TECHFUND Audit Score

Smart contract Audit Report

PHILAND Audit

Completed on 2022-10-21

Score **POSITIVE**

Risk level		
	Critical	0
	High	0
	Medium	0
	Low	6
	Note	6

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

Number of warnings

Compiler Version	1
Data Bounding / sanity checks	2
Expensive Loop	3
Re-Entrancy	6
State Variable Default Visibility	0
In-Correct Calculation Sequence	0
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Double Withdrawal	0

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

Solidity version used is inconsistent & not fixed. It is highly recommended to update the Solidity version to the latest. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements and using a fixed version of Solidity upon understanding the version specifications.

```
// SPDX-License-Identifier: GPL-2.0-or-later

//                ___    ___
//              /\__\  /\__\
//      ________/ /   /_ \/__/
//     /\_______\/   /__\__\
//     / /       /       /   /
//    / /   /   /   /   /   /
//   / /   /___/___/___/___/
//  / /   /
//  \/___/

pragma solidity ^0.8.7;
import { IQuestObject } from "./interfaces/IQuestObject.sol";
import { AccessControlUpgradeable } from "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
```

Data Bounding / sanity checks

2

Data unity check is not properly done for “setShopAddress" function. For all address inputs it is highly recommended to check at least “zero check” to make sure that address is not equal to zero.

contracts/utils/BaseObject.sol
contracts/PhiShop.sol

```
138     function setShopAddress(address _shopAddress) external onlyOwner {
139         shopAddress = _shopAddress;
140         emit SetShopAddress(shopAddress);
141     }
```

```
46      constructor(
47          address _freeObjectAddress,
48          address _premiumObjectAddress,
49          address _wallPaperAddress,
50          address _basePlateAddress,
51          address _mapAddress
52      ) {
53          freeObjectAddress = _freeObjectAddress;
54          premiumObjectAddress = _premiumObjectAddress;
55          wallPaperAddress = _wallPaperAddress;
56          basePlateAddress = _basePlateAddress;
57          mapAddress = _mapAddress;
58      }
```

Expensive Loop

3

Although we didn't find any major issue because of it, but it is suggested **NEVER** to have potentially costly operations in a loop. It is always suggested to utilise "from - to" pattern or limit loop to a "approximate predictable cost" checks. But since there is no critical issue with current implementation it is shown as "Note" level issue.

contracts/object/WallPaper.sol
contracts/object/BasePlate.sol
contracts/object/PremiumObject.sol

```
154     function batchWallPaper(uint256[] memory tokenIds) external payable nonReentrant {
155         uint256 allprice;
156         // check if the function caller is not an zero account address
157         require(msg.sender != address(0), "msg sender(0) is invalid");
158         // to prevent bots minting from a contract
159         require(msg.sender == tx.origin, "msg sender invalid");
160 
161         uint256 objectLength = tokenIds.length;
162         for (uint256 i = 0; i < objectLength; ++i) {
163             allprice = allprice + allObjects[tokenIds[i]].price;
164         }
165         // price sent in to buy should be equal to or more than the token's price
166         require(msg.value >= allprice, "should be equal to or more than the token's price");
167 
168         for (uint256 i = 0; i < objectLength; ++i) {
169             _buyWallPaper(tokenIds[i]);
170         }
171     }
```

```
84  function _buyWallPaper(address to, uint256 tokenId) internal {
85      // check the token id exists
86      isValid(tokenId);
87      // check token is open for sale
88      require(allObjects[tokenId].forSale, "not open for sale");
89      // check token's MaxClaimed
90      require(super.totalSupply(tokenId) <= allObjects[tokenId].maxClaimed, "reach maxClaimed");
91 
92      // Pay royality to artist, and remaining to sales address
93      (bool calcSuccess1, uint256 res) = SafeMath.tryMul(allObjects[tokenId].price, royalityFee);
94      require(calcSuccess1, "calc error");
95      (bool calcSuccess2, uint256 royality) = SafeMath.tryDiv(res, 10000);
96      require(calcSuccess2, "calc error");
97      (bool success1, ) = payable(allObjects[tokenId].creator).call{ value: royality }("");
98      require(success1, "cant pay royality");
99      (bool success2, ) = payable(treasuryAddress).call{ value: (allObjects[tokenId].price - royality) }("");
00      require(success2, "cant transfer sales");
01 
02      // mint the token
03      super._mint(to, tokenId, 1, "0x");
04      emit LogBuyWallPaper(to, tokenId, allObjects[tokenId].price);
05  }
```

