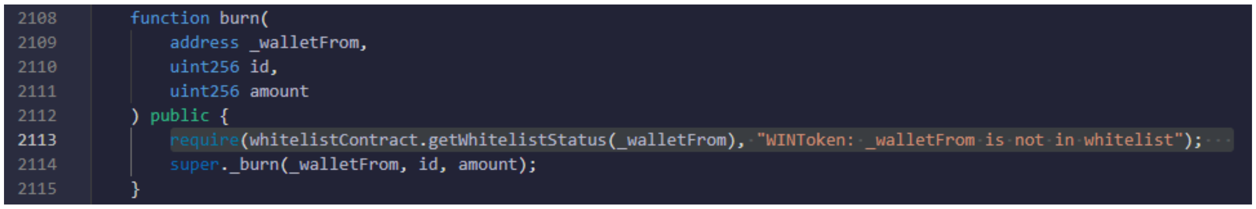
3.1 High severity issues [2]

WINtoken.sol

1. Insufficient authorization control

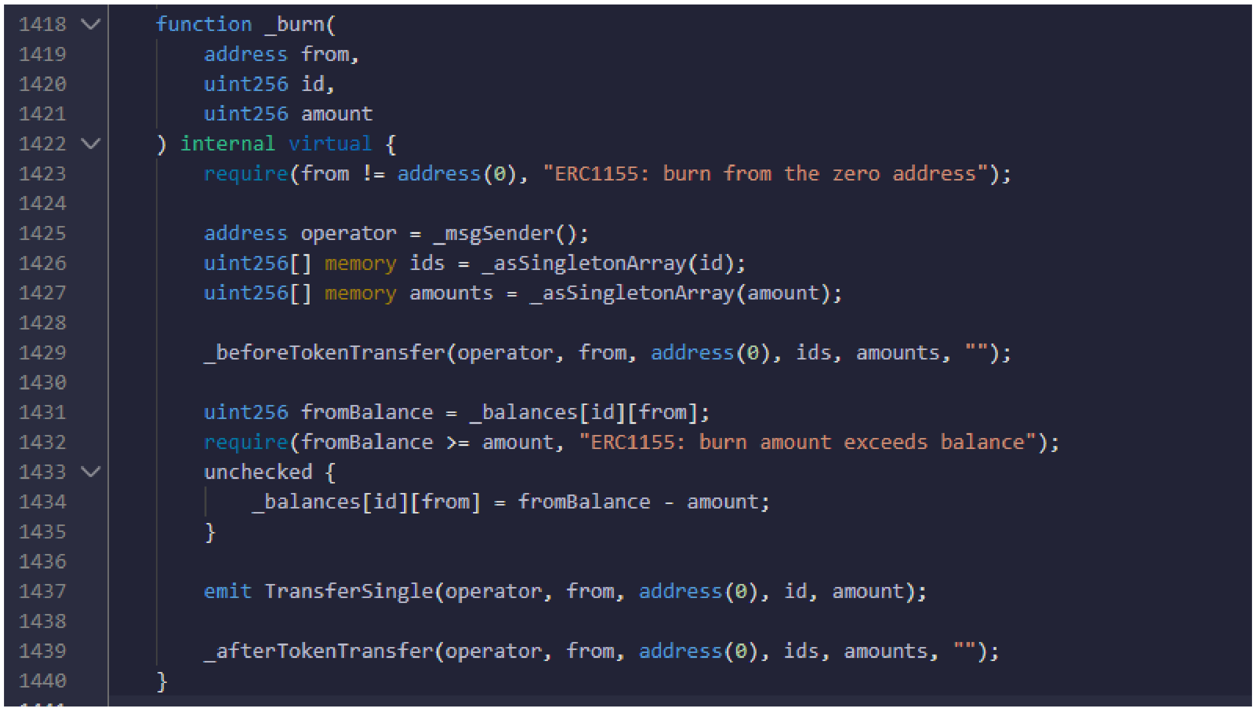
In the Burn() function, any user whose _walletfrom is included in the Whitelistcontract can burn tokens from any other wallet.

Solution: Implement a control to verify the authorization roles to prevent unauthorized users from burning tokens.



```
function burn(
    address _walletFrom,
    uint256 id,
    uint256 amount
) public {
    require(whitelistContract.getWhitelistStatus(_walletFrom), "WINToken: _walletFrom is not in whitelist");
    super._burn(_walletFrom, id, amount);
}
```

Fix: Burning function now has the required authorization controls:

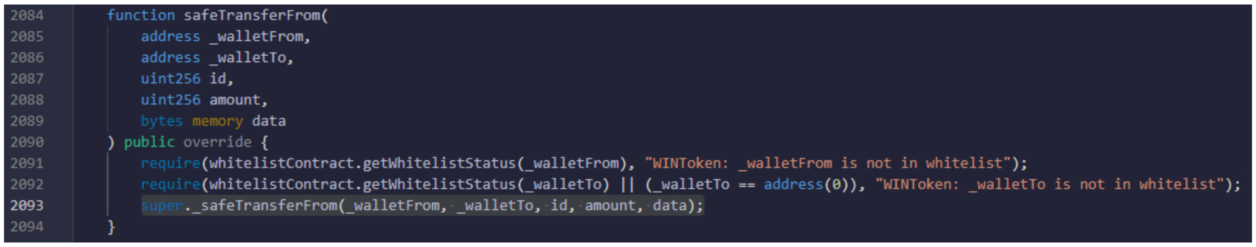


```
function _burn(
    address from,
    uint256 id,
    uint256 amount
) internal virtual {
    require(from != address(0), "ERC1155: burn from the zero address");

    address operator = _msgSender();
    uint256[] memory ids = _asSingletonArray(id);
    uint256[] memory amounts = _asSingletonArray(amount);

    _beforeTokenTransfer(operator, from, address(0), ids, amounts, "");

    uint256 fromBalance = _balances[id][from];
    require(fromBalance >= amount, "ERC1155: burn amount exceeds balance");
    unchecked {
        _balances[id][from] = fromBalance - amount;
    }

    emit TransferSingle(operator, from, address(0), id, amount);

    _afterTokenTransfer(operator, from, address(0), ids, amounts, "");
}
```

2. Lack of sender approval verification

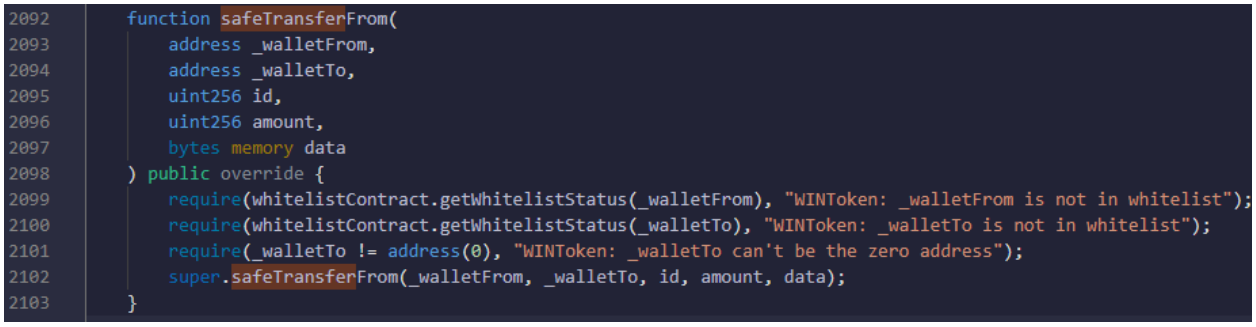
The SafeTransforFrom() function is being called incorrectly (internally), which allows users to transfer funds from wallets that they don't own, as sender approvals are not being checked.

