QuillAudits

Audit Report
June, 2024

For

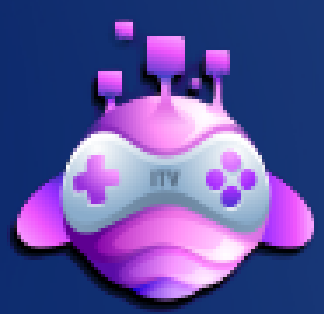


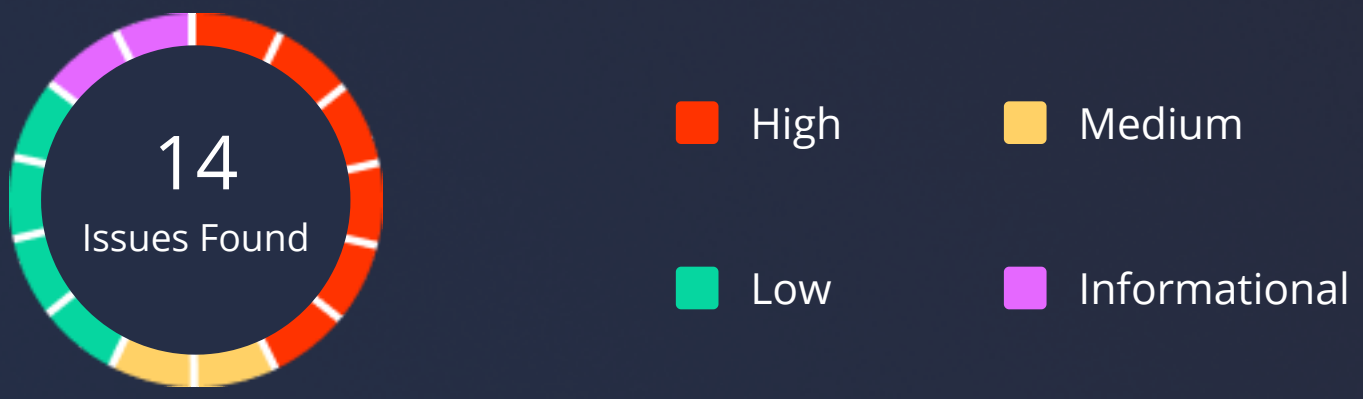
Table of Content

Table of Content

Executive Summary

Project Name	IntoTheVerse
Overview	The IntoTheVerse codebase allows a user to mint avatars that can own a maximum of 10 tree NFTs. These treeNFTs can be leveled up via staking on the GreenDonation contract. Users are able to participate in offsetting carbon emissions via the Toucan protocol by donating a portion of their rewards to buy TC02 tokens and minting NFT certificates to show their participation. Rewards are calculated and determined through the privileged address rewardsDistribution, which is authorized to call the notifyRewardAmount function that adjusts token and reward values.
Timeline	27th February, 2024 - 26th June, 2024 (audit on old main branch) From 8th March - 20th March, 2024 (audited merged changes from develop to main)
Updated Code Received	25th June 2024
Second Review	26th June 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the IntoTheVerse Token contracts for quality, security, and correctness.
Source Code	https://github.com/IntoTheVerse/IntoTheVerse-Celo-Contracts/tree/main/contracts
Branch	Main
Fixed In	e84aad9bbab92314f8398326ab69b9e13f22178c

Number of Security Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	1	0	0
Resolved Issues	6	1	4	2

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC721 API violation
- Malicious libraries
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Implementation of ERC-721A token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Manual Review, Hardhat, Slither

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. waterTree function can be manipulated by anyone to get an infinite tree level

Path

GreenDonation.sol#L174

Function

stake()

Description

Each TreeNFT has a level attribute which the tree NFT owners can only increase by staking in the GreenDonation contract which calls waterTree() in the tree contract.

```
function stake(
    uint256 tree,
    uint256 amount
) external nonReentrant updateReward(tree) onlyTreeOwner(tree, msg.sender) {
    require(amount > 0, “Cannot stake 0”);
    _totalSupply = _totalSupply.add(amount);
    _balances[tree] = _balances[tree].add(amount);
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);
    emit Staked(tree, msg.sender, amount);
    treeContract.waterTree(tree); // @audit
}
```

From the above, notice there is no specific amount required to trigger this crucial update. Also notice that there are no time periods or gaps preventing the next level up from happening immediately after another. This means with the most insignificant amount of stakingToken, an owner can call stake() multiple times and exponentially increase their treeNFT level in one transaction without any significant amount staked causing them to manipulate the system and get infinite tree level.

