LooksRare

Security Assessment

March 29, 2022

Prepared for:

Zodd

LooksRare

Guts

LooksRare

Prepared by: **Nat Chin, Jaime Iglesias, Anish Naik, and Robert Schneider**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

LooksRare engaged Trail of Bits to review the security of its exchange and token staking contracts. From February 28 to March 11, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in the compromise of a smart contract, a loss of funds, or unexpected behavior. We conducted this audit with full knowledge of the target system, including access to the source code and limited documentation. We performed static analysis and a manual review of the project's Solidity components.

Summary of Findings

The audit did not uncover significant flaws that could result in the compromise of a smart contract, a loss of funds, or unexpected behavior. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Low	5
Informational	7
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Authentication	1
Auditing and Logging	1
Configuration	1
Data Validation	5
Patching	4
Undefined Behavior	2

Notable Findings

Flaws that could impact the state of funds held in a contract or cause unexpected behavior are listed below.

- **Risk of reuse of signatures across forks (TOB-LR-1)**
  Due to the static generation of the `DOMAIN_SEPARATOR` in the contract constructor, the system may be vulnerable to signature replay attacks on forked chains. This is an issue inherent in EIP-712 implementations.

- **Risks associated with contract ownership transfer process (TOB-LR-2)**
  Contract ownership is transferred in a single step, which is risky. Implementing a two-step process for transferring contract ownership would allow ownership transfers to be reversed if an incorrect address is input. This is an issue inherent in OpenZeppelin contracts.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 24, 2022	Pre-project kickoff call
March 7, 2022	Status update meeting #1
March 11, 2022	Report readout meeting
March 11, 2022	Delivery of report draft
March 29, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the LooksRare exchange and token staking contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Are there denial-of-service attack vectors?
- Could users lose access to their funds?
- Does the system validate and limit fee amounts?
- Does the system validate data from external contracts?

Project Targets

The engagement involved a review and testing of the following target.

LooksRare Contracts

Repository	https://github.com/LooksRare/looksrare-contracts
Version	`072d96bc32081ff7aa5971973b351ecb1720b3cd`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

LooksRare Exchange

This non-upgradeable contract is the core smart contract of the LooksRare protocol. It defines the logic for users to match maker and taker orders, allowing them to buy and sell NFTs. It is supported by a series of modules, called manager contracts, which define the supported currencies, types of orders (strategies), royalty fees used in the system, and transfer functionality to support multiple NFT standards.

Manager Contracts (`CurrencyManager`, `ExecutionManager`, `RoyaltyFeeManager`)

These contracts allow LooksRare admins to add features to and remove features from the protocol without relying on upgradeable contracts. For example, the `CurrencyManager` contract is used to add and remove support for different currencies, and the `ExecutionManager` contract is used to add and remove different execution strategies, allowing the protocol to support multiple order types. We focused on reviewing the risks related to the various state changes, and we evaluated owner privileges throughout these contracts.

Execution Strategies

These standalone contracts define the way an order is executed. For example, a maker may want to sell a specific NFT or any item in a collection at a given price. Each contract implements a common interface with the main functions `canExecuteTakerAsk` and `canExecuteTakerBid` and determines whether a maker order can be executed. We evaluated the level of privilege that these strategies have on trades and used Echidna to test the Dutch auction strategy calculations.

Exchange Libraries

There are two libraries in the LooksRare ecosystem: `OrderTypes.sol` and `SignatureChecker.sol`. While reviewing the `OrderTypes.sol` library, we verified that the `MakerOrder` struct is hashed according to the EIP-712 specification. Additionally, we ensured that the `MakerOrder` and `TakerOrder` structs have the necessary parameters to perform correct order matching. While reviewing `SignatureChecker.sol`, we checked that the system correctly verifies and recovers EIP-712-signed `MakerOrders`.

Exchange—Royalty Fee Helper Contracts

These smart contracts are used to store and retrieve royalty payment information for NFT collections. We focused on checking whether the owner has excessive privileges and whether the `royaltyFeeLimit` threshold can be violated.

Exchange—Transfer Manager Contracts

The transfer manager contracts enable the LooksRare exchange to flexibly manage the transfers of ERC721, ERC1155, and non-compliant ERC721 tokens. We focused on reviewing the contracts' compliance with the ERC721 and ERC1155 standards and their resilience to calling non-compliant ERC721 tokens.

Token Staking—`TokenDistributor`

This contract allows users to stake LOOKS tokens to earn compounded interest over a series of reward periods. We focused on the reward and share calculations and ensured that the contract is resilient against flash loan attacks.

Token Staking—`FeeSharingSystem`

This contract provides another entry point to interact with the `TokenDistributor` contract and allows users to gain WETH rewards. We focused on verifying the correctness of the reward and share calculations.

Token Staking—`LooksRareToken`

This is an ERC20 token with a supply cap that is set on construction. We focused on identifying ways in which the code could encounter unexpected behavior due to its returning of a boolean instead of reverting when minting LOOKS tokens.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- **Token staking:** The token staking smart contracts allow users to stake LOOKS tokens, receive rewards (in LOOKS tokens and fees), participate in the LOOKS token airdrop, engage in private sales for LOOKS tokens, and participate in token vesting. During this audit, we were unable to review the following contracts:
  - `FeeSharingSetter.sol`
  - `OperatorControllerForRewards.sol`
  - `StakingPoolForUniswapV2Tokens.sol`
  - `TokenSplitter.sol`
  - `TradingRewardsDistributor.sol`
  - `VestingContractWithFeeSharing.sol`
  - `PrivateSaleWithFeeSharing.sol`
  - `LooksRareAirdrop.sol`

- **Off-chain orderbook:** The off-chain orderbook stores maker order signatures. This component was considered out of scope for our review. Our assumptions surrounding the off-chain orderbook are highlighted in appendix C.

Automated Testing Results

Trail of Bits has developed unique tools for testing smart contracts. In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations; for example, Echidna may not randomly generate an edge case that violates a property.

We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 10 million test cases per property.