Solution: Call super.safeTransferFrom() instead of super._safeTransferFrom() to ensure that sender approvals are being verified before transferring the funds.



```
function safeTransferFrom(
    address _walletFrom,
    address _walletTo,
    uint256 id,
    uint256 amount,
    bytes memory data
) public override {
    require(whitelistContract.getWhitelistStatus(_walletFrom), "WINToken: _walletFrom is not in whitelist");
    require(whitelistContract.getWhitelistStatus(_walletTo) || (_walletTo == address(0)), "WINToken: _walletTo is not in whitelist");
    super._safeTransferFrom(_walletFrom, _walletTo, id, amount, data);
}
```

Fix: SafeTransferFrom is correctly called now:



```
function safeTransferFrom(
    address _walletFrom,
    address _walletTo,
    uint256 id,
    uint256 amount,
    bytes memory data
) public override {
    require(whitelistContract.getWhitelistStatus(_walletFrom), "WINToken: _walletFrom is not in whitelist");
    require(whitelistContract.getWhitelistStatus(_walletTo), "WINToken: _walletTo is not in whitelist");
    require(_walletTo != address(0), "WINToken: _walletTo can't be the zero address");
    super.safeTransferFrom(_walletFrom, _walletTo, id, amount, data);
}
```

3.2 Medium severity issues [3]

WINtoken.sol

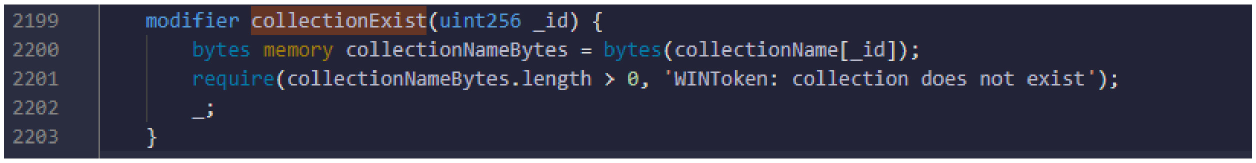
1. Inadequate collection check

The createCollection() function in the contract is used to create new collections. However, the CollectionExist() and collectionNotExist() functions only check if the length of the collection is greater than 0, which can cause issues if an empty collection is allocated to an existing one.

For example: ID collection 2 [string "hello"] and ID collection 2 [string ""].

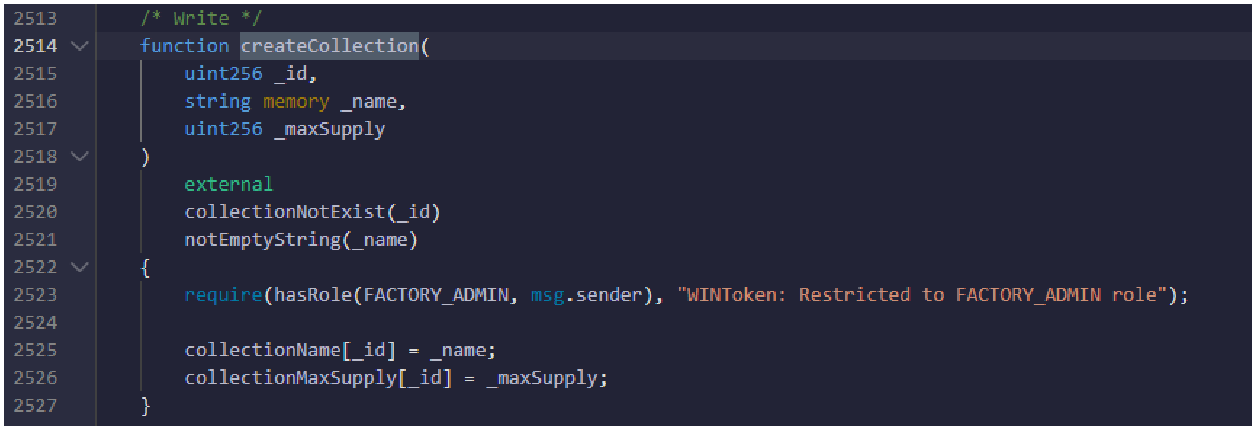
This behavior can disrupt the contract's logic and create security vulnerabilities.

Solution: Use the notEmptyString(_name) function to address this issue.



```
modifier collectionExist(uint256 _id) {
    bytes memory collectionNameBytes = bytes(collectionName[_id]);
    require(collectionNameBytes.length > 0, 'WINToken: collection does not exist');
    _;
}
```

Fix: createCollection() function is now checking that the string is not empty:



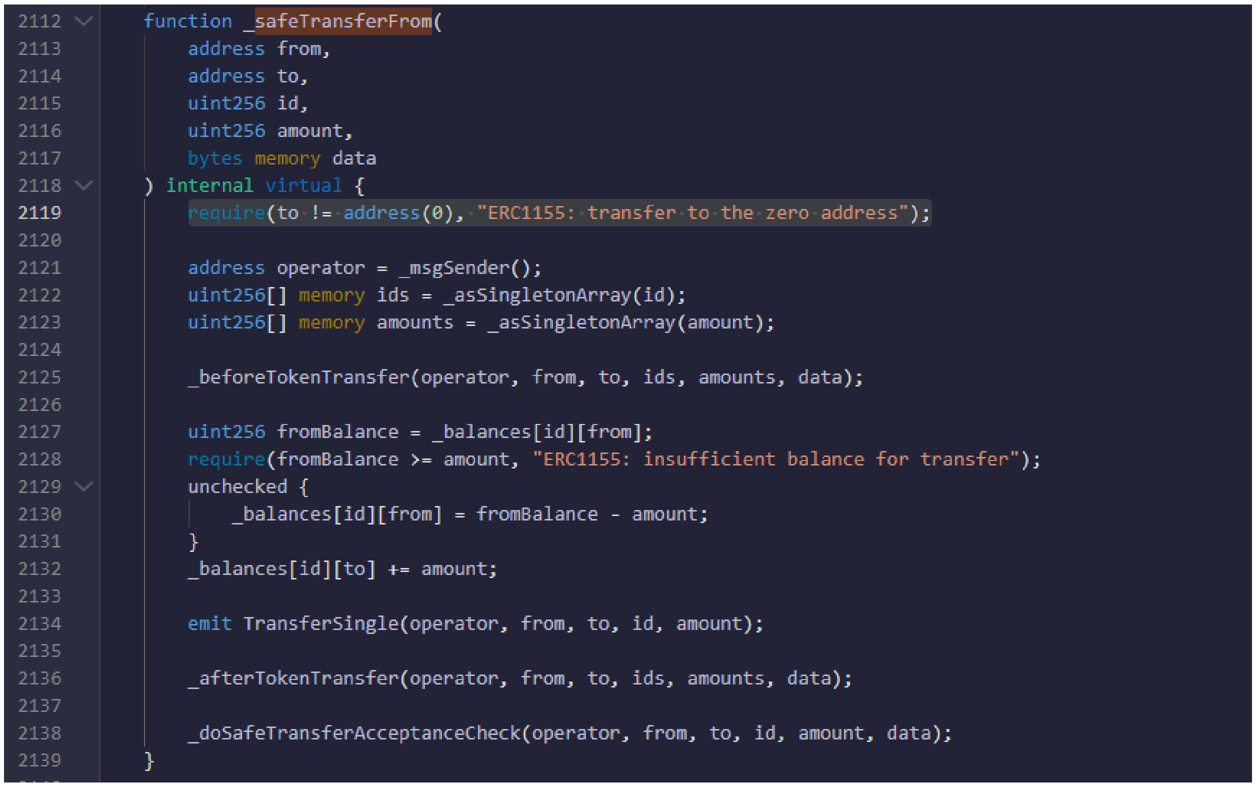
```
    /* Write */
    function createCollection(
        uint256 _id,
        string memory _name,
        uint256 _maxSupply
    )
        external
        collectionNotExist(_id)
        notEmptyString(_name)
    {
        require(hasRole(FACTORY_ADMIN, msg.sender), "WINToken: Restricted to FACTORY_ADMIN role");

        collectionName[_id] = _name;
        collectionMaxSupply[_id] = _maxSupply;
    }
```