Consider this Attack scenario:

- A user deploys his malicious contract which calls GreenDonation.sol:stake() multiple times in a loop and transfers ownership of all his tree NFT to the contract
- The user begin attack with contract, calling stake() multiple times with the smallest unit of stakingToken all in one transaction
- Transaction gets executed and user treeNFT level instantly increase drastically
- This is performed multiple times causing user to get infinite tree level

POC

```
it.only(‘ - Should level up whaleTree faster than ownerTree with the same cost’, async ()
=> {
    const { greenDonation, treeContract, owner, whale, ETH, rewardToken, stakingToken}
= await loadFixture(fetchFixtures)
    await rewardToken.mint(greenDonation.address, ETH.mul(1000));
    await stakingToken.mint(owner.address, ETH.mul(10));
    await stakingToken.approve(greenDonation.address, ETH);
    await treeContract.mint(1, ‘’, ‘’);
    await rewardToken.connect(whale).mint(greenDonation.address, ETH.mul(1000));
    await stakingToken.connect(whale).mint(whale.address, ETH.mul(10));
    await stakingToken.connect(whale).approve(greenDonation.address, ETH);
    await treeContract.connect(whale).mint(1, ‘’, ‘’);
    await treeContract.setGreenDonationContract(greenDonation.address);
    await expect(greenDonation.stake(1, ETH)).to.not.reverted;
    for(let I = 0; I < 10; I++) {
      await greenDonation.connect(whale).stake(2, ETH.div(10));
    }

    expect(await treeContract.balanceOf(owner.address)).to.eq(1);
    expect(await treeContract.ownerOf(1)).to.eq(owner.address);
    expect(await treeContract.ownerOf(2)).to.eq(whale.address);
    expect((await treeContract.trees(1)).level).to.eq(2);
    expect((await treeContract.trees(2)).level).to.eq(11);
    expect(await stakingToken.balanceOf(greenDonation.address)).to.eq(ETH.mul(2));
    expect(await stakingToken.balanceOf(owner.address)).to.eq(ETH.mul(9));
    expect(await stakingToken.balanceOf(whale.address)).to.eq(ETH.mul(9));
  });
```

Recommendation

- Consider having a minimum amount staked
- Consider having the tree level increase to be relative to the amount staked
- Consider having a maxLevel and time gap between each level increase

Status

Resolved

2. Claiming operation being allowed even when NFT is listed for sale allows for mischievous and unjust fraudulent activities through frontrunning

Path

NftMarketplace.sol#L142, GreenDonation.sol#L177, GreenDonation.sol#L256, RetirementCertificateEscrow.sol#L88

Function

listItem(), buyItem()

Description

Claiming operation being allowed even when NFT is listed for sale allows for mischievous and unjust fraudulent activities through frontrunning. These include but are not limited to:

- staked rewards claiming,
- certificate claiming

This made possible because users still have ownership of the NFT to be sold even after listing it up for sale in the listItem() function:

```
function listItem(address nftAddress, uint256 tokenId, uint256 price) external
    notListed(nftAddress, tokenId, msg.sender)
    isOwner(nftAddress, tokenId, msg.sender)
{
    if (price <= 0) {
        revert PriceMustBeAboveZero();
    }
    IERC721 nft = IERC721(nftAddress);
    if (nft.getApproved(tokenId) != address(this)) {
        revert NotApprovedForMarketplace();
    }
    s_listings[nftAddress][tokenId] = Listing(price, msg.sender);
    emit ItemListed(msg.sender, nftAddress, tokenId, price);
}
```

As you can see, it doesn't transfer the NFT to the contract and opens up this attack scenario:

- Bob's tree NFT has rewards and retirementCertificate that hasn't been claimed
- Bob promises to list his NFT for sale to Alice but at a high price since she stands to gain the rewards and retirementCertificate as well when she buys the NFT. But with the intent to trick Alice and defraud her.
- Alice fully convinced about this decides to buy Bob's NFT for the requested high price with the assurance that she gains both the staking rewards and retirementCertificate that hasn't been claimed
- Bob lists this NFT for sale, however it doesn't transfer the ownership of the NFT to the contract. This means Bob still has ownership of the NFT and can claim all rewards and certificates right before purchase.
- Trusting that the NFT has now been listed for sale, Alice sends the buyItem() transaction with the requested high price.
- Bob frontruns Alice's transaction and claims all the NFT's rewards and retirementCertificate Alice intended to gain from purchasing it.
- Alice's transaction then goes through leaving Alice with just the NFT which is now worthless and Bob still receives the fraudulent high price.

