// NFT MARKET PLACE CONTRACT

Smart Contracts CTF audit report

Prepared by: Michael Maximov

Date of Engagement:
2022, May 21st - June 5th

CONTACTS

Michael Maximov

maksymovmm@gmail.com

OVERVIEW

1.1 INTRO

I have decided to participate in your challenge on May 21st after an awesome talk with Thomas Richard.

I have been interested in crypto space since 2017. However, I never dug deep into it, so I decided to go ahead and give it a chance.

Thank you guys for this awesome opportunity, I enjoyed participating in your CTF a lot.

1.2 AUDIT SUMMARY

I have dedicated a week to familiarize myself with all the basics i need to dive into audit:

1. I have read the “Mastering Ethereum” book by Andreas Antonoupoulos and dr.Gavin Wood
2. Familiarized myself with the basics of solidity using cryptozombies.io
3. Made a “learning speed run” using “Ethereum, solidity, brownie course” by Patrick Collins on Youtube
4. Started “playing” with coding and deploying my own contracts. At the same time I have explored some of the basic smart contracts security topics. Mostly following

along with the “console cowboys course”

Then, during next week, I have started exploring Halborn CTF contracts:

1. For 6 days exploring these contracts, I identified and managed to exploit 6 vulnerabilities, outlined in the next sections
2. Had a lot of fun ;)

1.3 APPROACH

I have followed a fully manual approach while auditing smart contracts:

1. Firstly, I took a quick look at the architecture and the layout.
2. Then I googled literally everything that seemed unfamiliar to me. (At the end of this stage, I already had a full understanding what this contract is for)
3. When I identified purposes which all of the functions intended for, I have created a list of common pitfalls which may occur in implementing these functionalities.
4. In the next step I have created a list of possible attack vectors regarding given functionalities. (At the end of this phase, I already had few leads to investigate further)
5. Then I started a line-by-line code review. I have written what every particular function is supposed to do in plain english.
6. I have prepared a VScode brownie environment for testing.
7. Also I have used Remix for transactions debugging frequently.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur.

This framework works for communicating the characteristics and impacts of technology vulnerabilities.

The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores.

For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.

4 - High probability of an incident occurring.

3 - Potential of a security incident in the long term.

2 - Low probability of an incident occurring.

1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.

4 - May cause a significant level of impact or loss.

3 - May cause a partial impact or loss to many.

2 - May cause temporary impact or loss.

1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating

a value of 10 to 1 with 10 being the highest level of security risk.

10 - **CRITICAL**

9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The Engagement was scoped to the following contract:

NFTMarketPlace.sol

2. FINDINGS AND TECH DETAILS

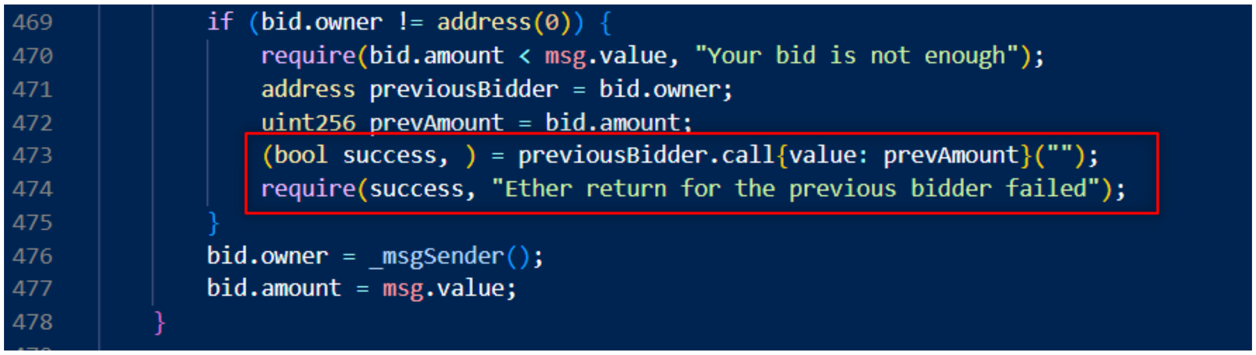
2.1 BID FUNCTION INTRODUCES DENIAL OF SERVICE OPPORTUNITY

The 'bid' function is vulnerable to unexpected revert attack.