2. Incorrect require - safeTransferFrom()

The function safeTransferFrom() adds a condition that check if _walletTo == address(0) causing a revert.

<u>Solution: Require _walletTo != address(0).</u>

<u>Fix: Require fixed:</u>



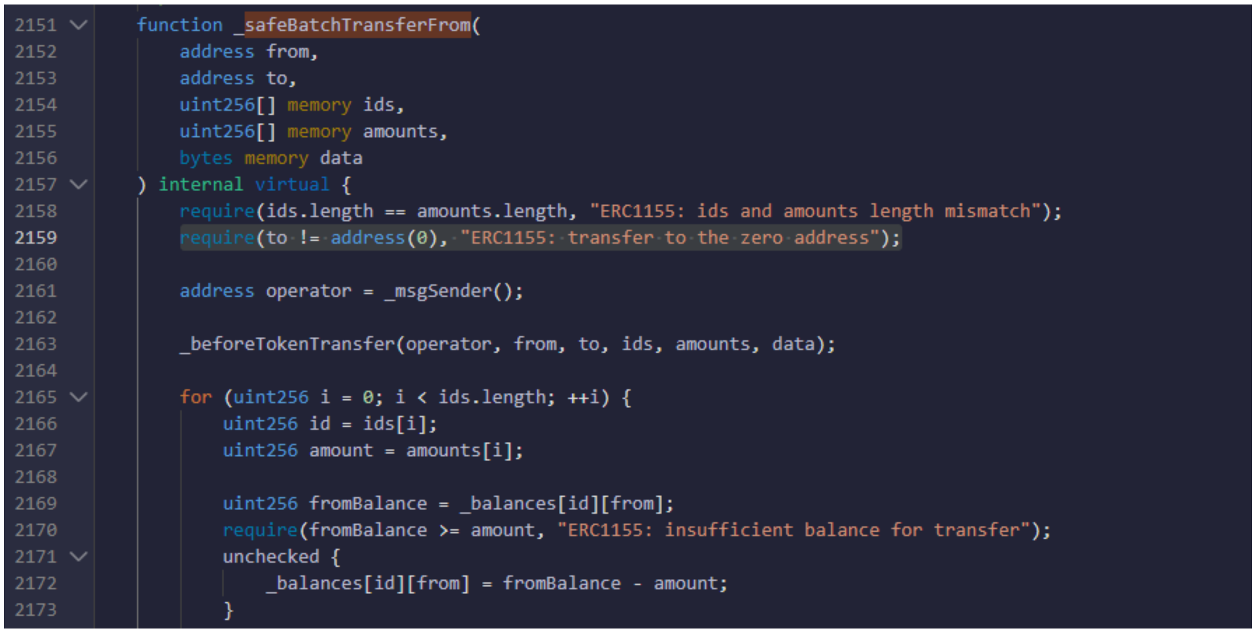
```
    function _safeTransferFrom(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) internal virtual {
        require(to != address(0), "ERC1155: transfer to the zero address");

        address operator = _msgSender();
        uint256[] memory ids = _asSingletonArray(id);
        uint256[] memory amounts = _asSingletonArray(amount);

        _beforeTokenTransfer(operator, from, to, ids, amounts, data);

        uint256 fromBalance = _balances[id][from];
        require(fromBalance >= amount, "ERC1155: insufficient balance for transfer");
        unchecked {
            _balances[id][from] = fromBalance - amount;
        }
        _balances[id][to] += amount;

        emit TransferSingle(operator, from, to, id, amount);

        _afterTokenTransfer(operator, from, to, ids, amounts, data);

        _doSafeTransferAcceptanceCheck(operator, from, to, id, amount, data);
    }
```

3. Incorrect require - safeBatchTransferFrom()

The function safeBatchTransferFrom() adds a condition that check if _walletTo == address(0) causing a revert.

Solution: Require _walletTo != address(0).

Fix: Require fixed:



```
function _safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");
    require(to != address(0), "ERC1155: transfer to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, to, ids, amounts, data);

    for (uint256 i = 0; i < ids.length; ++i) {
        uint256 id = ids[i];
        uint256 amount = amounts[i];

        uint256 fromBalance = _balances[id][from];
        require(fromBalance >= amount, "ERC1155: insufficient balance for transfer");
        unchecked {
            _balances[id][from] = fromBalance - amount;
        }
```

3.3 Low severity issues [3]

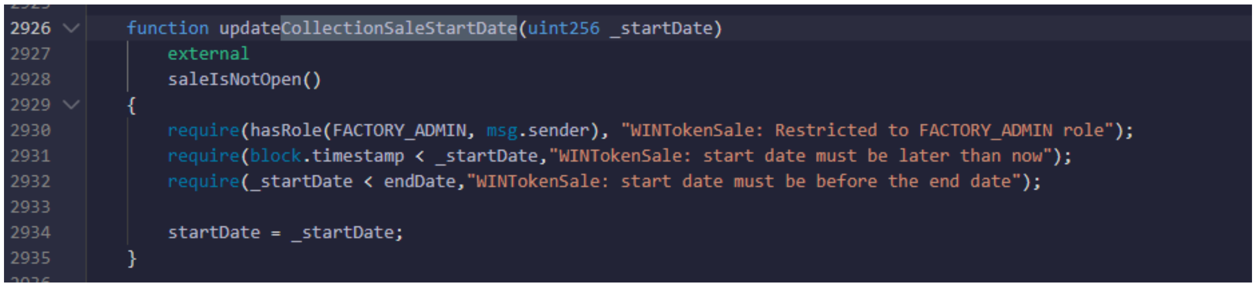
WinTokenSale.sol

1. The start date condition is not being verified

The function updateCollectionSaleStartDate() is not verifying whether the startDate is before the endDate.

Solution: Add a requirement to check that _startDate < _endDate.

Fix: updatecollectionSaleStartDate now is verifying the dates before the endDate.