Our automated testing and verification focused on the following system properties:

`StrategyDutchAuction.sol`. We used Echidna to check that the result of the Dutch auction price calculation falls between `endPrice` and `startPrice`. If this invariant does not hold, the `currentAuctionPrice` is faulty and may lead to additional price manipulation. To test this price variation, we also used Echidna to identify the *maximum* deviation from the bounds provided. Additionally, since the `currentAuctionPrice` calculation requires the multiplication of potentially large values, we also tested whether that operation could lead to an integer overflow.

ID	Property	Tool	Result
1	The calculated `currentAuctionPrice` is always inclusively bound between `endPrice` and `startPrice`.	Echidna	**PASSED**
2	The numerator for `currentAuctionPrice` cannot overflow.	Echidna	**FAILED**
3	Calculation of the maximum deviation that `currentAuctionPrice` takes *if* it exits the inclusive bounds of `endPrice` and `startPrice`	Echidna	**0**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Solidity v0.8.0 arithmetic operations are used in the system and are tested through unit tests. Most arithmetic and parameter-tuning operations for the NFT exchange contracts are documented. However, automated fuzz testing has not been integrated. See appendix F for our recommendations on system invariants that can be tested.	**Moderate**
Auditing	Many critical state-changing operations emit events; however, one event is missing (TOB-LR-8). The LooksRare team provided additional details on the off-chain computations and the way they are performed. However, the team did not provide an incident response plan. See appendix G for our recommendations on documenting and maintaining an incident response plan.	**Moderate**
Authentication / Access Controls	The level of access provided to privileged users is limited and is controlled by multisignature (multisig) wallet(s) (appendix E). The principle of least privilege is mostly followed (TOB-LR-5), and users can enter and exit the system at will. Users can also use nonces to cancel orders on a smart contract. The process of transferring contract ownership can be improved by implementing a two-step process (TOB-LR-2).	**Satisfactory**
Complexity Management	The system has a large number of peripheral managers and helpers that increase its overall complexity. Most functions have clear purposes, and the system has a modular architecture. We did not identify instances of excessive branching. We recommend implementing our code quality recommendations (appendix H) to reduce the	**Moderate**

	system's complexity.	
Cryptography and Key Management	EIP-712 struct hashing and signature verification is performed correctly. Additionally, the use of the EIP-712 standard decreases the risk of phishing attacks against users. See the "Summary of Recommendations" section for additional recommendations to mitigate phishing risks. Hot wallet key management was not evaluated as part of this audit.	**Satisfactory**
Decentralization	Privileged users can specify the tokens, strategies, and transfer managers that will be used for system execution. Additionally, the off-chain system acts as a single point of failure. Compromise of the off-chain system can lead to denial-of-service conditions.	**Weak**
Documentation	We were provided with sufficient documentation to reason through the protocol's various process flows and data structures. We recommend developing additional documentation regarding the off-chain computations and token staking contracts.	**Satisfactory**
Front-Running Resistance	The protocol's smart contracts are not upgradeable; therefore, there is no risk of initialization front-running. However, because takers submit makers' signatures to the blockchain, there is a risk that taker orders could be front-run.	**Moderate**
Low-Level Calls	The system uses low-level calls minimally with the necessary safeguards and interacts with external contracts primarily through allowlists. We recommend using `try-catch` statements carefully while performing conditional low-level calls (appendix H).	**Satisfactory**
Testing and Verification	The test suite has 97% branch coverage, and Slither is used for static analysis. The test suite would benefit from automated verification such as Echidna fuzzing tests and	**Moderate**

	formal verification to test system properties.	

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Risk of reuse of signatures across forks due to lack of chainID validation	Authentication	High
2	Lack of two-step process for contract ownership changes	Data Validation	High
3	Project dependencies contain vulnerabilities	Patching	Medium
4	Users that create ask orders cannot modify minPercentageToAsk	Data Validation	Low
5	Excessive privileges of RoyaltyFeeSetter and RoyaltyFeeRegistry owners	Access Controls	Low
6	Insufficient protection of sensitive information	Configuration	Low
7	Contracts used as dependencies do not track upstream changes	Patching	Low
8	Missing event for a critical operation	Auditing and Logging	Low
9	Taker orders are not EIP-712 signatures	Data Validation	Informational
10	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
11	isContract may behave unexpectedly	Undefined Behavior	Informational

12	tokenId and amount fully controlled by the order strategy when matching two orders	Data Validation	**Informational**
13	Risk of phishing due to data stored in maker order params field	Data Validation	**Informational**
14	Use of legacy openssl version in solidity-coverage plugin	Patching	**Informational**
15	TypeScript compiler errors during deployment	Patching	**Informational**

Detailed Findings

1. Risk of reuse of signatures across forks due to lack of chainID validation

Severity: **High**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-LR-1
Target: `contracts/LooksRareExchange.sol`	

Description

At construction, the `LooksRareExchange` contract computes the domain separator using the network's `chainID`, which is fixed at the time of deployment. In the event of a post-deployment chain fork, the `chainID` cannot be updated, and the signatures may be replayed across both versions of the chain.

```
constructor(
    address _currencyManager,
    address _executionManager,
    address _royaltyFeeManager,
    address _WETH,
    address _protocolFeeRecipient
) {
    // Calculate the domain separator
    DOMAIN_SEPARATOR = keccak256(
        abi.encode(
            0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f, //
keccak256("EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)")
            0xda9101ba92939daf4bb2e18cd5f942363b9297fbc3232c9dd964abb1fb70ed71, //
keccak256("LooksRareExchange")
            0xc89efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bc6, //
keccak256(bytes("1")) for versionId = 1
            block.chainid,
            address(this)
        )
    );

    currencyManager = ICurrencyManager(_currencyManager);
    executionManager = IExecutionManager(_executionManager);
    royaltyFeeManager = IRoyaltyFeeManager(_royaltyFeeManager);
    WETH = _WETH;
    protocolFeeRecipient = _protocolFeeRecipient;
```