This vulnerability introduces 'Denial of Service' under next circumstances:

- Attacker places a bid using his own contract (intentionally configured to revert upon Ether receive)

- When other participant tries to place a bid, NFTMarketPlace's logic tries to send Ether back to previous bidder:



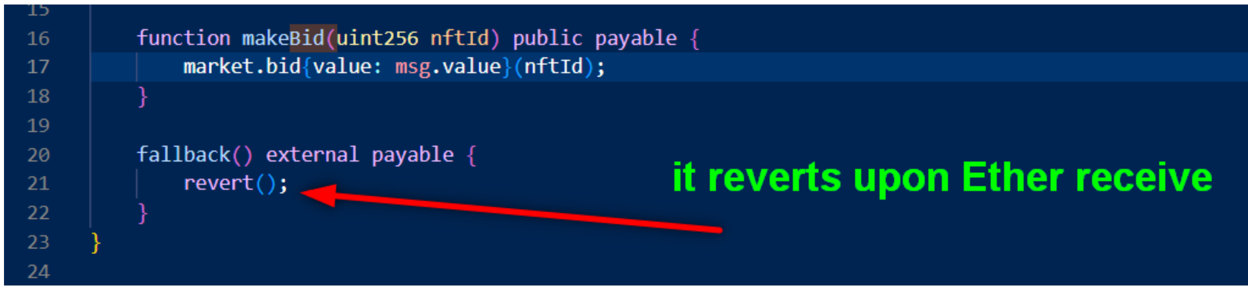
```
        if (bid.owner != address(0)) {
            require(bid.amount < msg.value, "Your bid is not enough");
            address previousBidder = bid.owner;
            uint256 prevAmount = bid.amount;
            (bool success, ) = previousBidder.call{value: prevAmount}("");
            require(success, "Ether return for the previous bidder failed");
        }
        bid.owner = _msgSender();
        bid.amount = msg.value;
    }
```

- This ‘require’ statement does not allow auction to continue until the Ether is sent back to the previous bidder.

- If the previous bidder is a contract configured not to receive it ever, refund is impossible, so the auction is stuck forever.

Proof of Concept:

- deploy a contract from the attacker’s address with the following function:



```
    function makeBid(uint256 nftId) public payable {
        market.bid{value: msg.value}(nftId);
    }

    fallback() external payable {
        revert();
    }
}
```

- place a bid for some of the NFTs from attacker’s contract
- try to place a bid from another account
- observe, it does not accept a bid

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1. Open halborn.7z archive
2. You’ll find **NFT/scripts/deploy_NFT_market.py**
3. Run this script. It will deploy a NFTMarketPlace.sol contract
4. Find **NFT/scripts/poc_bid.py**
5. Run this script. Firstly, it will deploy the attack.sol (you can find it at **NFT/contracts/attack.sol**) contract and also prepare the required state of the NFTMarketPlace.sol for successful attack.

   (###I have included multiple comment lines to help you understand the flow)

Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

The whole ‘Ether return’ logic must be rewritten.

Consider using ‘pull’ not ‘push’ pattern in this case.

you can refer to this article:

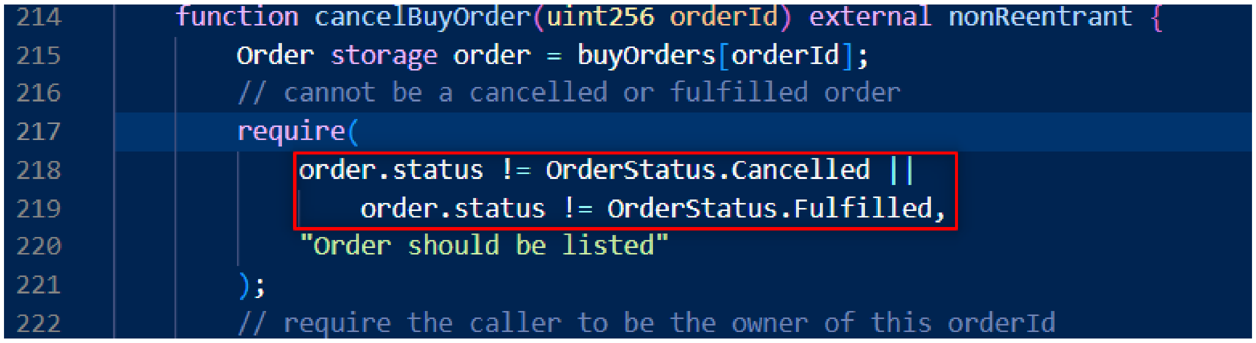
https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/#favor-pull-over-push-for-external-calls