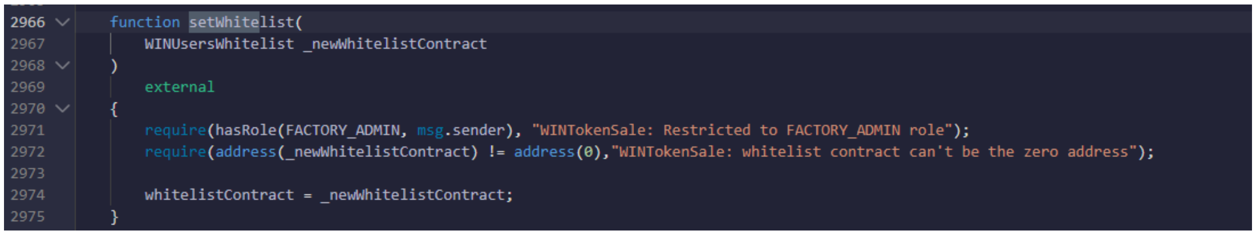
```solidity
function updateCollectionSaleStartDate(uint256 _startDate)
    external
    saleIsNotOpen()
{
    require(hasRole(FACTORY_ADMIN, msg.sender), "WINTokenSale: Restricted to FACTORY_ADMIN role");
    require(block.timestamp < _startDate,"WINTokenSale: start date must be later than now");
    require(_startDate < endDate,"WINTokenSale: start date must be before the end date");

    startDate = _startDate;
}
```

2. Lack of verification for zero address in two functions

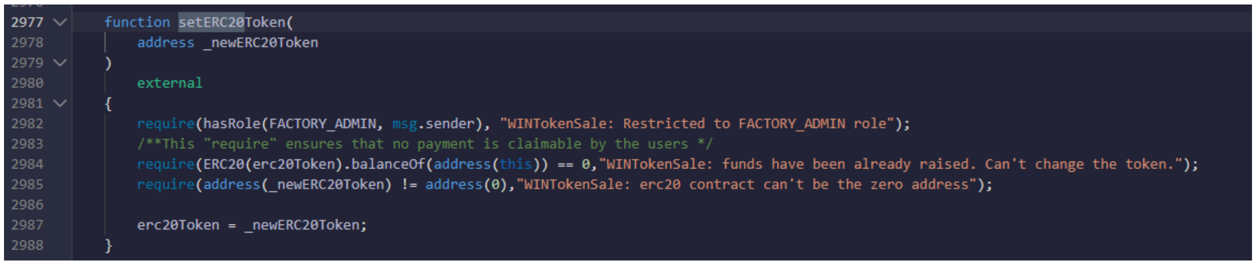
The setWhitelist() and setERC20Token() functions do not verify zero addresses.

<u>Solution: Add a requirement to verify that _newERC20Token != address(0) and _newWhitelistContract != address(0).</u>

<u>Fix: Both functions are checking zero address now:</u>



```solidity
function setWhitelist(
    WINUsersWhitelist _newWhitelistContract
)
    external
{
    require(hasRole(FACTORY_ADMIN, msg.sender), "WINTokenSale: Restricted to FACTORY_ADMIN role");
    require(address(_newWhitelistContract) != address(0),"WINTokenSale: whitelist contract can't be the zero address");

    whitelistContract = _newWhitelistContract;
}
```



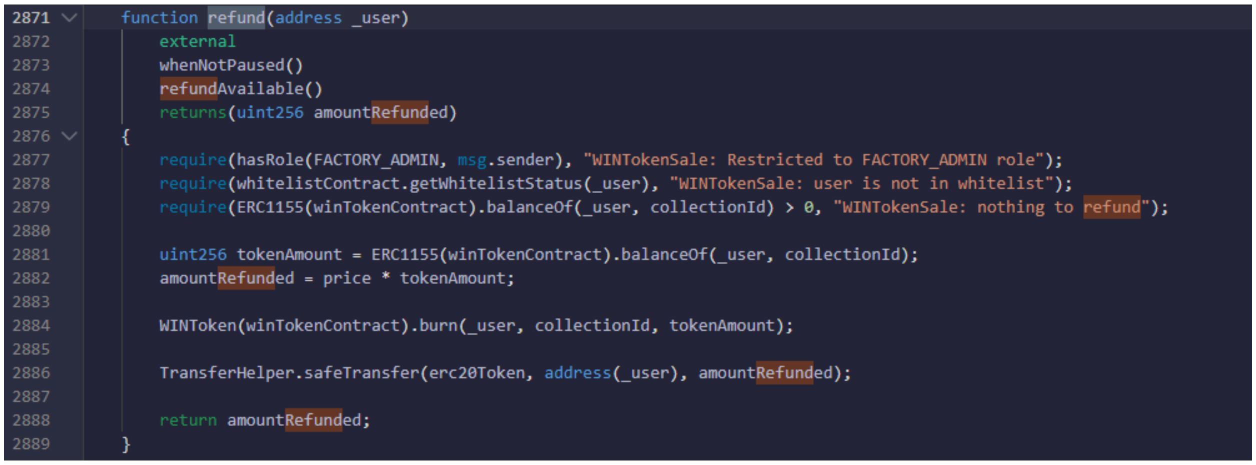
```solidity
function setERC20Token(
    address _newERC20Token
)
    external
{
    require(hasRole(FACTORY_ADMIN, msg.sender), "WINTokenSale: Restricted to FACTORY_ADMIN role");
    /**This "require" ensures that no payment is claimable by the users */
    require(ERC20(erc20Token).balanceOf(address(this)) == 0,"WINTokenSale: funds have been already raised. Can't change the token.");
    require(address(_newERC20Token) != address(0),"WINTokenSale: erc20 contract can't be the zero address");

    erc20Token = _newERC20Token;
}
```

3. Insufficient balance verification before transaction

The refund() function should verify the balances before transferring tokens.

<u>Solution: Add a requirement to check that ERC1155(winTokenContract).balanceOf(_user, collectionId) is less than 0.</u>