Recommendation

Consider transferring the ownership of the NFT to the contract upon listing to avoid this issue. If the NFT is unlisted, it can be transferred back to the owner.

Status

Resolved

3. Missing input validation introduces no restriction to which NFT can be traded on the NFTMarketplace [Wash-Trading]

Path

NftMarketplace.sol#L197-238

Function

listItem(), buyItem()

Description

The NFTMarketplace currently does not check which NFT is being listed for sale. As of now, a malicious user could create a worthless NFT collection and put it up for sale in the marketplace for anyone to purchase. These NFTs may not have any in-game functionality useful to the player's avatars. This attack vector also opens up the transfer of Tree NFTs which should be an evolving NFT that is linked to a single user.

POC

- Alice (attacker) lists her rugged 'pudgy-p' NFT on the marketplace for 100 CELO.
- Alice uses another account to buy the NFT and pays a variable redemptionFee.
- Alice gets the Toucan NFT minted to her address without viable interaction on the IntoTheVerse metaverse.

Recommendation

If there are multiple NFTs allowed to be listed for trading in the marketplace, include an allow-list mapping (address => bool) for admins to allow specific NFTs to be added to the marketplace.

Status

Resolved

4. Users can mint infinite amount of avatars due to missing mapping update

Path

Avatar.sol#L32, Avatar.sol#L55

Function

mint()

Description

The Avatar.sol contract allows for anyone to create their Avatar NFT and store information about the avatar. However, a core invariant of this contract is that this Avatar NFT should only be minted once by each person. This is done using _hasMintedAvatar, a mapping of the caller address to a boolean:

```
mapping(address => bool) public _hasMintedAvatar;
  ...
  function mint(string memory image) external {
    require(!_hasMintedAvatar[msg.sender], “You already have an avatar”); @audit
...
    avatars[tokenId] = AvatarInfo(tokenId, image, 0, false, msg.sender);
  }
```

As you can see, there is no where this mapping was set to true throughtout the mint operation. This means that _hasMintedAvatar[msg.sender] will always return false and the validation, require(!_hasMintedAvatar[msg.sender]) will ALWAYS pass. This vulnerability allows anyone to mint and own an infinite amount of avatars.

POC

```
it(“Should mint more than 1 avatar”, async function () {
    const { avatar, owner } = await loadFixture(deployAvatarFixture);
    for (let I = 0; I < 10; I++) {
      avatar.connect(owner).mint(‘infinite mint’);
    }
    // Users should not have access to NFT id 8
    await expect(avatar.connect(owner).changeAvatarImage(8, ‘infinite change’)).to.not.reverted;
  });
```

Recommendation

This line of code can be added to mitigate the risk:

```
  function mint(string memory image) external {
      require(!_hasMintedAvatar[msg.sender], “You already have an avatar”);
+     _hasMintedAvatar[msg.sender] = true;
      // Mint new Avatar NFT to the owner
      uint256 tokenId = totalSupply() + 1;
      _safeMint(msg.sender, tokenId);


      // Store information about the avatar
      avatars[tokenId] = AvatarInfo(tokenId, image, 0, false);
  }
```

Also note this is done before the _safeMint call to avoid leaving a reentrancy bug.

Status

Resolved

5. Users are forced to swap all reward tokens with no slippage protection

Path

GreenDonation.sol#L228, GreenDonation.sol#L242, GreenDonation.sol#L197

Function

_swapRewardTokenForTC02()

Description

In the GreenDonation.sol, whenever getReward() is called, the GreenDonation swaps about 10 percent of the user's reward for TC02. It forces users to swap this reward amount to TC02 but doesn't allow them to specify any slippage values.

```
function _swapRewardTokenForTC02(
    uint256 rewardsAmountToSwap
) internal returns (uint256) {
    address[] memory path = new address[](2);
    path[0] = address(rewardsToken);
    path[1] = address(tc02);
    uint256[] memory amountSwapped = swapRotuer.swapExactTokensForTokens(
        rewardsAmountToSwap,
        0, // TOOD: use proper method to fetch amount for TC02 to avoid slippage.
        path,
        address(this),
        block.timestamp
    );
    return amountSwapped[amountSwapped.length - 1];
}
```

From the above, all rewardsAmountToSwap are swapped and always use 0 for min out meaning that deposits will be sandwiched and stolen.