Figure 1.1: contracts/contracts/LooksRareExchange.sol#L137-L145

The `_validateOrder` function in the `LooksRareExchange` contract uses a `SignatureChecker` function, `verify`, to check the validity of a signature:

```
// Verify the validity of the signature
require(
    SignatureChecker.verify(
        orderHash,
        makerOrder.signer,
        makerOrder.v,
        makerOrder.r,
        makerOrder.s,
        DOMAIN_SEPARATOR
    ),
    "Signature: Invalid"
);
```

Figure 1.2: contracts/contracts/LooksRareExchange.sol#L576-L587

However, the `verify` function checks only that a user has signed the `domainSeparator`. As a result, in the event of a hard fork, an attacker could reuse signatures to receive user funds on both chains. To mitigate this risk, if a change in the `chainID` is detected, the domain separator can be cached and regenerated. Alternatively, instead of regenerating the entire domain separator, the `chainID` can be included in the schema of the signature passed to the order hash.

```
/**
 * @notice Returns whether the signer matches the signed message
 * @param hash the hash containing the signed mesage
 * @param signer the signer address to confirm message validity
 * @param v parameter (27 or 28)
 * @param r parameter
 * @param s parameter
 * @param domainSeparator paramer to prevent signature being executed in other chains and
environments
 * @return true --> if valid // false --> if invalid
 */
function verify(
    bytes32 hash,
    address signer,
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32 domainSeparator
) internal view returns (bool) {
    // \x19\x01 is the standardized encoding prefix
```

```
    // https://eips.ethereum.org/EIPS/eip-712#specification
    bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, hash));
    if (Address.isContract(signer)) {
        // 0x1626ba7e is the interfaceId for signature contracts (see IERC1271)
        return IERC1271(signer).isValidSignature(digest, abi.encodePacked(r, s, v)) ==
0x1626ba7e;
    } else {
        return recover(digest, v, r, s) == signer;
    }
}
```

Figure 1.3: contracts/contracts/libraries/SignatureChecker.sol#L41-L68

The signature schema does not account for the contract's chain. If a fork of Ethereum is made after the contract's creation, every signature will be usable in both forks.

Exploit Scenario

Bob signs a maker order on the Ethereum mainnet. He signs the domain separator with a signature to sell an NFT. Later, Ethereum is hard-forked and retains the same chain ID. As a result, there are two parallel chains with the same chain ID, and Eve can use Bob's signature to match orders on the forked chain.

Recommendations

Short term, to prevent post-deployment forks from affecting signatures, add the chain ID opcode to the signature schema.

Long term, identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.

2. Lack of two-step process for contract ownership changes

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LR-2
Target: LooksRare protocol	

Description

The owner of a LooksRare protocol contract can be changed by calling the `transferOwnership` function in OpenZeppelin's `Ownable` contract. This function internally calls the `_transferOwnership` function, which immediately sets the contract's new owner. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions anymore. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby removing any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
```

```
}
```

Figure 2.1: OpenZeppelin's `Ownable` contract

Exploit Scenario

Alice and Bob invoke the `transferOwnership()` function on the LooksRare multisig wallet to change the address of an existing contract's owner. They accidentally enter the wrong address, and ownership of the contract is transferred to the incorrect address. As a result, access to the contract is permanently revoked.

Recommendations

Short term, implement a two-step process to transfer contract ownership, in which the owner proposes a new address and then the new address executes a call to accept the role, completing the transfer.

Long term, identify and document all possible actions that can be taken by privileged accounts (appendix E) and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

3. Project dependencies contain vulnerabilities

Severity: **Medium**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-LR-3
Target: LooksRare protocol	

Description

Although dependency scans did not identify a direct threat to the project under review, `npm` and `yarn audit` identified a dependency with a known vulnerability. Due to the sensitivity of the deployment code and its environment, it is important to ensure that dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details the high severity issue:

CVE ID	Description	Dependency
CVE-2021-23358	Vulnerability to arbitrary code injection in `underscore`	`underscore`

Figure 3.1: An advisory affecting `underscore`, a `LooksRare` dependency

Exploit Scenario

Alice installs the dependencies of an in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure that LooksRare protocol dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, integrate automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure the code does not use and is not affected by the vulnerable functionality of the dependency.

4. Users that create ask orders cannot modify minPercentageToAsk

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LR-4
Target: `contracts/LooksRareExchange.sol`	

Description

Users who sell their NFTs on LooksRare are unable to protect their orders against arbitrary changes in royalty fees set by NFT collection owners; as a result, users may receive less of a sale's value than expected.

Ideally, when a user lists an NFT, he should be able to set a threshold at which the transaction will execute based on the amount of the sale's value that he will receive. This threshold is set via the `minPercentageToAsk` variable in the `MakerOrder` and `TakerOrder` structs. The `minPercentageToAsk` variable protects users who create ask orders from excessive royalty fees. When funds from an order are transferred, the `LooksRareExchange` contract ensures that the percentage amount that needs to be transferred to the recipient is greater than or equal to `minPercentageToAsk` (figure 3.1).

```
function _transferFeesAndFunds(
    address strategy,
    address collection,
    uint256 tokenId,
    address currency,
    address from,
    address to,
    uint256 amount,
    uint256 minPercentageToAsk
) internal {
    // Initialize the final amount that is transferred to seller
    uint256 finalSellerAmount = amount;

    // 1. Protocol fee
    {
        uint256 protocolFeeAmount = _calculateProtocolFee(strategy, amount);
        [...]
            finalSellerAmount -= protocolFeeAmount;
        }
    }
```

```
    // 2. Royalty fee
    {
        (
            address royaltyFeeRecipient,
            uint256 royaltyFeeAmount
        ) = royaltyFeeManager.calculateRoyaltyFeeAndGetRecipient(
                collection,
                tokenId,
                amount
            );
        // Check if there is a royalty fee and that it is different to 0
        [...]
            finalSellerAmount -= royaltyFeeAmount;
        [...]