<u>Fix: Refund balances are checked now:</u>

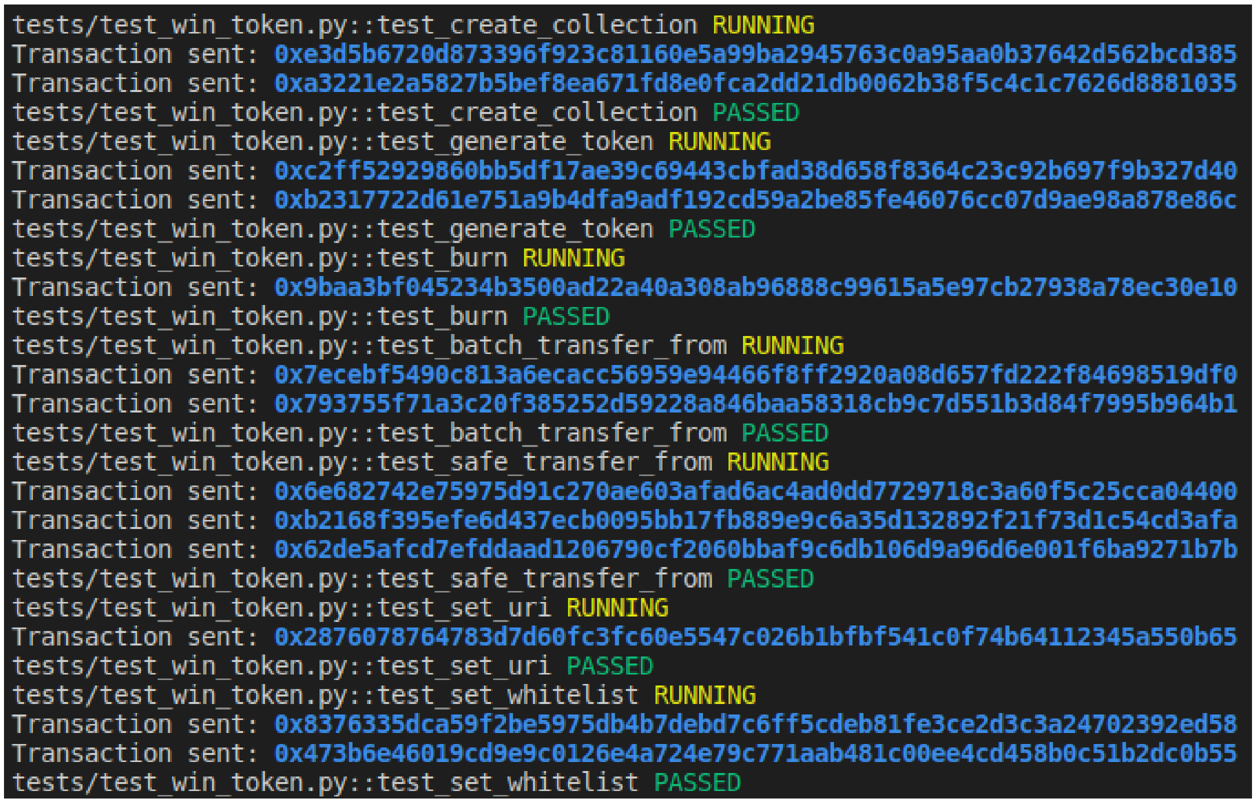


```
function refund(address _user)
    external
    whenNotPaused()
    refundAvailable()
    returns(uint256 amountRefunded)
{
    require(hasRole(FACTORY_ADMIN, msg.sender), "WINTokenSale: Restricted to FACTORY_ADMIN role");
    require(whitelistContract.getWhitelistStatus(_user), "WINTokenSale: user is not in whitelist");
    require(ERC1155(winTokenContract).balanceOf(_user, collectionId) > 0, "WINTokenSale: nothing to refund");

    uint256 tokenAmount = ERC1155(winTokenContract).balanceOf(_user, collectionId);
    amountRefunded = price * tokenAmount;

    WINToken(winTokenContract).burn(_user, collectionId, tokenAmount);

    TransferHelper.safeTransfer(erc20Token, address(_user), amountRefunded);

    return amountRefunded;
}
```

4.0 Testing coverage - python

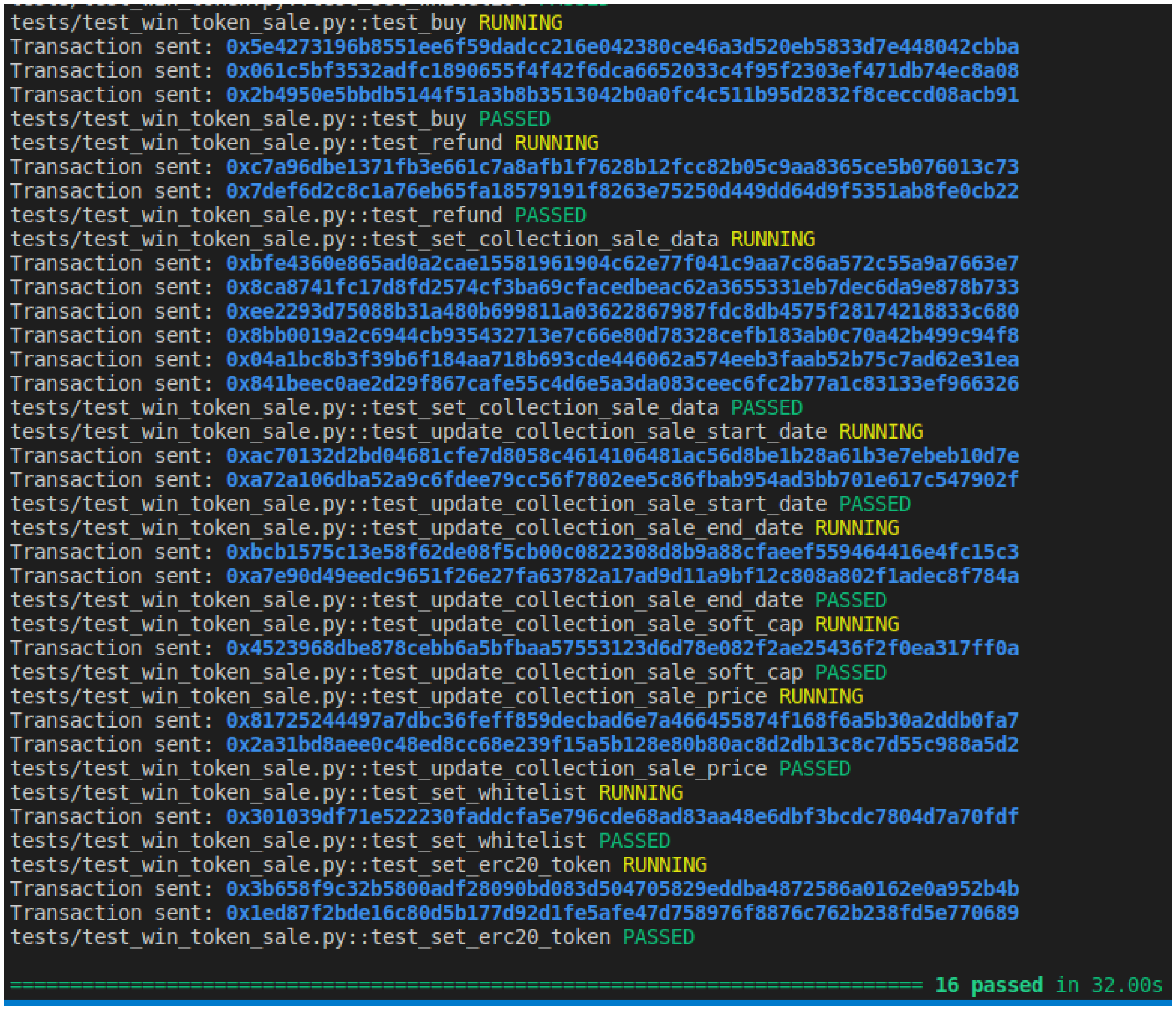
During the testing phase, custom use cases were written to cover all the logic of contracts in python language. *Check “5 Annexes” to see the testing code.*