Recommendation

Allow users to specify slippage parameters for rewardsAmountToSwap when calling getReward()

Status

Resolved

6. When batch minting trees, only a single ID (_nextTokenId) receives the retirement certificate while other trees minted in the batch do not

Path

TreeContract.sol#L191

Function

mint()

Description

TreeContract.sol allows a user to mint up to 10 tokens in a batch but only registers certificates for claiming to the last tokenId minted because the _mint function completes the minting loop internally.

Following the narrative below:

- Alice mints tree NFT with ID 001 and passes in cost as the msg.value.
- At the end of the mint function, her retirementCertificateTokenId (e.g. 1) is approved and registered for claim.
- Alice calls retirementCertificateEscrow:claimRetirementCertificate(1, [0]) and claims her token successfully.
- Bob chooses to batch mint 9 trees (ID 002 - 010), and pays cost * 9.
- Bob expects tokenId 002 - 010 to all have retirementCertificateTokenIds and certificates registered for claim, but only retirementCertificateTokenId (2) would be minted and registered.
- Bob calls retirementCertificateEscrow:claimRetirementCertificate(2, [0]) and claims his first token successfully.
- Bob calls retirementCertificateEscrow:claimRetirementCertificate(3, [0]) expecting to claim the next retirement certificate accrued to treeId 003 but it reverts.

If there is no difference in the rarity/value of the retirement certificates for each transaction, i.e. a user paying 1 CELO and another batch minting with 10 CELO both have the same rewards (retirement certificates) there would be an unfair advantage to singly-minted trees, thereby putting batch minters at a disadvantage.

POC

```
it.only(‘ - Should mint retirementCertificate tokens to all tree token IDs’, async () => {
    const { treeContract, owner, whale, retirementCertificateEscrow, retirementCertificate}
= await loadFixture(fetchFixtures)
    await treeContract.mint(1, ‘’, ‘’)
    await treeContract.connect(whale).mint(3, ‘’, ‘’)
    expect(await treeContract.balanceOf(owner.address)).to.eq(1)
    expect(await treeContract.balanceOf(whale.address)).to.eq(3)
    const userRetirementCertificate = await
retirementCertificateEscrow.getUserRetirementCertificates(1);
    expect(userRetirementCertificate[0][0].claimed).eq(false)
    expect(userRetirementCertificate[0][0].retirementCertificate).eq(1)
    expect(userRetirementCertificate[0][0].tree).eq(1)
    expect(userRetirementCertificate[1]).eq(1)



    const userRetirementCertificate2 = await
retirementCertificateEscrow.getUserRetirementCertificates(2);
    expect(userRetirementCertificate2[0][0].claimed).eq(false)
    expect(userRetirementCertificate2[0][0].retirementCertificate).eq(2)
    expect(userRetirementCertificate2[0][0].tree).eq(2)
    expect(userRetirementCertificate2[1]).eq(1)



    const userRetirementCertificate3 = await
retirementCertificateEscrow.getUserRetirementCertificates(3);
    console.log(userRetirementCertificate3) // empty
    // expect(userRetirementCertificate3[0][0].tree).to.not.eq(3)



    await
retirementCertificateEscrow.setRetirementCertificate(retirementCertificate.address);
    await retirementCertificateEscrow.claimRetirementCertificate(1, [0]);
    await retirementCertificateEscrow.connect(whale).claimRetirementCertificate(2, [0]);
    await retirementCertificateEscrow.connect(whale).claimRetirementCertificate(3, [0]);
    await
expect(retirementCertificateEscrow.connect(whale).claimRetirementCertificate(3,
[0])).to.be.reverted;
  });
```

Recommendation

Individual trees should be minted with their respective retirement certificates. A retirementCertificate minting loop can be implemented using the quantity of trees minted.