In other words, simply:

1. create a mapping which will store approved returns
2. lock user’s ether while they are a previous bidder
3. when the role of previous bidder goes to the next participant, unblock user’s tokens and store approved amount in this mapping
4. make users pull their tokens back based on the ownership and amount in this mapping

2.2 CANCELBUYORDER FUNCTION ALLOWS AN ORDER TO BE CANCELED EVEN IF THE STATUS IS FULFILLED

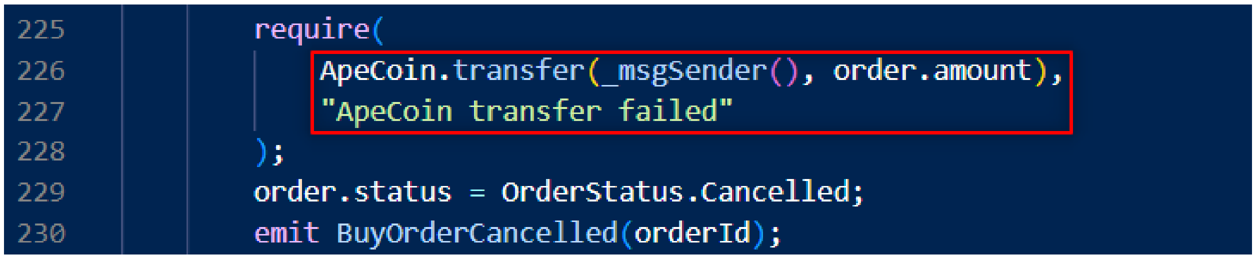
This is happening due to the faulty status check. It is supposed to require order status to be Listed:



```
    function cancelBuyOrder(uint256 orderId) external nonReentrant {
        Order storage order = buyOrders[orderId];
        // cannot be a cancelled or fulfilled order
        require(
            order.status != OrderStatus.Cancelled ||
                order.status != OrderStatus.Fulfilled,
            "Order should be listed"
        );
        // require the caller to be the owner of this orderId
```

It actually allows the order to be either fulfilled or canceled.

Next line of code returns the token collateral back to participant in case of cancelation:



```
        require(
            ApeCoin.transfer(_msgSender(), order.amount),
            "ApeCoin transfer failed"
        );
        order.status = OrderStatus.Cancelled;
        emit BuyOrderCancelled(orderId);
```

These things combined allows an attacker to 'return' token amount even after successfully buying NFT (Fulfilled order status)

Proof of Concept:

- post 2 separate buy Orders. One from attacker’s account, and one from victim’s (We need the contract to have a balance in ApeCoins equal to the amount of attacker’s Buy Order)
- use nft_owner’s account to sell the NFT to the attacker. Use sellToOrderId(0)
- use the attacker's account to cancel their Buy Order and receive a ‘return’.

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1. Open halborn.7z archive
2. You’ll find **NFT/scripts/deploy_NFT_market.py**
3. Run this script. It will deploy a HalbornToken.sol contract
4. Find **NFT/scripts/poc_cancel.py**
5. Run this script. It will prepare the required state for you and execute the whole attack.

(###I have included multiple comment lines to help you understand the flow)

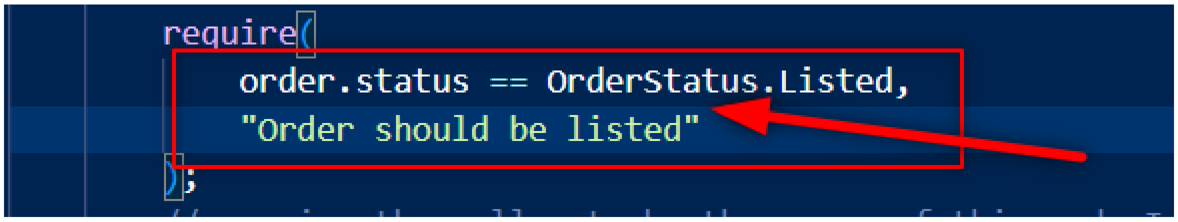
Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