Wintoken tests:



```
tests/test_win_token.py::test_create_collection RUNNING
Transaction sent: 0xe3d5b6720d873396f923c81160e5a99ba2945763c0a95aa0b37642d562bcd385
Transaction sent: 0xa3221e2a5827b5bef8ea671fd8e0fca2dd21db0062b38f5c4c1c7626d8881035
tests/test_win_token.py::test_create_collection PASSED
tests/test_win_token.py::test_generate_token RUNNING
Transaction sent: 0xc2ff52929860bb5df17ae39c69443cbfad38d658f8364c23c92b697f9b327d40
Transaction sent: 0xb2317722d61e751a9b4dfa9adf192cd59a2be85fe46076cc07d9ae98a878e86c
tests/test_win_token.py::test_generate_token PASSED
tests/test_win_token.py::test_burn RUNNING
Transaction sent: 0x9baa3bf045234b3500ad22a40a308ab96888c99615a5e97cb27938a78ec30e10
tests/test_win_token.py::test_burn PASSED
tests/test_win_token.py::test_batch_transfer_from RUNNING
Transaction sent: 0x7ecebf5490c813a6ecacc56959e94466f8ff2920a08d657fd222f84698519df0
Transaction sent: 0x793755f71a3c20f385252d59228a846baa58318cb9c7d551b3d84f7995b964b1
tests/test_win_token.py::test_batch_transfer_from PASSED
tests/test_win_token.py::test_safe_transfer_from RUNNING
Transaction sent: 0x6e682742e75975d91c270ae603afad6ac4ad0dd7729718c3a60f5c25cca04400
Transaction sent: 0xb2168f395efe6d437ecb0095bb17fb889e9c6a35d132892f21f73d1c54cd3afa
Transaction sent: 0x62de5afcd7efddaad1206790cf2060bbaf9c6db106d9a96d6e001f6ba9271b7b
tests/test_win_token.py::test_safe_transfer_from PASSED
tests/test_win_token.py::test_set_uri RUNNING
Transaction sent: 0x2876078764783d7d60fc3fc60e5547c026b1bfbf541c0f74b64112345a550b65
tests/test_win_token.py::test_set_uri PASSED
tests/test_win_token.py::test_set_whitelist RUNNING
Transaction sent: 0x8376335dca59f2be5975db4b7debd7c6ff5cdeb81fe3ce2d3c3a24702392ed58
Transaction sent: 0x473b6e46019cd9e9c0126e4a724e79c771aab481c00ee4cd458b0c51b2dc0b55
tests/test_win_token.py::test_set_whitelist PASSED
```

Wintoken sale tests:


```
tests/test_win_token_sale.py::test_buy RUNNING
Transaction sent: 0x5e4273196b8551ee6f59dadcc216e042380ce46a3d520eb5833d7e448042cbba
Transaction sent: 0x061c5bf3532adfc1890655f4f42f6dca6652033c4f95f2303ef471db74ec8a08
Transaction sent: 0x2b4950e5bbdb5144f51a3b8b3513042b0a0fc4c511b95d2832f8ceccd08acb91
tests/test_win_token_sale.py::test_buy PASSED
tests/test_win_token_sale.py::test_refund RUNNING
Transaction sent: 0xc7a96dbe1371fb3e661c7a8afb1f7628b12fcc82b05c9aa8365ce5b076013c73
Transaction sent: 0x7def6d2c8c1a76eb65fa18579191f8263e75250d449dd64d9f5351ab8fe0cb22
tests/test_win_token_sale.py::test_refund PASSED
tests/test_win_token_sale.py::test_set_collection_sale_data RUNNING
Transaction sent: 0xbfe4360e865ad0a2cae15581961904c62e77f041c9aa7c86a572c55a9a7663e7
Transaction sent: 0x8ca8741fc17d8fd2574cf3ba69cfacedbeac62a3655331eb7dec6da9e878b733
Transaction sent: 0xee2293d75088b31a480b699811a03622867987fdc8db4575f28174218833c680
Transaction sent: 0x8bb0019a2c6944cb935432713e7c66e80d78328cefb183ab0c70a42b499c94f8
Transaction sent: 0x04a1bc8b3f39b6f184aa718b693cde446062a574eeb3faab52b75c7ad62e31ea
Transaction sent: 0x841beec0ae2d29f867cafe55c4d6e5a3da083ceec6fc2b77a1c83133ef966326
tests/test_win_token_sale.py::test_set_collection_sale_data PASSED
tests/test_win_token_sale.py::test_update_collection_sale_start_date RUNNING
Transaction sent: 0xac70132d2bd04681cfe7d8058c4614106481ac56d8be1b28a61b3e7ebeb10d7e
Transaction sent: 0xa72a106dba52a9c6fdee79cc56f7802ee5c86fbab954ad3bb701e617c547902f
tests/test_win_token_sale.py::test_update_collection_sale_start_date PASSED
tests/test_win_token_sale.py::test_update_collection_sale_end_date RUNNING
Transaction sent: 0xbcb1575c13e58f62de08f5cb00c0822308d8b9a88cfaeef559464416e4fc15c3
Transaction sent: 0xa7e90d49eedc9651f26e27fa63782a17ad9d11a9bf12c808a802f1adec8f784a
tests/test_win_token_sale.py::test_update_collection_sale_end_date PASSED
tests/test_win_token_sale.py::test_update_collection_sale_soft_cap RUNNING
Transaction sent: 0x4523968dbe878cebb6a5bfbaa57553123d6d78e082f2ae25436f2f0ea317ff0a
tests/test_win_token_sale.py::test_update_collection_sale_soft_cap PASSED
tests/test_win_token_sale.py::test_update_collection_sale_price RUNNING
Transaction sent: 0x81725244497a7dbc36feff859decbad6e7a466455874f168f6a5b30a2ddb0fa7
Transaction sent: 0x2a31bd8aee0c48ed8cc68e239f15a5b128e80b80ac8d2db13c8c7d55c988a5d2
tests/test_win_token_sale.py::test_update_collection_sale_price PASSED
tests/test_win_token_sale.py::test_set_whitelist RUNNING
Transaction sent: 0x301039df71e522230faddcfa5e796cde68ad83aa48e6dbf3bcdc7804d7a70fdf
tests/test_win_token_sale.py::test_set_whitelist PASSED
tests/test_win_token_sale.py::test_set_erc20_token RUNNING
Transaction sent: 0x3b658f9c32b5800adf28090bd083d504705829eddba4872586a0162e0a952b4b
Transaction sent: 0x1ed87f2bde16c80d5b177d92d1fe5afe47d758976f8876c762b238fd5e770689
tests/test_win_token_sale.py::test_set_erc20_token PASSED

=========================================================================== 16 passed in 32.00s
```