Status

Resolved

Medium Severity Issues

7. Reward swaps can be manipulated by node operators due to using block.timestamp in setting deadline

Path

GreenDonation.sol#L200

Function

_swapRewardTokenForTC02()

Description

The _swapRewardTokenForTC02 function which swaps reward tokens for TC02 performs swaps using block.timestamp is setting the deadline.

```
 function _swapRewardTokenForTC02(
      uint256 rewardsAmountToSwap
  ) internal returns (uint256) {
      address[] memory path = new address[](2);
      path[0] = address(rewardsToken);
      path[1] = address(tc02);
      uint256[] memory amountSwapped = swapRotuer.swapExactTokensForTokens(
         rewardsAmountToSwap,
         0, // TOOD: use proper method to fetch amount for TC02 to avoid slippage.
         path,
         address(this),
         block.timestamp
      );
      return amountSwapped[amountSwapped.length - 1];
  }
```

Note, whenever it gets executed that will be the block.timestamp therefore this will always pass regardless of when it gets executed, allowing node operators to manipulate however the market shifts. In the PoS model, proposers know well in advance if they will propose one or consecutive blocks ahead of time. In such a scenario, a malicious validator can hold back the transaction and execute it at a block number more favorable to them.

Recommendation

Deadline should be taken in as an input parameter not set with block.timestamp

Status

Partially Resolved

8. Inadequate buyer protection allows sellers to be paid more than their listed NFT selling price

Path

NftMarketplace.sol#L204

Function

buyItem()

Description

When a user lists an item for sale on the marketplace, a definite selling price is attached which the buyer has to match to receive the NFT. The check for pricing is a relative-price check and not an exact-price check because of the comparison used below.

```
if (msg.value < listedItem.price) {
```

If the buyer passes in msg.value greater than listedItem.price, the transaction continues on the execution pathway because it passes the comparison used above. The seller's accrued proceeds should be msg.value - amountToSwap, an amount expected to be less than msg.value provided redemptionRate is greater than 0.

```
uint256 amountToSwap = (msg.value * redemptionRate) / 100;
s_proceeds[listedItem.seller] += (msg.value - amountToSwap);
```

POC

- Alice lists NFT for sale for 100 CELO
- Admin sets redemptionRate as 10%
- Bob buys NFT but passes in 2000 CELO as msg.value
- Retirement Certificate gets minted for 200 CELO with Bob as beneficiary
- Alice's proceeds will be 2000 - 200 CELO = 1800 CELO thereby breaking the invariant.

This puts the buyer at risk of significant financial loss while the seller would be compensated beyond their expectations.

Recommendation

The check for msg.value and listing price can be re-written as:

```
-   if (msg.value < listedItem.price) {
+   if (msg.value == listedItem.price)
```

The protocol accounting can be updated to allow refunds to buyers who buy items with a value greater than the listing price.

Status

Resolved

Low Severity Issues

9. setRedemptionRate() function exists in the documentation but is not implemented in the GreenDonation codebase

Path

GreenDonation.sol#213

Function

getReward()

Description

The documentation specifies that the rewards to be swapped for TC02 can vary based on the setRedemptionRate function. This function is not implemented in the codebase as expected and hardcodes 10% as the amount of any reward claimed to be converted to TC02. Currently any calls to getReward() will always charge a 10% fee on the claimed rewards to swap to TC02

Recommendation

Implement the function as specified in the docs or update the docs to match the code.

Status

Resolved

10. Minting from TreeContract and NFTMarketplace will have the same retiringEntityString.

Path

NftMarketplace.sol#L212

Function

buyItem()

Description

When minting certificates in buyItem(), the retiringEntityString used here is the same as used in the TreeContract. Since the string is tied to the certificates, when a user transacts on the marketplace (buys an NFT) they will be minted a retirement certificate with an erroneous parameter. It may need to be updated to show where the retirement is happening from (the NFT marketplace and not the Tree Contract).

Recommendation

Update “Into The Verse Tree User” to “Into The Verse Marketplace User”.

Status

Resolved

11. Privileged user (owner and rewardsDistribution) management

Path

Avatar.sol, GreenDonation.sol, TreeContract.sol, NftMarketplace.sol, MarketplaceRetirementCertificateEscrow.sol, RetirementCertificateEscrow.sol, RewardsDistributionRecipient.sol

Function

renounceOwnership( ), transferOwnership( ), notifyRewardAmount( )

Description

Privileged accounts within the contract are able to adjust core functionality. It is paramount that these accounts are properly managed via the use of a multi-sig wallet or DAO governance approval before changes are made. A 2 of 3 multi-signature wallet requires at least 2 signers to sign off (approve) a transaction before it gets executed. This reduces the likelihood of the protocol being completely grounded because of 1 private key being hacked.

Also if ownership transfer is not properly done, the current contract owner can renounce/transfer ownership, thereby losing owner privileges. When ownership is renounced, all of the contract's methods with the onlyOwner modifier will be rendered unusable.