    require(
        (finalSellerAmount * 10000) >= (minPercentageToAsk * amount),
        "Fees: Higher than expected"
    );
    [...]
}
```

Figure 4.1: The `_transferFeesAndFunds` function in `LooksRareExchange:422-466`

However, users creating ask orders cannot modify `minPercentageToAsk`. By default, the `minPercentageToAsk` of orders placed through the LooksRare platform is set to 85%. In cases in which there is no royalty fee and the protocol fee is 2%, `minPercentageToAsk` could be set to 98%.

Exploit Scenario

Alice lists an NFT for sale on LooksRare. The protocol fee is 2%, `minPercentageToAsk` is 85%, and there is no royalty fee. The NFT project grows in popularity, which motivates Eve, the owner of the NFT collection, to raise the royalty fee to 9.5%, the maximum fee allowed by the `RoyaltyFeeRegistry` contract. Bob purchases Alice's NFT. Alice receives 89.5% of the sale even though she could have received 98% of the sale at the time of the listing.

Recommendations

Short term, set `minPercentageToAsk` to 100% minus the sum of the protocol fee and the max value for a royalty fee, which is 9.5%.

Long term, identify and validate the bounds for all parameters and variables in the smart contract system.

5. Excessive privileges of RoyaltyFeeSetter and RoyaltyFeeRegistry owners

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-LR-5
Target: `contracts/royaltyFeeHelpers/RoyaltyFeeSetter.sol`, `contracts/royaltyFeeHelpers/RoyaltyFeeRegistry.sol`	

Description

The `RoyaltyFeeSetter` and `RoyaltyFeeRegistry` contract owners can manipulate an NFT collection's royalty information, such as the fee percentage and the fee receiver; this violates the principle of least privilege.

NFT collection owners can use the `RoyaltyFeeSetter` contract to set the royalty information for their NFT collections. This information is stored in the `RoyaltyFeeRegistry` contract.

However, the owners of the two contracts can also update this information (figures 5.1 and 5.2).

```
function updateRoyaltyInfoForCollection(
    address collection,
    address setter,
    address receiver,
    uint256 fee
) external override onlyOwner {
    require(fee <= royaltyFeeLimit, "Registry: Royalty fee too high");
    _royaltyFeeInfoCollection[collection] = FeeInfo({
        setter: setter,
        receiver: receiver,
        fee: fee
    });

    emit RoyaltyFeeUpdate(collection, setter, receiver, fee);
}
```

Figure 5.1: The `updateRoyaltyInfoForCollection` function in `RoyaltyFeeRegistry:54-64`

```
 function updateRoyaltyInfoForCollection(
    address collection,
    address setter,
    address receiver,
    uint256 fee
) external onlyOwner {
    IRoyaltyFeeRegistry(royaltyFeeRegistry).updateRoyaltyInfoForCollection(
        collection,
        setter,
        receiver,
        fee
    );
}
```

Figure 5.2: The `updateRoyaltyInfoForCollection` function in `RoyaltyFeeSetter:102-109`

This violates the principle of least privilege. Since it is the responsibility of the NFT collection's owner to set the royalty information, it is unnecessary for contract owners to have the same ability.

Exploit Scenario

Alice, the owner of the `RoyaltyFeeSetter` contract, sets the incorrect `receiver` address when updating the royalty information for Bob's NFT collection. Bob is now unable to receive fees from his NFT collection's secondary sales.

Recommendations

Short term, remove the ability for users to update an NFT collection's royalty information.

Long term, clearly document the responsibilities and levels of access provided to privileged users of the system.

6. Insufficient protection of sensitive information

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LR-6
Target: `contracts/hardhat.config.ts`	

Description

Sensitive information, such as API keys, is stored in process environments. This method of storage could make it easier for an attacker to compromise the information; compromise of the Alchemy key, for example, could enable an attacker to use the key and use up all the compute units, leading to a denial-of-service attack and the failure of other services reliant on the node.

The following portion of the `contracts/hardhat.config.ts` file exposes an Alchemy and Etherscan API key stored in the process environment:

```
/**
  rinkeby: {
    url: `https://eth-rinkeby.alchemyapi.io/v2/${process.env.ALCHEMY_RINKEBY_PRIVATE_KEY}`,
    accounts: [process.env.RINKEBY_KEY],
  },
  */
  /**
   mainnet: {
     url: `https://eth-mainnet.alchemyapi.io/v2/${process.env.ALCHEMY_MAINNET_PRIVATE_KEY}`,
     accounts: [process.env.MAINNET_KEY],
   },
   */
```

```
},

etherscan: {

  apiKey: process.env.ETHERSCAN_KEY,

},
```

Figure 6.1: The Alchemy and Etherscan API keys from the process environment in the `hardhat.config.ts` file

Exploit Scenario

Alice has the owner's mainnet Alchemy key stored in her process environment. Eve, an attacker, gains access to Alice's device and extracts the key. Eve uses the Alchemy key, using up all available compute units.

Recommendations

Short term, use a hardware security module to ensure that none of the keys can be extracted.

Long term, take the following steps:

- Move key material from the process environment to a dedicated secret management system with trusted computing abilities. The best options for such a system are the Google Cloud Key Management System and Hashicorp Vault (with hardware security module backing).
- Determine the business risk that would result from a lost or stolen key and develop disaster recovery and business continuity policies to be implemented in such a case.

7. Contracts used as dependencies do not track upstream changes

Severity: **Low**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-LR-7
Target: `contracts/libraries/SignatureChecker.sol`	

Description
The LooksRare codebase uses a third-party contract, `SignatureChecker`, but the LooksRare documentation does not specify which version of the contract is used or whether it was modified. This indicates that the LooksRare protocol does not track upstream changes in contracts used as dependencies. Therefore, the LooksRare contracts may not reliably reflect updates or security fixes implemented in their dependencies, as those updates must be manually integrated into the contracts.

Exploit Scenario
A third-party contract used in LooksRare receives an update with a critical fix for a vulnerability, but the update is not manually integrated in the LooksRare version of the contract. An attacker detects the use of a vulnerable contract in the LooksRare protocol and exploits the vulnerability against one of the contracts.

Recommendations
Short term, review the codebase and document the source and version of each dependency. Include third-party sources as submodules in the project's Git repository to maintain internal path consistency and ensure that dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages in the project.

8. Missing event for a critical operation

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-LR-8
Target: `contracts/LooksRareExchange.sol`	

Description

The system does not emit an event when a protocol fee is levied in the `_transferFeesAndFunds` and `_transferFeesAndFundsWithWETH` functions. Operations that transfer value or perform critical operations should trigger events so that users and off-chain monitoring tools can account for important state changes.