5.0 Annexes

Wintoken attack code:

```
from brownie import reverts, WINToken

from scripts.helpful_scripts import (
    ZERO_ADDRESS,
    get_account,
    get_factory_role,
)

from scripts.deploy import (
    deploy_win_token,
    deploy_win_users_whitelist,
    deploy_win_token_with_default_collection,
    add_to_whitelist
)

def test_create_collection(only_local):
    # arrage
    owner = get_account(0)
    win_token = deploy_win_token(owner)

    # assert
    with reverts("WINToken: Restricted to FACTORY_ADMIN role"):
        win_token.createCollection(1, "A", 1000, {"from": owner})
    win_token.grantRole(get_factory_role(), owner)
    win_token.createCollection(1, "A", 1000, {"from": owner})
    with reverts("WINToken: collection already exists"):
        win_token.createCollection(1, "A", 1000, {"from": owner})


def test_generate_token(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)

    win_token, whitelist = deploy_win_token_with_default_collection(owner)

    # assert
    with reverts("WINToken: Restricted to FACTORY_ADMIN role"):
        win_token.generateToken(other, 1, 100, {"from": other})
    win_token.grantRole(get_factory_role(), owner)
    with reverts("WINToken: _wallet is not in whitelist"):
        win_token.generateToken(other, 1, 100, {"from": owner})

    add_to_whitelist(owner, whitelist, [other])
    win_token.generateToken(other, 1, 100, {"from": owner})

def test_burn(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    extra = get_account(1)

    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    with reverts("WINToken: _walletFrom is not in whitelist"):
        win_token.burn(other, 1, 10, {"from": owner})
    add_to_whitelist(owner, whitelist, [other])
    win_token.generateToken(other, 1, 100, {"from": owner})
    assert win_token.balanceOf(other, 1) == 100
    win_token.burn(other, 1, 10, {"from": extra})
    assert win_token.balanceOf(other, 1) == 90

def test_batch_transfer_from(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    extra = get_account(2)
    another = get_account(3)
    hello_data = '0x68656c6c6f000000000000000000000'

    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    with reverts("WINToken: _walletFrom is not in whitelist"):
        win_token.safeBatchTransferFrom(other, extra, [1], [10],
hello_data, {"from": owner})

    add_to_whitelist(owner, whitelist, [other])
    with reverts("WINToken: _walletTo is not in whitelist"):
        win_token.safeBatchTransferFrom(other, extra, [1], [10],
hello_data, {"from": owner})
```

```
    add_to_whitelist(owner, whitelist, [extra])
    win_token.generateToken(other, 1, 100, {"from": owner})
    assert win_token.balanceOf(other, 1) == 100
    assert win_token.balanceOf(extra, 1) == 0
    win_token.safeBatchTransferFrom(other, extra, [1], [10], hello_data,
{"from": other})
    assert win_token.balanceOf(other, 1) == 90
    assert win_token.balanceOf(extra, 1) == 10

def test_safe_transfer_from(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    extra = get_account(2)
    hello_data = '0x68656c6c6f0000000000000000000000'

    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    with reverts("WINToken: _walletFrom is not in whitelist"):
        win_token.safeTransferFrom(other, extra, 1, 10, hello_data,
{"from": owner})

    add_to_whitelist(owner, whitelist, [other])
    with reverts("WINToken: _walletTo is not in whitelist"):
        win_token.safeTransferFrom(other, extra, 1, 10, hello_data,
{"from": owner})

    with reverts():
        win_token.safeTransferFrom(other, ZERO_ADDRESS, 1, 10, hello_data,
{"from": owner})


    add_to_whitelist(owner, whitelist, [extra])
    win_token.generateToken(other, 1, 100, {"from": owner})
    assert win_token.balanceOf(other, 1) == 100
    assert win_token.balanceOf(extra, 1) == 0
    win_token.safeTransferFrom(other, extra, 1, 10, hello_data, {"from":
other})
    assert win_token.balanceOf(other, 1) == 90
    assert win_token.balanceOf(extra, 1) == 10

def test_set_uri(only_local):
    # arrage
    owner = get_account(0)
    win_token = deploy_win_token(owner)

    # assert
    with reverts("WINToken: Restricted to FACTORY_ADMIN role"):
        win_token.setURI("new_uri", {"from": owner})
    win_token.grantRole(get_factory_role(), owner)

    assert win_token.uri(1) == "_some_base_uri_1"
    win_token.setURI("new_uri_", {"from": owner})
    assert win_token.uri(1) == "new_uri_1"

def test_set_whitelist(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    white = deploy_win_users_whitelist(owner)
    win_token =  WINToken.deploy("_some_base_uri_", white.address, {"from":
owner})

    # assert
    with reverts("WINToken: Restricted to FACTORY_ADMIN role"):
        win_token.setWhitelist(other, {"from": owner})
    win_token.grantRole(get_factory_role(), owner)
    with reverts("WINToken: whitelist contract can't be zero address"):
        win_token.setWhitelist(ZERO_ADDRESS, {"from": owner})

    assert win_token.getWhitelistContractAddress() == white.address
    new_whitelist = deploy_win_users_whitelist(owner)
    win_token.setWhitelist(new_whitelist.address, {"from": owner})
    assert win_token.getWhitelistContractAddress() == new_whitelist.address
```

