TABLE OF CONTENTS

OVERVIEW

Project Summary

Project Name	Guest TOKEN
Logo	
Platform	AVAX
Language	Solidity
Code Link	https://snowtrace.io/address/0x892bb36c427b6e64ab5d1d155e7c8a0b1791b28b

File Summary

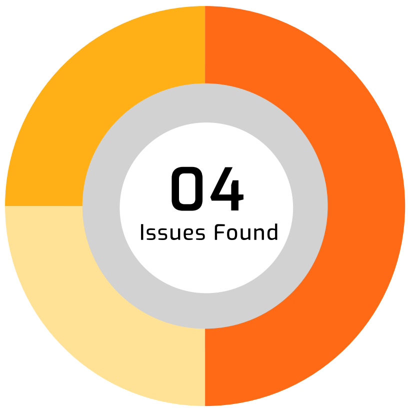
ID	File Name	Audit Status
GUEST	GUEST-Guest.sol	Failed

Audit Summary

Date of Delivery	14 Feb 2021
Audit Methodology	Code Analysis. Automatic Assesment, Manual Review
Audit Result	Failed ✗
Audit Team	BlockAudit Report Team

FINDINGS

🟥	Critical	0	0.0%
🟧	High	2	50.0%
🟨	Medium	0	0.0%
🟨	Low	2	50.0%
🟦	Informational	0	0.0%
🟪	Ownership	0	0.0%
🟩	Gas Optimization	0	0.0%



Vulnerability Findings Summary

ID	Type	Line	Severity	Status
GUEST01	Unchecked Transfer	84	🟧 High	Reported
GUEST02	Reentrancy	79-93	🟧 High	Reported
GUEST03	Divide Before Multiply	73	🟧 Medium	Reported
GUEST04	Missing Zero Address Validation	68 / 69 / 70 / 72	🟨 Low	Reported

GUEST01

Type	Unchecked Transfer
Severity	🟧 High
File	Guest.sol
Line	84
Status	**Reported**

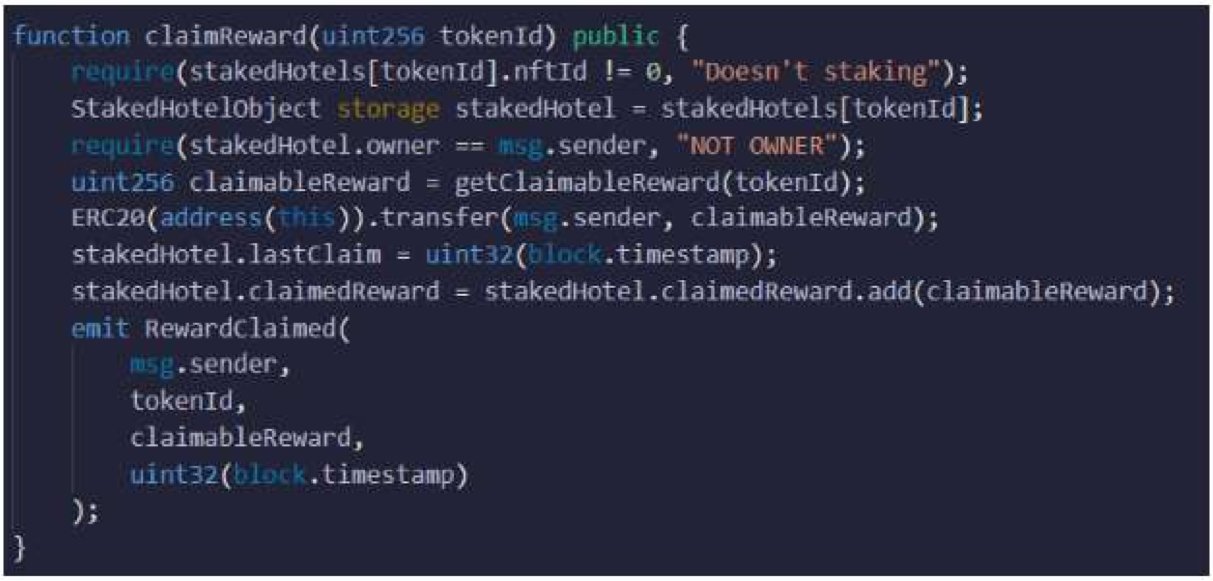
Description

The return value of an external transfer/transferFrom call is not checked, Several tokens do not revert in case of failure and return false. If one of these tokens is used in this contract, the function will not revert if the transfer fails, and an attacker can call it for free

Remediation

Use SafeERC20, or ensure that the transfer/transferFrom return value is checked.