```
if ((protocolFeeRecipient != address(0)) && (protocolFeeAmount != 0)) {
    IERC20(currency).safeTransferFrom(from, protocolFeeRecipient, protocolFeeAmount);
    finalSellerAmount -= protocolFeeAmount;
}
```

Figure 8.1: Protocol fee transfer in `_transferFeesAndFunds` function (`contracts/executionStrategies/StrategyDutchAuction.sol#L440-L443`)

Exploit Scenario

A smart contract wallet provider has a LooksRare integration that enables its users to buy and sell NFTs. The front end relies on information from LooksRare's subgraph to itemize prices, royalties, and fees. Because the system does not emit an event when a protocol fee is incurred, an under-calculation in the wallet provider's accounting leads its users to believe they have been overcharged.

Recommendations

Short term, add events for all critical operations that transfer value, such as when a protocol fee is assessed. Events are vital aids in monitoring contracts and detecting suspicious behavior.

Long term, consider adding or accounting for a new protocol fee event in the LooksRare subgraph and any other off-chain monitoring tools LooksRare might be using.

9. Taker orders are not EIP-712 signatures

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LR-9
Target: `contracts/libraries/OrderTypes.sol`	

Description

When takers attempt to match order proposals, they are presented with an obscure blob of data. In contrast, makers are presented with a formatted data structure that makes it easier to validate transactions.

```
struct TakerOrder {
    bool isOrderAsk; // true --> ask / false --> bid
    address taker; // msg.sender
    uint256 price; // final price for the purchase
    uint256 tokenId;
    uint256 minPercentageToAsk; // // slippage protection (9000 --> 90% of the final price
must return to ask)
    bytes params; // other params (e.g., tokenId)
}
```

Figure 9.1: The `TakerOrder` struct in `OrderTypes.sol:31-38`

While this issue cannot be exploited directly, it creates an asymmetry between the user experience (UX) of makers and takers. Because of this, users depend on the information that the user interface (UI) displays to them and are limited by the UX of the wallet software they are using.

Exploit Scenario 1

Eve, a malicious user, lists a new collection with the same metadata as another, more popular collection. Bob sees Eve's listing and thinks that it is the legitimate collection. He creates an order for an NFT in Eve's collection, and because he cannot distinguish the parameters of the transaction he is signing, he matches it, losing money in the process.

Exploit Scenario 2

Alice, an attacker, compromises the UI, allowing her to manipulate the information displayed by it in order to make illegitimate collections look legitimate. This is a more extreme exploit scenario.

Recommendations

Short term, evaluate and document the current UI and the pitfalls that users might encounter when matching and creating orders.

Long term, evaluate whether adding support for EIP-712 signatures in `TakerOrder` would minimize the issue and provide a better UX.

10. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-LR-10
Target: LooksRare protocol	

Description
The LooksRare contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the LooksRare contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

11. isContract may behave unexpectedly

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-LR-11
Target: `contracts/libraries/SignatureChecker.sol`	

Description

The LooksRare exchange relies on OpenZeppelin's `SignatureChecker` library to verify signatures on-chain. This library, in turn, relies on the `isContract` function in the `Address` library to determine whether the signer is a contract or an externally owned account (EOA). However, in Solidity, there is no reliable way to definitively determine whether a given address is a contract, as there are several edge cases in which the underlying `extcodesize` function can return unexpected results.

```
function isContract(address account) internal view returns (bool) {
    // This method relies on extcodesize, which returns 0 for contracts in
    // construction, since the code is only stored at the end of the
    // constructor execution.

    uint256 size;
    assembly {
      size := extcodesize(account)
    }
    return size > 0;
}
```

Figure 11.1: The `isContract` function in `Address.sol#L27-37`

Exploit Scenario

A maker order is created and signed by a smart contract wallet. While this order is waiting to be filled, `selfdestruct` is called on the contract. The call to `extcodesize` returns 0, causing `isContract` to return false. Even though the order was signed by an ERC1271-compatible contract, the `verify` method will attempt to validate the signer's address as though it were signed by an EOA.

Recommendations

Short term, clearly document for developers that `SignatureChecker.verify` is not guaranteed to accurately distinguish between an EOA and a contract signer, and emphasize that it should never be used in a manner that requires such a guarantee.

Long term, avoid adding or altering functionality that would rely on a guarantee that a signature's source remains consistent over time.

12. tokenId and amount fully controlled by the order strategy when matching two orders

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LR-12
Target: `contracts/LooksRareExchange.sol`	

Description

When two orders are matched, the strategy defined by the `MakerOrder` is called to check whether the order can be executed.

```
function matchAskWithTakerBidUsingETHAndWETH(
    OrderTypes.TakerOrder calldata takerBid,
    OrderTypes.MakerOrder calldata makerAsk
) external payable override nonReentrant {
    [...]

    // Retrieve execution parameters
    (bool isExecutionValid, uint256 tokenId, uint256 amount) =
IExecutionStrategy(makerAsk.strategy)
        .canExecuteTakerBid(takerBid, makerAsk);

    require(isExecutionValid, "Strategy: Execution invalid");
    [...]
}
```

Figure 12.1: `matchAskWithTakerBidUsingETHAndWETH` (`LooksRareExchange.sol#186-212`)

The strategy call returns a boolean indicating whether the order match can be executed, the `tokenId` to be sold, and the `amount` to be transferred. The `LooksRareExchange` contract does not verify these last two values, which means that the strategy has full control over them.