Wintoken Sale Attack Code:

```
from brownie import reverts, WINTokenSale

from scripts.helpful_scripts import (
    ZERO_ADDRESS,
    get_account,
    get_timestamp,
    evm_increase_time
)

from scripts.deploy import (
    deploy_test_erc20,
    deploy_win_users_whitelist,
    deploy_win_token_sale,
    deploy_win_token_with_default_collection,
    add_to_whitelist
)


def test_buy(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)

    erc20 = deploy_test_erc20(owner)
    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    collection_id = 1
    start_date = get_timestamp(1)
    end_date = get_timestamp(5)
    soft_cap = 1000000000000000000 # 1.00 ETH
    price = 10000000000000000 # 0.01 ETH
    win_token_sale = WINTokenSale.deploy(erc20.address, win_token.address,
        collection_id, whitelist.address, start_date, end_date,
        soft_cap, price, {"from": owner})

    evm_increase_time(172800) # increase 2 days
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.buy(other, 100, {"from": other})
    with reverts("WINTokenSale: user is not in whitelist"):
        win_token_sale.buy(other, 100, {"from": owner})
    add_to_whitelist(owner, whitelist, [other])
    with reverts("WINTokenSale: not enough tokens remaining to sale"):
        win_token_sale.buy(other, 100, {"from": owner})
    add_to_whitelist(owner, whitelist, [other, win_token_sale])
    # add tokens to WINTokenSale contract
    win_token.generateToken(win_token_sale.address, 1, 1000, {"from":
owner})
    # mint some tokens to buyer and aprove
    erc20.mint(other, price * 100, {"from": owner})
    erc20.approve(win_token_sale.address, price * 10, {"from": other})
    # Buy
    win_token_sale.buy(other, 10, {"from": owner})
    assert win_token.balanceOf(other, 1) == 10
    assert win_token.balanceOf(win_token_sale.address, 1) == 990

def test_refund(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)

    erc20 = deploy_test_erc20(owner)
    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    collection_id = 1
    start_date = get_timestamp(1)
    end_date = get_timestamp(3)
    soft_cap = 1000000000000000000 # 1.00 ETH
    price = 10000000000000000 # 0.01 ETH
    win_token_sale = WINTokenSale.deploy(erc20.address, win_token.address,
        collection_id, whitelist.address, start_date, end_date,
        soft_cap, price, {"from": owner})

    # buy some tokens before
    add_to_whitelist(owner, whitelist, [other, win_token_sale])
    evm_increase_time(86400 * 2) # increase 2 days to start sale
    win_token.generateToken(win_token_sale.address, 1, 1000, {"from":
owner})
    erc20.mint(other, price * 100, {"from": owner})
    erc20.approve(win_token_sale.address, price * 10, {"from": other})
    win_token_sale.buy(other, 10, {"from": owner})

    evm_increase_time(86400 * 4) # increase 3 days when refund available
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.refund(other, {"from": other})
    whitelist.editWhitelist([other], False, {"from": owner})
    with reverts("WINTokenSale: user is not in whitelist"):
        win_token_sale.refund(other, {"from": owner})
    add_to_whitelist(owner, whitelist, [other])
    assert win_token.balanceOf(other, 1) == 10
    win_token_sale.refund(other, {"from": owner})
    assert win_token.balanceOf(other, 1) == 0

def test_set_collection_sale_data(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    win_token_sale = deploy_win_token_sale(owner)

    start = get_timestamp(5)
    end = get_timestamp(15)

    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.setCollectionSaleData(start, end, 100, 10, {"from":
other})
    with reverts("WINTokenSale: start date must be later than now"):
        win_token_sale.setCollectionSaleData(1, end, 100, 10, {"from":
owner})
    with reverts("WINTokenSale: end date must be later than now"):
        win_token_sale.setCollectionSaleData(start, 1, 100, 10, {"from":
owner})
    with reverts("WINTokenSale: end date must be later than start date"):
        win_token_sale.setCollectionSaleData(end, start, 100, 10, {"from":
owner})
    with reverts("WINTokenSale: price must be bigger than 0"):
        win_token_sale.setCollectionSaleData(start, end, 100, 0, {"from":
owner})

    win_token_sale.setCollectionSaleData(start, end, 100, 10, {"from":
owner})
    evm_increase_time(86400 * 10) # increase 10 days

    assert win_token_sale.getCollectionSaleStartDate() == start
    assert win_token_sale.getCollectionSaleEndDate() == end
    assert win_token_sale.getCollectionSalePrice() == 10
    assert win_token_sale.getCollectionSaleSoftCap() == 100

    with reverts("WINTokenSale: sale has already started"):
        win_token_sale.setCollectionSaleData(start, end, 100, 10, {"from":
owner})

def test_update_collection_sale_start_date(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    win_token_sale = deploy_win_token_sale(owner)

    start = get_timestamp(5)

    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.updateCollectionSaleStartDate(start, {"from":
other})
    with reverts("WINTokenSale: start date must be later than now"):
        win_token_sale.updateCollectionSaleStartDate(1, {"from": owner})
    win_token_sale.updateCollectionSaleStartDate(start, {"from": owner})
    assert win_token_sale.getCollectionSaleStartDate() == start

def test_update_collection_sale_end_date(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    win_token_sale = deploy_win_token_sale(owner)

    end = get_timestamp(5)

    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.updateCollectionSaleEndDate(end, {"from": other})

    win_token_sale.updateCollectionSaleStartDate(get_timestamp(6), {"from":
owner})
    with reverts("WINTokenSale: end date must be later than start date"):
        win_token_sale.updateCollectionSaleEndDate(end, {"from": owner})

    new_end = get_timestamp(7)
    win_token_sale.updateCollectionSaleEndDate(new_end, {"from": owner})
    assert win_token_sale.getCollectionSaleEndDate() == new_end

def test_update_collection_sale_soft_cap(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    win_token_sale = deploy_win_token_sale(owner)
    old_soft_cap = 1000000000000000000 # 1.00 ETH
    new_soft_cap = 2000000000000000000 # 2.00 ETH
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.updateCollectionSaleSoftCap(new_soft_cap, {"from":
other})

    assert win_token_sale.getCollectionSaleSoftCap() == old_soft_cap
    win_token_sale.updateCollectionSaleSoftCap(new_soft_cap, {"from":
owner})
    assert win_token_sale.getCollectionSaleSoftCap() == new_soft_cap

def test_update_collection_sale_price(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    win_token_sale = deploy_win_token_sale(owner)

    old_price = 500000000000000000 # 0.50 ETH
    new_price = 100000000000000000 # 0.10 ETH
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.updateCollectionSalePrice(new_price, {"from":
other})
    with reverts("WINTokenSale: price must be bigger than 0"):
        win_token_sale.updateCollectionSalePrice(0, {"from": owner})

    assert win_token_sale.getCollectionSalePrice() == old_price
    win_token_sale.updateCollectionSalePrice(new_price, {"from": owner})
    assert win_token_sale.getCollectionSalePrice() == new_price

def test_set_whitelist(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    erc20 = deploy_test_erc20(owner)
    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    collection_id = 1
    start_date = get_timestamp(1)
    end_date = get_timestamp(2)
    soft_cap = 1000000000000000000 # 1.00 ETH
    price = 10000000000000000 # 0.01 ETH
    win_token_sale = WINTokenSale.deploy(erc20.address, win_token.address,
        collection_id, whitelist.address, start_date, end_date,
        soft_cap, price, {"from": owner})

    new_whitelist = deploy_win_users_whitelist(owner)
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.setWhitelist(new_whitelist.address, {"from": other})
    #with reverts(): #TODO Fix
        #win_token_sale.setWhitelist(ZERO_ADDRESS, {"from": owner})

    assert win_token_sale.getWhitelistContract() == whitelist.address
    win_token_sale.setWhitelist(new_whitelist.address, {"from": owner})
    assert win_token_sale.getWhitelistContract() == new_whitelist.address

def test_set_erc20_token(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    erc20 = deploy_test_erc20(owner)
    win_token, whitelist = deploy_win_token_with_default_collection(owner)
    collection_id = 1
    start_date = get_timestamp(1)
    end_date = get_timestamp(3)
    soft_cap = 1000000000000000000 # 1.00 ETH
    price = 10000000000000000 # 0.01 ETH
    win_token_sale = WINTokenSale.deploy(erc20.address, win_token.address,
        collection_id, whitelist.address, start_date, end_date,
        soft_cap, price, {"from": owner})

    # buy some tokens before
    add_to_whitelist(owner, whitelist, [other, win_token_sale])
    evm_increase_time(86400 * 2) # increase 2 days to start sale
    win_token.generateToken(win_token_sale.address, 1, 1000, {"from":
owner})
    erc20.mint(other, price * 100, {"from": owner})
    erc20.approve(win_token_sale.address, price * 10, {"from": other})
    win_token_sale.buy(other, 10, {"from": owner})

    new_erc20 = deploy_test_erc20(owner)
    # assert
    with reverts("WINTokenSale: Restricted to FACTORY_ADMIN role"):
        win_token_sale.setERC20Token(new_erc20.address, {"from": other})
    # assert
    with reverts("WINTokenSale: funds have been already raised. Can't
change the token."):
        win_token_sale.setERC20Token(new_erc20.address, {"from": owner})
    #with reverts(): #TODO Fix
        #win_token_sale.setERC20Token(ZERO_ADDRESS, {"from": owner})

    # refund to allow set new erc20 token
    evm_increase_time(86400 * 4) # increase 4 days when refund available
    win_token_sale.refund(other, {"from": owner})

    assert win_token_sale.getERC20Token() == erc20.address
    win_token_sale.setERC20Token(new_erc20.address, {"from": owner})
    assert win_token_sale.getERC20Token() == new_erc20.address
```

6.0 Summary of the audit

The cybersecurity audit report has identified critical vulnerabilities that must be reviewed prior to deployment. From ctdsec We are committed to providing you with the necessary support to resolve these issues.

Update: All issues were solved by development team and the contract is safe to be deployed.