Snapshot



```
function claimReward(uint256 tokenId) public {
    require(stakedHotels[tokenId].nftId != 0, "Doesn't staking");
    StakedHotelObject storage stakedHotel = stakedHotels[tokenId];
    require(stakedHotel.owner == msg.sender, "NOT OWNER");
    uint256 claimableReward = getClaimableReward(tokenId);
    ERC20(address(this)).transfer(msg.sender, claimableReward);
    stakedHotel.lastClaim = uint32(block.timestamp);
    stakedHotel.claimedReward = stakedHotel.claimedReward.add(claimableReward);
    emit RewardClaimed(
        msg.sender,
        tokenId,
        claimableReward,
        uint32(block.timestamp)
    );
}
```

GUEST02

Type	Reentrancy
Severity	🟧 High
File	Guest.sol
Line	79-93
Status	**Reported**

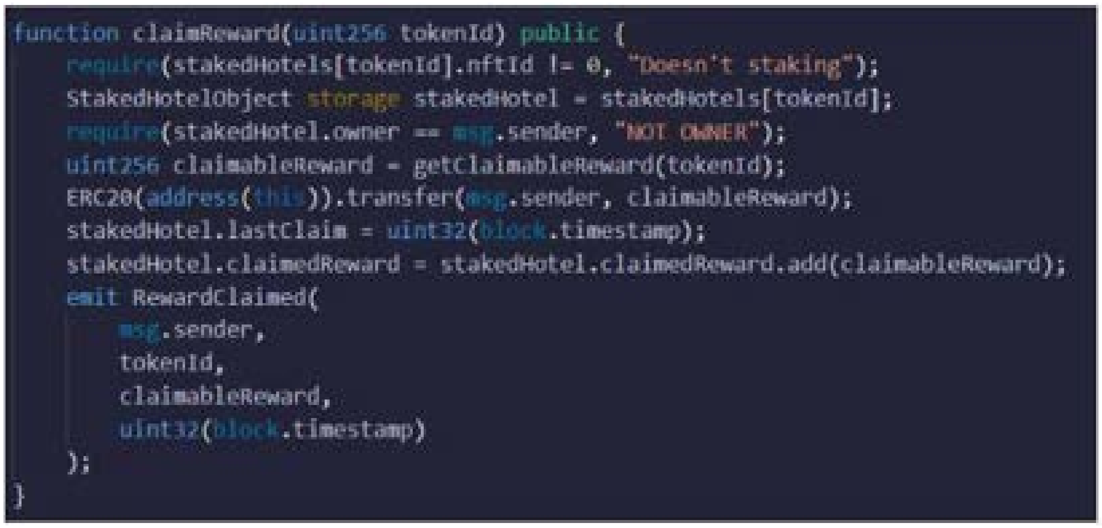
Description

claimReward(uint256) performs state variable changes after external calls, making it a potential target for a reentrancy attack if the inputs are malicious contracts.

Remediation

Update all bookkeeping state variables before transferring execution to an external contract or use Openzeppelin's ReentrancyGuard.

Snapshot



```
function claimReward(uint256 tokenId) public {
    require(stakedHotels[tokenId].nftId != 0, "Doesn't staking");
    StakedHotelObject storage stakedHotel = stakedHotels[tokenId];
    require(stakedHotel.owner == msg.sender, "NOT OWNER");
    uint256 claimableReward = getClaimableReward(tokenId);
    ERC20(address(this)).transfer(msg.sender, claimableReward);
    stakedHotel.lastClaim = uint32(block.timestamp);
    stakedHotel.claimedReward = stakedHotel.claimedReward.add(claimableReward);
    emit RewardClaimed(
        msg.sender,
        tokenId,
        claimableReward,
        uint32(block.timestamp)
    );
}
```

GUEST03

Type	Divide before multiply
Severity	🟧 Medium
File	Guest.sol
Line	73
Status	**Reported**