External

```
function batchBasePlate(uint256[] memory tokenIds) external payable nonReentrant {
    uint256 allprice;
    // check if the function caller is not an zero account address
    require(msg.sender != address(0), "msg sender(0) is invalid");
```

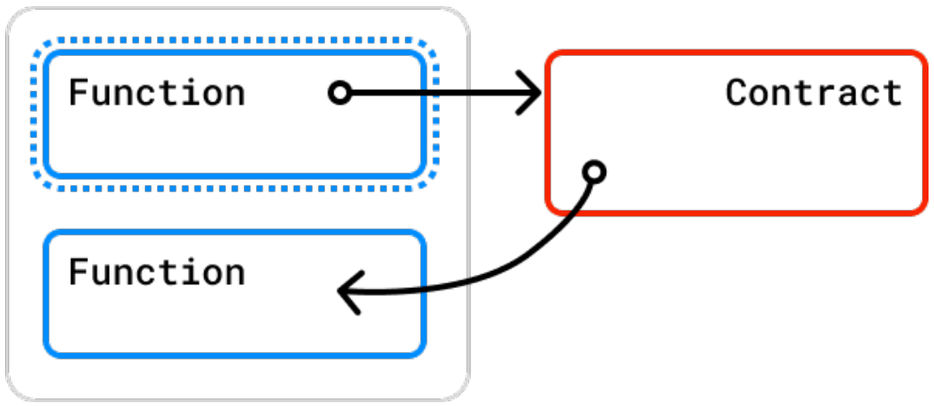
```
function batchBuyObject(uint256[] memory tokenIds) external payable nonReentrant {
    uint256 allprice;
    // check if the function caller is not an zero account address
    require(msg.sender != address(0), "msg sender invalid");
```

Re-Entry 6

A reentrancy attack can occur when you create a function that makes an external call to another untrusted contract before it resolves any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

There are multiple places where event emits after an external call. However, which means we can get control at a point when event has not been fired. Which opens up the possibility of re-entry. Please follow “Check => Effect => Interaction” pattern to prevent such issues.

object/BasePlate.sol
object/FreeObject.sol
object/PremiumObject.sol
object/QuestObject.sol
object/WallPaper.sol
PhiShop.sol




```
function _buyBasePlate(uint256 tokenId) internal {
    // check the token id exists
    isValid(tokenId);
    // check token is open for sale
    require(allObjects[tokenId].forSale, "not open forSale");
    // check token's MaxClaimed
    require(super.totalSupply(tokenId) <= allObjects[tokenId].maxClaimed, "reach maxClaimed");

    // Pay royality to artist, and remaining to sales address
    (bool calcSuccess1, uint256 res) = SafeMath.tryMul(allObjects[tokenId].price, royalityFee);
    require(calcSuccess1, "calc error");
    (bool calcSuccess2, uint256 royality) = SafeMath.tryDiv(res, 10000);
    require(calcSuccess2, "calc error");
    (bool success1, ) = payable(allObjects[tokenId].creator).call{ value: royality }("");
    require(success1, "cant pay royality");
    (bool success2, ) = payable(treasuryAddress).call{ value: (allObjects[tokenId].price - royality) }("");
    require(success2, "cant transfer sale");

    // mint the token
    super._mint(msg.sender, tokenId, 1, "0x");
    emit LogBuyBasePlate(msg.sender, tokenId, allObjects[tokenId].price);
}
```