```
function matchAskWithTakerBidUsingETHAndWETH(
    OrderTypes.TakerOrder calldata takerBid,
    OrderTypes.MakerOrder calldata makerAsk
) external payable override nonReentrant {
    [...]
    // Execution part 1/2
    _transferFeesAndFundsWithWETH(
        makerAsk.strategy,
        makerAsk.collection,
        tokenId,
        makerAsk.signer,
        takerBid.price,
        makerAsk.minPercentageToAsk
    );

    // Execution part 2/2
    _transferNonFungibleToken(makerAsk.collection, makerAsk.signer, takerBid.taker, tokenId,
amount);

    emit TakerBid(
        askHash,
        makerAsk.nonce,
        takerBid.taker,
        makerAsk.signer,
        makerAsk.strategy,
        makerAsk.currency,
        makerAsk.collection,
        tokenId,
        amount,
        takerBid.price
    );
}
```

Figure 12.2: `matchAskWithTakerBidUsingETHAndWETH` (`LooksRareExchange.sol#217-228`)

This ultimately means that a faulty or malicious strategy can cause a loss of funds (e.g., by returning a different `tokenId` from the one that was intended to be sold or bought).

Additionally, this issue may become problematic if strategies become trustless and are no longer developed or allowlisted by the LooksRare team.

Exploit Scenario

A faulty strategy, which returns a different `tokenId` than expected, is allowlisted in the protocol. Alice creates a new order using that strategy to sell one of her tokens. Bob matches Alice's order, but because the `tokenId` is not validated before executing the order, he gets a different token than he intended to buy.

Recommendations

Short term, evaluate and document this behavior and use this documentation when integrating new strategies into the protocol.

Long term, consider adding further safeguards to the `LooksRareExchange` contract to check the validity of the `tokenId` and the `amount` returned by the call to the strategy.

13. Risk of phishing due to data stored in maker order params field

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LR-13
Target: `LooksRare/contracts/`	

Description

The `MakerOrder` struct contains a `params` field, which holds arbitrary data for each strategy. This storage of data may increase the chance that users could be phished.

```
struct MakerOrder {
    bool isOrderAsk; // true --> ask / false --> bid
    address signer; // signer of the maker order
    address collection; // collection address
    uint256 price; // price (used as )
    uint256 tokenId; // id of the token
    uint256 amount; // amount of tokens to sell/purchase (must be 1 for ERC721, 1+ for
ERC1155)
    address strategy; // strategy for trade execution (e.g., DutchAuction,
StandardSaleForFixedPrice)
    address currency; // currency (e.g., WETH)
    uint256 nonce; // order nonce (must be unique unless new maker order is meant to
override existing one e.g., lower ask price)
    uint256 startTime; // startTime in timestamp
    uint256 endTime; // endTime in timestamp
    uint256 minPercentageToAsk; // slippage protection (9000 --> 90% of the final price must
return to ask)
    bytes params; // additional parameters
    uint8 v; // v: parameter (27 or 28)
    bytes32 r; // r: parameter
    bytes32 s; // s: parameter
}
```

Figure 13.1: The `MakerOrder` struct in `contracts/libraries/OrderTypes.sol#L12-29`

In the Dutch auction strategy, the maker `params` field defines the start price for the auction. When a user generates the signature, the UI must specify the purpose of `params`.

```
function canExecuteTakerBid(OrderTypes.TakerOrder calldata takerBid, OrderTypes.MakerOrder
calldata makerAsk)
    external
    view
```

```
    override
    returns (
        bool,
        uint256,
        uint256
    )
{
    uint256 startPrice = abi.decode(makerAsk.params, (uint256));
    uint256 endPrice = makerAsk.price;
}
```

Figure 13.2: The canExecuteTakerBid function in contracts/executionStrategies/StrategyDutchAuction.sol#L39-L70

When used in a `StrategyPrivateSale` transaction, the `params` field holds the buyer address that the private sale is intended for.

```
function canExecuteTakerBid(OrderTypes.TakerOrder calldata takerBid, OrderTypes.MakerOrder
calldata makerAsk)
    external
    view
    override
    returns (
        bool,
        uint256,
        uint256
    )
{
    // Retrieve target buyer
    address targetBuyer = abi.decode(makerAsk.params, (address));

    return (
        ((targetBuyer == takerBid.taker) &&
            (makerAsk.price == takerBid.price) &&
            (makerAsk.tokenId == takerBid.tokenId) &&
            (makerAsk.startTime <= block.timestamp) &&
            (makerAsk.endTime >= block.timestamp)),
        makerAsk.tokenId,
        makerAsk.amount
    );
}
```

Figure 13.3: The canExecuteTakerBid function in contracts/executionStrategies/StrategyPrivateSale.sol

Exploit Scenario

Alice receives an EIP-712 signature request through MetaMask. Because the value is masked in the `params` field, Alice accidentally signs an incorrect parameter that allows an attacker to match.

Recommendations

Short term, document the expected values for the `params` value for all strategies and add in-code documentation to ensure that developers are aware of strategy expectations.

Long term, document the risks associated with off-chain signatures and always ensure that users are aware of the risks of signing arbitrary data.

14. Use of legacy openssl version in solidity-coverage plugin

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-LR-14
Target: LooksRare protocol	

Description

The `LooksRare` codebase uses a version of `solidity-coverage` that relies on a legacy version of `openssl` to run. While this plugin does not alter protocol contracts deployed to production, the use of outdated security protocols anywhere in the codebase may be risky or prone to errors.

```
Error in plugin solidity-coverage: Error: error:0308010C:digital envelope
routines::unsupported
```

Figure 14.1: Error raised by `npx hardhat coverage`

Recommendations

Short term, refactor the code to use a new version of `openssl` to prevent the exploitation of `openssl` vulnerabilities.

Long term, avoid using outdated or legacy versions of dependencies.

15. TypeScript compiler errors during deployment

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-LR-15
Target: LooksRare protocol	

Description
TypeScript throws an error while trying to compile scripts during the deployment process.

```
scripts/helpers/deploy-exchange.ts:29:5 - error TS7053: Element implicitly has an 'any' type
because expression of type 'string' can't be used to index type '{ mainnet: string; rinkeby:
string; localhost: string; }'. No index signature with a parameter of type 'string' was
found on type '{ mainnet: string; rinkeby: string; localhost: string; }'.

29     config.Fee.Standard[activeNetwork]

       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 15.1: TypeScript error raised by `npx hardhat run --network localhost scripts/hardhat/deploy-hardhat.ts`

In the `config.ts` file, the `config` object does not explicitly allow `string` types to be used as an index type for accessing its keys. Hardhat assigns a `string` type as the value of `activeNetwork`. As a result, TypeScript throws a compiler error when it tries to access a member of the `config` object using the `activeNetwork` value.