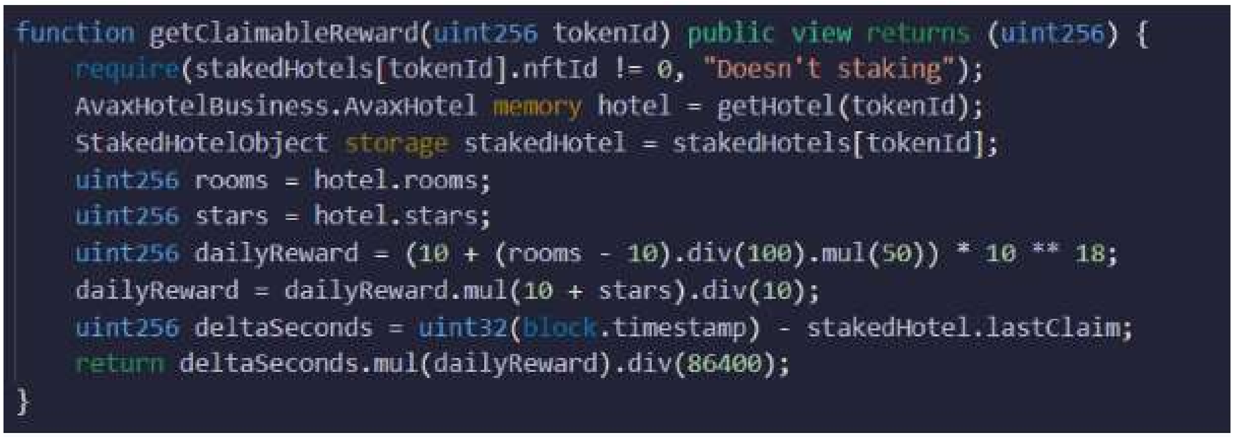
Description

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision.

Remediation

In general, it's usually a good idea to re-arrange arithmetic to perform multiplication before division, unless the limit of a smaller type makes this dangerous.

Snapshot



```
function getClaimableReward(uint256 tokenId) public view returns (uint256) {
    require(stakedHotels[tokenId].nftId != 0, "Doesn't staking");
    AvaxHotelBusiness.AvaxHotel memory hotel = getHotel(tokenId);
    StakedHotelObject storage stakedHotel = stakedHotels[tokenId];
    uint256 rooms = hotel.rooms;
    uint256 stars = hotel.stars;
    uint256 dailyReward = (10 + (rooms - 10).div(100).mul(50)) * 10 ** 18;
    dailyReward = dailyReward.mul(10 + stars).div(10);
    uint256 deltaSeconds = uint32(block.timestamp) - stakedHotel.lastClaim;
    return deltaSeconds.mul(dailyReward).div(86400);
}
```

GUEST04

Type	Missing zero address validation
Severity	🟨 Low
File	Guest.sol
Line	68 / 69 / 70 / 72
Status	**Reported**

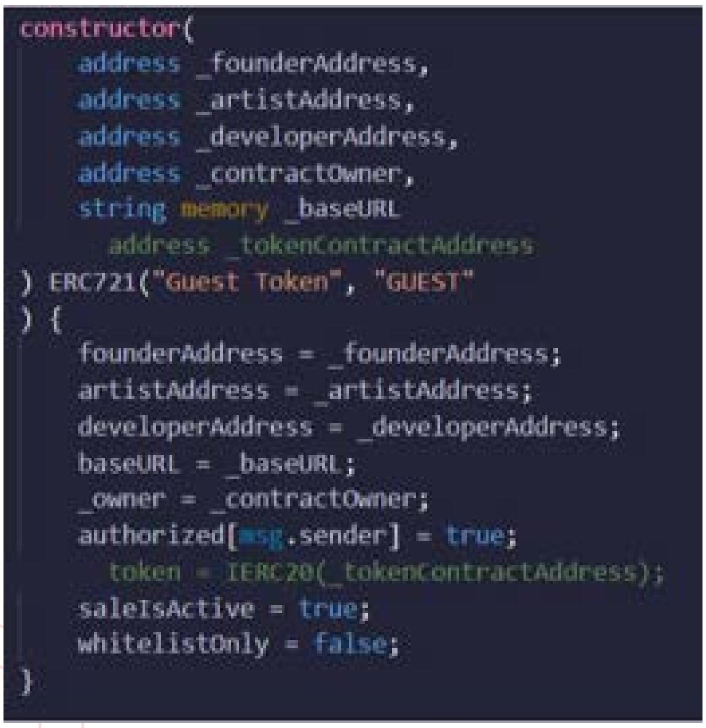
Description

constructor(address,address,address,address,string) does not check if inputs are zero addresses, So if Bob inputs zero addresses. funds can be lost.