Rewrite the cancelBuyOrder function so it correctly checks the status:

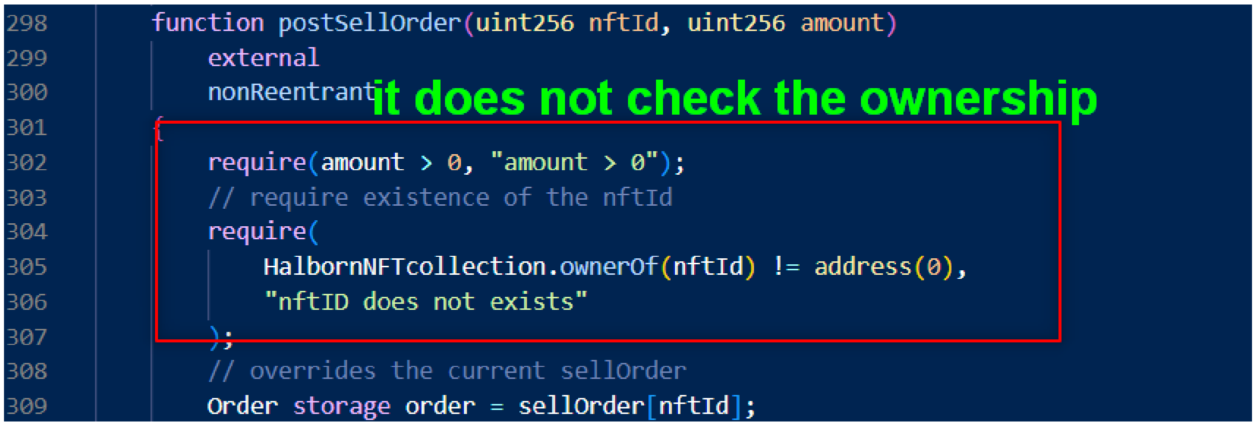


```
require(
    order.status == OrderStatus.Listed,
    "Order should be listed"
);
```

2.3 POSTSELLORDER FUNCTION ALLOWS ANYONE EDIT THE SELL ORDER AMOUNT AND BUY THE NFT FOR 1 WEI

postSellOrder function does not check the ownership.

Also it allows you to edit the current order.



```
    function postSellOrder(uint256 nftId, uint256 amount)
        external
        nonReentrant
    {
        require(amount > 0, "amount > 0");
        // require existence of the nftId
        require(
            HalbornNFTcollection.ownerOf(nftId) != address(0),
            "nftID does not exists"
        );
        // overrides the current sellOrder
        Order storage order = sellOrder[nftId];
```

This means anyone can edit the sell order (including amount) and buy the given NFT for 1 Wei

Proof of Concept:

- post the sell order (for one specific NFT, say '101') from the nft_owner account. Set the price amount to 1000000000000000001

- post the sell order (for the same nft_id) from the attacker's account. Set the price amount to 1.

(This basically edits the current nft_owner's sell Order)

- buy this 'edited' sell order from the attacker's account

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1.Open halborn.7z archive

2. You’ll find **NFT/scripts/deploy_NFT_market.py**

3. Run this script. It will deploy a NFTMarketPlace.sol contract

4. Find **NFT/scripts/poc_post.py**

5. Run this script. It will prepare the required state for you and execute the attack.

(###I have included multiple comment lines to help you understand the flow)

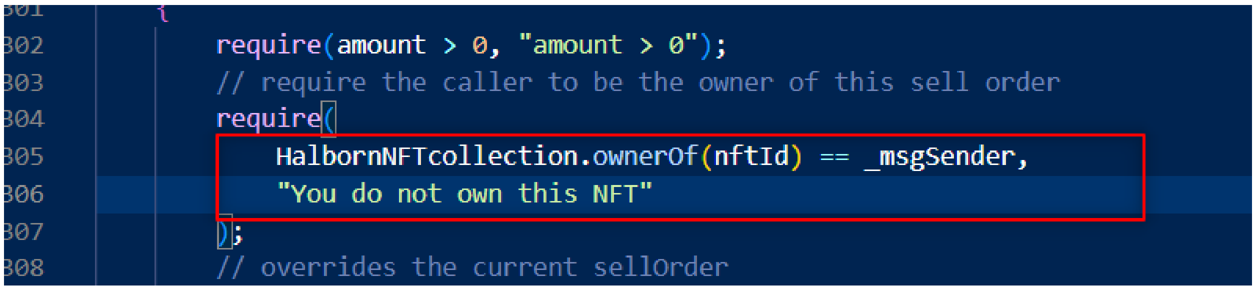
Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