Recommendations
Short term, add type information to the `config` object that allows its keys to be accessed using `string` types.

Long term, ensure that TypeScript can compile properly without errors in any and every potential context.

Summary of Recommendations

The LooksRare exchange and token staking contracts follow smart-contract best practices. Trail of Bits recommends that LooksRare address the findings detailed in this report and take the following additional steps prior to deployment:

- Use fuzzing and/or symbolic execution to ensure the correctness of the arithmetic operations. Use the system invariants listed in appendix F as a starting point for property-based testing.
- Develop a detailed incident response plan to ensure that any issues that arise can be addressed promptly and without confusion.
- Modularize the test suite by segmenting it into a setup script and isolated test cases. Additionally, standardize the different types of `MakerOrder` and `TakerOrder` structs used for testing, and import/customize them as necessary for each test case.
- Provide documentation and warnings to end users regarding the risks of phishing attacks. Here are a few risks and recommendations that LooksRare and its users should be aware of:
    - Although EIP-712 signatures are more resilient to phishing attacks, the use of these signatures does not completely mitigate the risk. User funds can still be stolen through a phishing campaign that asks for signatures outside the context of the LooksRare application. Inform users of what a maker order signature should look like and what kinds of malicious values they should be wary of.
    - The `MakerOrder` struct's ambiguous data parameter (TOB-LR-13) introduces the risk that users could sign data that may lead to undefined behavior. Document the various uses of this data parameter and update it as more strategies are added.
    - Ensure that the system is not vulnerable to domain or subdomain takeovers by malicious parties. Additionally, inform users that the LooksRare exchange is hosted only at looksrare.org and nowhere else.
    - Carefully review the information included in web requests to third-party servers for potential privacy breaches. For example, an AJAX request in a user's browser to fetch metadata or images may expose the user's IP address to a nefarious actor that has access to that server's logs, allowing the actor to match IP address locations to wallet addresses.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Calls	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Off-Chain Orderbook Assumptions and Recommendations

The LooksRare orderbook is a centralized off-chain service that manages maker orders (which are EIP-712 signatures) and is critical to the functionality of the LooksRare protocol. This makes it a single point of failure. Because this component was out of scope for this audit, we made a series of assumptions about how it behaves.

Below, we document those assumptions so that the LooksRare team can examine them, maintain them to help drive internal development, and, if those assumptions do not hold true, evaluate whether they should be implemented. We also offer recommendations for improving the security of the orderbook and mitigating risks.

Order Management

Because maker orders are EIP-712 signatures, they are always valid, even if they cannot be matched, and might be stored in the orderbook for a long period of time.

Therefore, it is very important that orders be validated both at the time of creation and after creation at regular intervals. Below are the checks that we assume the system performs during maker order validation:

- The system verifies that the signature matches the order signer parameter; if the signer is a contract using EIP-1271, the system calls the `isValidSignature()` function to verify that the signature is valid.
- The system verifies that the collection is supported by the on-chain exchange.
- The system verifies that the maker owns the NFT and the amount to be sold.
- The system verifies the start and end time of the order to ensure either that the order does not immediately expire or that it can be matched only after a substantial amount of time has passed (depending on the LooksRare team's definition of "substantial").
- The system verifies that the strategy is allowlisted by the on-chain exchange.
- The system verifies that the currency is allowlisted by the on-chain exchange.
- The system verifies token approvals for all NFT and ERC20 tokens accepted.
- The system validates and compares the nonce generated by the off-chain orderbook against the user's minimum nonce defined on the on-chain exchange.

- The system verifies that the nonce generated by the off-chain orderbook monotonically increases unless the order is meant to override a previous one.
- The system flags reverting transactions and possibly filters them out from the orderbook if they have reverted several times. This prevents order takers from being griefed by invalid transactions that may have passed the previous checks.
- LooksRare periodically recommends that users invalidate their old orders on-chain to minimize the risk of old orders being matched on-chain if they become valid in the future.

Failure to enforce these checks can allow users to conduct griefing attacks on order takers since they are responsible for matching orders on-chain and paying for the gas.

Additionally, the LooksRare team should evaluate whether orders should be removed from the orderbook when they are invalidated on-chain or if they should simply be filtered out.

Order Overriding

The LooksRare orderbook allows makers to create a new order with the same nonce as an existing order (i.e., to override an order). However, the orderbook cannot invalidate the original order's signature, which will always remain valid. Since the orderbook is visible to other users, those who had access to the original order before it was filtered out or removed from the orderbook could attempt to match the first order if it is more profitable than matching the second.

We recommend taking the following steps to mitigate this risk:

- Through documentation, inform users that they should not override orders to fix mistakes; instead, users should invalidate orders by calling the `LooksRareExchange` contract (though this action could still be vulnerable to front-running).
- Inform users that if someone had access to their original order, it may still be matched against if it is more profitable to do so. If the maker's intention is for the original order to become invalid, the maker should invalidate the order on-chain.

Off-Chain Service Neutrality and Security Considerations

Since the orderbook is a centralized off-chain service, it is critical that it keeps its neutrality, does not give privileged access to any actors, and is protected against attacks. To ensure that the system is resilient against malicious behavior and that users have the same access to the orderbook, consider the following recommendations:

- **Ensure that the orderbook is sufficiently protected against denial-of-service conditions due to spam.** Maker orders are gasless signatures and rely on

traditional server-hosted infrastructure to keep track of them. This means that the orderbook must have sufficient protection against a large number of requests.

- **Ensure that the orderbook does not arbitrarily censor maker orders.** The expected "safe" values for all parameters signed in a transaction must be explicitly defined. The orderbook must drop *only* transactions that fail to meet these explicit requirements.
- **Ensure that no user has privileged access to the orderbook.** All users should have the same access to the maker order database, and new orders should be immediately available to the public. If this is not the case, then privileged users will have access to newly posted orders before other users do, giving them a competitive advantage over others and allowing them to extract value out of the orderbook (e.g., by analyzing newly posted orders to get information on the market).