Remediation

Check for zero address validation

Snapshot



```
constructor(
    address _founderAddress,
    address _artistAddress,
    address _developerAddress,
    address _contractOwner,
    string memory _baseURL
      address _tokenContractAddress
) ERC721("Guest Token", "GUEST"
) {
    founderAddress = _founderAddress;
    artistAddress = _artistAddress;
    developerAddress = _developerAddress;
    baseURL = _baseURL;
    _owner = _contractOwner;
    authorized[msg.sender] = true;
      token = IERC20(_tokenContractAddress);
    saleIsActive = true;
    whitelistOnly = false;
}
```

APPENDIX

Auditing Approach and Methodologies applied

The Block Audit Report team has performed rigorous testing of the project including the analysis of the code design patterns where we reviewed the smart contract architecture to ensure it is structured along with the safe use of standard inherited contracts and libraries. Our team also conducted a formal line by line inspection of the Smart Contract i.e., a manual review, to find potential issues including but not limited to

- Race conditions
- Zero race conditions approval attacks
- Re-entrancy
- Transaction-ordering dependence
- Timestamp dependence
- Check-effects-interaction pattern (optimistic accounting)
- Decentralized denial-of-service attacks
- Secure ether transfer pattern
- Guard check pattern
- Fail-safe mode
- Gas-limits and infinite loops
- Call Stack depth

In the Unit testing Phase, we coded/conducted custom unit tests written against each function in the contract to verify the claimed functionality from our client. In Automated Testing, we tested the Smart Contract with our standard set of multifunctional tools to identify vulnerabilities and security flaws. The code was tested in collaboration of our multiple team members and this included but not limited to;

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analyzing the complexity of the code in depth and in detail line-by-line manual review of the code.
- Deploying the code on testnet using multiple clients to run live tests.
- Analyzing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analyzing the security of the on-chain data.

Issue Categories:

Every issue in this report was assigned a severity level from the following:

🟥 Critical Severity Issues

Issues of Critical Severity leaves smart contracts vulnerable to major exploits and can lead to asset loss and data loss. These can have significant impact on the functionality/performance of the smart contract.
We recommend these issues must be fixed before proceeding to MainNet..

🟧 High Severity Issues

Issues of High Severity are not as easy to exploit but they might endanger the execution of the smart contract and potentially create crucial problems.
Fixing these issues is highly recommended before proceeding to MainNet.

🟨 Medium Severity Issues

Issues on this level are not a major cause of vulnerability to the smart contract, they cannot lead to data-manipulations or asset loss but may affect funtionality.
It is important to fix these issues before proceeding to MainNet.

🟨 Low Severity Issues

Issues at this level are very low in their impact on the overall functionality and execution of the smart contract. These are mostly code-level violations or improper formatting.
These issues can be remain unfixed or can be fixed at a later date if the code is redeployed or forked.

🟦 Informational Findings

These are finding that our team comes accross when manually reviewing a smart contract which are important to know for the owners as well as users of a contract.
These issues must be acknowledged by the owners before we publish our report.

🟪 Ownership Privileges

Owner of a smart contract can include certain rights and Privileges while deploying a smart contract that might be hidden deep inside the codebase and may make the project vulnerable to rug-pulls or other types of scams.
We at BlockAudit believe in transparency and hence we showcase Ownership Privileges separately so the owner as well as the investors can get a better understanding about the project.

🟩 Gas Optimization

Solidity gas optimization is the process of lowering the cost of operating your Solidity smart code. The term "gas" refers to the level of processing power required to perform specific tasks on the Ethereum network.
Each Ethereum transaction costs a fee since it requires the use of computer resources. It will deduct a fee anytime any function in the smart contract is invoked by the contract's owner or users.