Recommendation

- Implement a multi party agreement (using a multi-signature wallet or DAO) to prevent a single hacked owner from bricking the protocol.
- Functions such as renounceOwnership and transferOwnership can be overridden or set up for 2-step verification to prevent mistaken privilege transfer or renouncing.

References

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol

https://github.com/razzorsec/RazzorSec-Contracts/blob/main/AccessControl/SafeOwn.sol

Status

Resolved

12. Prior to watering and delay calculation, trees are set to level 0

Path

NftMarketplace.sol#L204

Function

buyItem()

Description

The calculation in _calculateDelay() allows tree levels to be set depending on how much time has passed between the last watering event and the current time (block.timestamp). If the tree's level is required anytime before waterTree() is called, the tree's level while unset is 0.
If waterTree() is called and it is still within the decayPeriod (i.e. while decayedLevels == 0), then the value is incremented to 2. This means the treeLevel would go from level 0 –> level 2.

```
// Update the lastWatered timestamp
      trees[_tokenId].lastWatered = block.timestamp;
      trees[_tokenId].level++; // @audit
```

POC

```
it.only(' - Should level tree up from level 0 to level 2 by one waterTree() call', async () => {
    const { greenDonation, whale, treeContract, owner, ETH, rewardToken, stakingToken}
= await loadFixture(fetchFixtures)

    await stakingToken.connect(whale).mint(whale.address, ETH.mul(10));
    await stakingToken.connect(whale).approve(greenDonation.address, ETH);
    await treeContract.connect(whale).mint(1, '', '');
    await stakingToken.mint(whale.address, ETH.mul(10));
    await treeContract.setGreenDonationContract(greenDonation.address);
    // Tree level 0 (undefined)
    expect((await treeContract.trees(1)).level).to.eq(0);
    await greenDonation.connect(whale).stake(1, ETH);
    // Tree is now level 2
    expect((await treeContract.trees(1)).level).to.eq(2);
  });
```

Recommendation

Introduce a minimum level when minting trees to avoid non-sequential tree level-up.

Status

Resolved

Informational Issues

13. Modifiers and require checks contain repeated code

Path

MarketplaceRetirementCertificate.sol#L94

Function

RetirementCertificateEscrow.sol:claimRetirementCertificate(),
MarketplaceRetirementCertificateEscrow.sol:claimRetirementCertificate()

Description

There are modifiers made to check that the caller is not a specific address, either the nftMarketplace or greenDonation contracts as well as require checks that perform the same function. As modifiers are inlined during function execution, this becomes a repetition that can be avoided.

```
modifier notNftMarketplace(address caller) {
    require(caller != nftMarketplace, “Caller NFTMarketplace”);
    _;
}

function claimRetirementCertificate(
    address nftAddress,
    uint256 tokenId,
    uint256[] memory userRetirementCertificatesIndexes
  )
    external nonReentrant onlyNFTOwner(nftAddress, tokenId, msg.sender)
    notNftMarketplace(msg.sender)  // @audit require check below
  {
    require(msg.sender != nftMarketplace, “Marketplace cannot claim”);
...
  }
```

```
modifier notGreenDonation(address caller) {
    require(caller != greenDonation, “Caller GreenDonation”);
    _;
}
```

```
 function claimRetirementCertificate(
    uint256 tree,
    uint256[] memory userRetirementCertificatesIndexes
 )
    external
    nonReentrant
    onlyTreeOwner(tree, msg.sender)
    notGreenDonation(msg.sender) // @audit require check below
 {
    require(msg.sender != greenDonation, “GreenDonation cannot claim”);
 ...
 }
```

Recommendation

Since the modifier is only used once, it can be removed to use only the require(...) check.

Status

Resolved

14. Changes to state can be marked by events

Path

TreeContract.sol

Function

setGreenDonationContract(), setRedemptionRate(), setSwapRouter(), setRetirementCertificateEscrow(), setCost(), withdraw(), upgradeTree(), downgradeTree()

Description

Some of the functions (listed above) can cause changes to state. It is advisable to emit events for these functions.

Recommendation

Emit events for critical changes to state.

Status

Resolved

Functional Tests Cases

Some of the tests performed are mentioned below:

- Users should be able to mint only one avatar
- Washtrading NFTs should not yield benefits to the user
- Batch minting should batch mint retirement certificates
- Users should not overpay for NFTs

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the IntoTheVerse codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the end, the IntotheVerse Team Resolved almost all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in IntoTheVerse smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of IntoTheVerse smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the IntoTheVerse to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
June, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com