MythX

REPORT 6265816EAAE5F20018766476

Created	Sun Apr 24 2022 16:57:18 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	6265706d5ec4940334c82dc0

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
f3de54cc-1918-427e-9c57-c24ecc70c34d	Marketplace_flat.sol	18

Analysis f3de54cc-1918-427e-9c57-c24ecc70c34d MythX

Started	Sun Apr 24 2022 16:57:23 GMT+0000 (Coordinated Universal Time)
Finished	Sun Apr 24 2022 17:12:50 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Remythx
Main Source File	Marketplace_flat.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	1	17

ISSUES

MEDIUM

Multiple calls are executed in the same transaction.

SWC-113

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file

Marketplace_flat.sol

Locations

```
1864  item.sold = true;
1865  // transfer nft to buyer
1866  item.nft.transferFrom(address(this), msg.sender, item.tokenId);
1867  // emit Bought event
1868  emit Bought(
```

LOW

A floating pragma is set.

SWC-103

The current pragma Solidity directive is ""^0.4.22<0.9.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

Marketplace_flat.sol

Locations

```
5
6
7   pragma solidity ^0.4.22 <0.9.0;
8
9   library console {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

Marketplace_flat.sol

Locations

```
1542 // OpenZeppelin Contracts v4.4.1 (security/ReentrancyGuard.sol)
1543
1544 pragma solidity ^0.8.0;
1545
1546 /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

Marketplace_flat.sol

Locations

```
1608 // OpenZeppelin Contracts v4.4.1 (utils/introspection/IERC165.sol)
1609
1610 pragma solidity ^0.8.0;
1611
1612 /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

Marketplace_flat.sol

Locations

```
1636 // OpenZeppelin Contracts v4.4.1 (token/ERC721/IERC721.sol)
1637
1638 pragma solidity ^0.8.0;
1639
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.4"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

Marketplace_flat.sol

Locations

```
1779
1780
1781 pragma solidity ^0.8.4;
1782
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1871 item.tokenId,
1872 item.price,
1873 item.seller,
1874 msg.sender
1875 );
```

LOW

SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Source file

Marketplace_flat.sol

Locations

```
1831 itemCount ++;
1832 // transfer nft
1833 _nft.transferFrom(msg.sender, address(this), _tokenId); //moves the nft to the smartcontract
1834 // add new item to items mapping
1835 items[itemCount] = Item (
```

LOW

SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Source file

Marketplace_flat.sol

Locations

```
1864 item.sold = true;
1865 // transfer nft to buyer
1866 item.nft.transferFrom(address(this), msg.sender, item.tokenId);
1867 // emit Bought event
1868 emit Bought(
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1833 _nft.transferFrom(msg.sender, address(this), _tokenId); //moves the nft to the smartcontract
1834 // add new item to items mapping
1835 items[itemCount] = Item (
1836 itemCount,
1837 _nft,
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1868 emit Bought(
1869 _itemId,
1870 address(item.nft),
1871 item.tokenId,
1872 item.price,
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1870 address(item.nft),
1871 item.tokenId,
1872 item.price,
1873 item.seller,
1874 msg.sender
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1599 // By storing the original value once again, a refund is triggered (see
1600 // https://eips.ethereum.org/EIPS/eip-2200)
1601 _status = _NOT_ENTERED;
1602 }
1603 }
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1833 _nft.transferFrom(msg.sender, address(this), _tokenId); //moves the nft to the smartcontract
1834 // add new item to items mapping
1835 items[itemCount] = Item (
1836 itemCount,
1837 _nft,
1838 _tokenId,
1839 _price,
1840 payable(msg.sender),
1841 false
1842 );
1843 // emit Offered event
1844 emit Offered(
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1843 // emit Offered event
1844 emit Offered(
1845 itemCount,
1846 address(_nft), //The nft address is fetched by casting it in the address operator
1847 _tokenId,
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1869 _itemId,
1870 address(item.nft),
1871 item.tokenId,
1872 item.price,
1873 item.seller,
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

Marketplace_flat.sol

Locations

```
1834 // add new item to items mapping
1835 items[itemCount] = Item (
1836 itemCount,
1837 _nft,
1838 _tokenId,
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

Marketplace_flat.sol

Locations

```
1831 itemCount ++;
1832 // transfer nft
1833 _nft.transferFrom(msg.sender, address(this), _tokenId); //moves the nft to the smartcontract
1834 // add new item to items mapping
1835 items[itemCount] = Item (
```

Source file

Marketplace_flat.sol

Locations

```
1784
1785
1786 contract Marketplace is ReentrancyGuard {
1787
1788 // Variables
1789 //address payable public immutable feeAccount; // the account that receives fees from sales, immutable means they can be assigned a value once
1790 //uint public immutable feePercent; // the fee percentage on sales
1791 uint public itemCount;
1792
1793 struct Item {
1794 uint itemId;
1795 IERC721 nft;
1796 uint tokenId;
1797 uint price;
1798 address payable seller;
1799 bool sold;
1800 }
1801
1802 // itemId -> Item
1803 mapping(uint => Item) public items;
1804
1805 event Offered(
1806 uint itemId,
1807 address indexed nft,
1808 uint tokenId,
1809 uint price,
1810 address indexed seller
1811 );
1812 event Bought(
1813 uint itemId,
1814 address indexed nft,
1815 uint tokenId,
1816 uint price,
1817 address indexed seller,
1818 address indexed buyer
1819 );
1820
1821 //constructor(uint _feePercent) {
1822 constructor() {
1823 //feeAccount = payable(msg.sender);
1824 //feePercent = _feePercent;
1825 }
1826
1827 // Make item to offer on the marketplace
1828 function makeItem(IERC721 _nft, uint _tokenId, uint _price) external nonReentrant { //IERC721 _nft takes the address of the nft and make it an nft instance, nonReentrant is from
```

```
the imported reentrancyguard
require(_price > 0, "Price must be greater than zero");
// increment itemCount
itemCount ++;
// transfer nft
_nft.transferFrom(msg.sender, address(this), _tokenId); //moves the nft to the smartcontract
// add new item to items mapping
items[itemCount] = Item (
itemCount,
_nft,
_tokenId,
_price,
payable(msg.sender),
false
);
// emit Offered event
emit Offered(
itemCount,
address(_nft), //The nft address is fetched by casting it in the address operator
_tokenId,
_price,
msg.sender
);
}



function purchaseItem(uint _itemId) external payable nonReentrant { //external to prevent accessing it from within the smart contract
uint _totalPrice = getTotalPrice(_itemId);
Item storage item = items[_itemId]; //Storage is used to declare that it is reading directly from the mapping (not creating memory copy)
require(_itemId > 0 && _itemId <= itemCount, "item doesn't exist");
require(msg.value >= _totalPrice, "not enough ether to cover item price and market fee");
require(!item.sold, "item already sold");
// pay seller and feeAccount
item.seller.transfer(item.price);
//feeAccount.transfer(_totalPrice - item.price);
// update item to sold
item.sold = true;
// transfer nft to buyer
item.nft.transferFrom(address(this), msg.sender, item.tokenId);
// emit Bought event
emit Bought(
_itemId,
address(item.nft),
item.tokenId,
item.price,
item.seller,
msg.sender
);
}
function getTotalPrice(uint _itemId) view public returns(uint){ //view because it only views variables without modifying them
//return((items[_itemId].price*(100 + feePercent))/100);
return((items[_itemId].price));
}
}
```