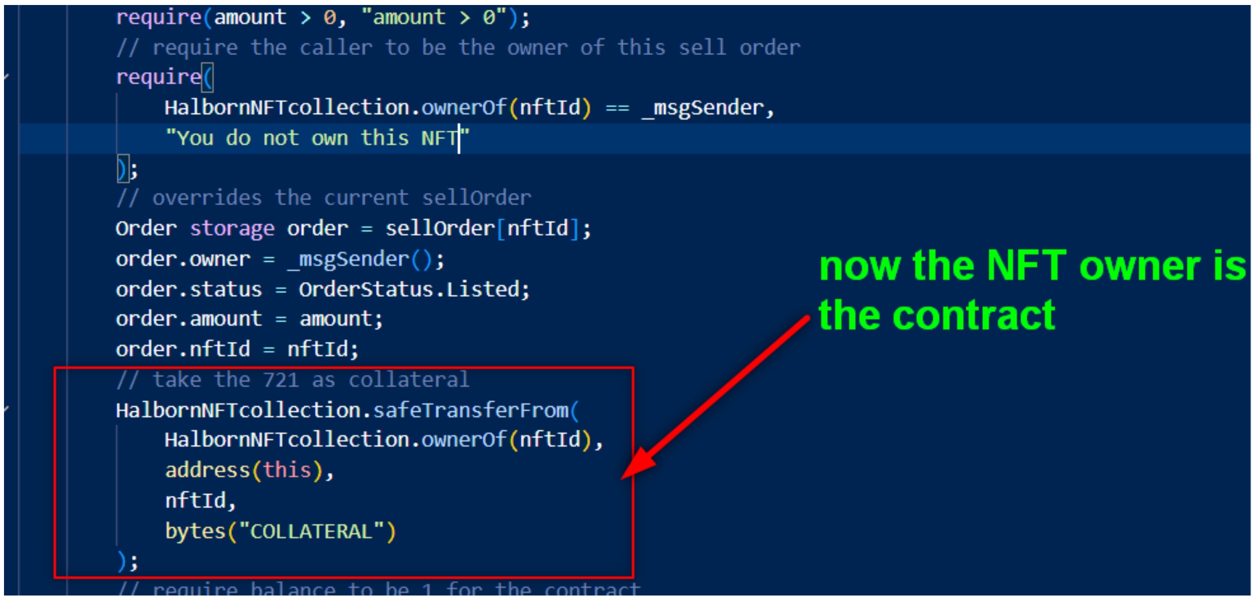
include the ownership check as a require statement, like this:



```
{
    require(amount > 0, "amount > 0");
    // require the caller to be the owner of this sell order
    require(
        HalbornNFTcollection.ownerOf(nftId) == _msgSender,
        "You do not own this NFT"
    );
    // overrides the current sellOrder
```

Also, note that user's won't be able to edit or modify existing Sell order. Now, if the user decide to change something, they simply cancel this sell order and post a new one.

This is because of that line:



```
require(amount > 0, "amount > 0");
// require the caller to be the owner of this sell order
require(
    HalbornNFTcollection.ownerOf(nftId) == _msgSender,
    "You do not own this NFT"
);
// overrides the current sellOrder
Order storage order = sellOrder[nftId];
order.owner = _msgSender();
order.status = OrderStatus.Listed;
order.amount = amount;
order.nftId = nftId;
// take the 721 as collateral
HalbornNFTcollection.safeTransferFrom(
    HalbornNFTcollection.ownerOf(nftId),
    address(this),
    nftId,
    bytes("COLLATERAL")
);
// require balance to be 1 for the contract
```

So, just make sure, your users know about that.