TECHFUND Audit Score

Smart contract Audit Report

Re-Entry

6

```
function _getFreeObject(uint256 tokenId) internal {
    // check token is open for sale
    require(allObjects[tokenId].forSale, "not open for sale");
    // check the token id exists
    isValid(tokenId);
    // mint the token
    super._mint(msg.sender, tokenId, 1, "0x00");
    emit LogGetObject(msg.sender, tokenId);
}
```

```
function _buyObject(uint256 tokenId) internal {
    // check the token id exists
    isValid(tokenId);
    // check token is open for sale
    require(allObjects[tokenId].forSale, "not open for sale");
    // check token's MaxClaimed
    require(super.totalSupply(tokenId) <= allObjects[tokenId].maxClaimed, "reach maxClaimed");

    // Pay royality to artist, and remaining to sales address
    (bool calcSuccess1, uint256 res) = SafeMath.tryMul(allObjects[tokenId].price, royalityFee);
    require(calcSuccess1, "calc error");
    (bool calcSuccess2, uint256 royality) = SafeMath.tryDiv(res, 10000);
    require(calcSuccess2, "calc error");
    (bool success1, ) = payable(allObjects[tokenId].creator).call{ value: royality }("");
    require(success1, "cant pay royality");
    (bool success2, ) = payable(treasuryAddress).call{ value: (allObjects[tokenId].price - royality) }("");
    require(success2, "cant transfer sales");

    // mint the token
    super._mint(msg.sender, tokenId, 1, "0x");
    emit LogBuyObject(msg.sender, tokenId, allObjects[tokenId].price);
}
```

```
function getObject(address to, uint256 tokenId) external onlyOwner {
    // check if the function caller is not an zero account address
    require(to != address(0), "to(0) is invalid");
    // check token is open for sale
    require(allObjects[tokenId].forSale, "not open forSale");
    // check the token id exists
    isValid(tokenId);
    // check token's MaxClaimed
    require(super.totalSupply(tokenId) <= allObjects[tokenId].maxClaimed, "reach maxClaimed"
    // mint the token
    super._mint(to, tokenId, 1, "0x00");
    emit LogGetQuestObject(msg.sender, tokenId);
}
```

```
function _buyWallPaper(uint256 tokenId) internal {
    // check the token id exists
    isValid(tokenId);
    // check token is open for sale
    require(allObjects[tokenId].forSale, "not open forSale");
    // check token's MaxClaimed
    require(super.totalSupply(tokenId) <= allObjects[tokenId].maxClaimed, "reach maxClaimed");

    // Pay royality to artist, and remaining to sales address
    (bool calcSuccess1, uint256 res) = SafeMath.tryMul(allObjects[tokenId].price, royalityFee);
    require(calcSuccess1, "calc error");
    (bool calcSuccess2, uint256 royality) = SafeMath.tryDiv(res, 10000);
    require(calcSuccess2, "calc error");
    (bool success1, ) = payable(allObjects[tokenId].creator).call{ value: royality }("");
    require(success1, "cant pay royality");
    (bool success2, ) = payable(treasuryAddress).call{ value: (allObjects[tokenId].price - royality) }("");
    require(success2, "cant transfer sales");

    // mint the token
    super._mint(msg.sender, tokenId, 1, "0x");
    emit LogBuyWallPaper(msg.sender, tokenId, allObjects[tokenId].price);
}
```

```
function shopBuyAndDepositObject(
    string memory name,
    uint256[] memory ftokenIds,
    uint256[] memory ptokenIds,
    uint256[] memory wtokenIds,
    [redacted]

    emit LogShopBuyObject(
        msg.sender,
        msg.sender,
        ftokenIds.length + ftokenIds.length + wtokenIds.length + btokenIds.length,
        msg.value
    );

    [redacted]
```