D. Risks Associated with Strategies

One feature of the LooksRare protocol is its support for multiple types of maker orders. For example, a user may sell a specific token or any item in a collection at a given price. To provide the functionality related to each order type and to support future order types, the system includes a smart contract module called `ExecutionManager`, which allows the admin to add or remove order types.

These order types, internally known as "strategies," are smart contracts that perform a series of validations for each order type to determine whether the order can be executed. For example, in a standard sale (i.e., in which a token is sold at a fixed price), the strategy will verify that the order is not expired and that the `tokenId` and `price` defined in the maker order match those defined in the taker order.

Since these strategies determine the validity of an order, it is crucial that they behave correctly; incorrect strategy behavior could lead to the loss of funds. For this reason, we provide the following recommendations to help the LooksRare team create an internal due-diligence process to validate the correctness of new strategies:

- **Avoid using upgradeable strategies.** Supporting upgradeable strategies, either through proxy contracts or metamorphic contracts, means that these contracts can be arbitrarily changed by the strategy owner at any point, which creates a significant trust issue for users. For example, orders that were previously invalid could be upgraded to become valid.

  Additionally, the use of `delegatecall` to make strategy contracts upgradeable could introduce further security risks. For example, there might be storage clashes between the proxy and implementation contracts, or poorly executed upgrades could leave the new implementation unusable.

- **Prioritize using immutable variables over mutable ones.** Since the strategy contracts determine the validity of an order match, state changes to them should be minimal to prevent situations in which the internal state of the strategy affects the validation of an order.

- **Document each strategy's functionality, properties, and risks.** Given the strategy contracts' importance, each strategy's execution flow and associated risks should be thoroughly documented.

- **Validate and compare strategy outputs with the associated `MakerOrder` and `TakerOrder`.** As stated in TOB-LR-12, strategies have full control over the `tokenId` and `amount` used when settling an order match. Therefore, the LooksRare team should consider adding additional safeguards to the `LooksRareExchange` contract

to prevent cases in which a poorly coded or malicious strategy returns a different `tokenId` or `amount` than those intended by the maker and taker.

- **Develop unit tests for each strategy.** Ensure that all “happy paths” and expected revert flows are tested in the contracts prior to deployment.
- **Test each strategy’s properties with Echidna.** Since some strategies will perform mathematical calculations and proofs to determine order validity, we recommend using Echidna to test any invariants or properties that result from those computations. We recommend properties and invariants to test in appendix F.

E. Roles and Privileges

A survey of the roles controlled by a multisig wallet or another contract and their associated privileges is provided below.

Role	Privileges
`LooksRareExchange` (multisig owner)	• Updating the `CurrencyManager`, `ExecutionManager`, `RoyaltyFeeManager`, and `TransferSelectorNFT` addresses • Updating the `protocolFeeRecipient` address
`CurrencyManager` (multisig owner)	• Adding currencies to and removing them from the `_whitelistedCurrencies` allowlist
`ExecutionManager` (multisig owner)	• Adding strategies to and removing them from the `_whitelistedStrategies` allowlist
`TransferSelectorNFT` (multisig owner)	• Adding `transferManagers` to and removing them from an NFT collection
`RoyaltyFeeSetter` (multisig owner)	• Updating royalty information for an NFT collection • Updating the owner of the `RoyaltyFeeRegistry` contract • Updating the `royaltyFeeLimit` for the `RoyaltyFeeRegistry` contract
`RoyaltyFeeRegistry` (`RoyaltyFeeSetter` owner)	• Updating royalty information for an NFT collection • Updating the `royaltyFeeLimit`
`StrategyDutchAuction` (multisig owner)	• Updating `minimumAuctionLengthInSeconds`
`LooksRareToken` (`TokenDistributor` owner)	• Minting LOOKS tokens
`FeeSharingSystem` (`FeeSharingSetter` owner)	• Updating the reward per block for a given block duration

F. System Invariants

This appendix lists the properties of the LooksRare token staking system that should always hold true. We recommend testing these invariants through unit tests and property-based testing with Echidna. We also recommend documenting any additional system properties.

Access Controls

- After a transfer of contract ownership, the new owner can call privileged owner-only functions. This should prevent the old owner from invoking any of these privileged functions.

Arithmetic

- Rewards are always calculated before the effects (i.e., deposit or withdrawal) and are never calculated more than once per block. If rewards were calculated more than once per block, users would be able to manipulate their balances by taking out flash loans to earn more rewards than they should.
- `TokenDistributor._getMultiplier()` always returns a value within the range of `[0, endBlock - from]`.
- Invoking the `TokenDistributor.deposit()` function results in the following:
  - A decrease in the `msg.sender`'s LOOKS token balance
  - An increase in the `TokenDistributor` contract's token balance
  - An increase in the user's token balance and reward debt amounts
- Invoking the `TokenDistributor.withdraw()` function results in the following:
  - A decrease in the `TokenDistributor` contract's LOOKS token balance
  - An increase in the `msg.sender`'s token balance
  - A decrease in the user's amount and reward debt amounts
- Calling `TokenDistributor.withdrawAll()` always results in zeroed user amount and reward debt balances.
- `TokenDistributor.deposit()` and `TokenDistributor.withdraw()` are inverse functions. Executing a deposit followed by a withdrawal in the contracts should result in the same state pre- and post-call.

- Multiple calls to `TokenDistributor.deposit()` followed by a call to `TokenDistributor.withdrawAll()` are inverses of each other. Executing multiple deposits followed by a call to `withdrawAll` in the contracts should result in the same state pre- and post-call. The same invariant can be tested by providing the sum of all deposits in an argument for a call to `withdraw`.

Token Properties (LOOKS Token)

- The `totalSupply` of tokens never exceeds the `SUPPLY_CAP` of `1,000,000,000`.
- Transferring tokens to `address(0)` always reverts.
- The `null` address cannot own any tokens.
- Transferring a valid number of tokens to a non-`null` address reduces the sender's balance and increases the recipient's balance.
- Transferring an invalid number of tokens to a non-`null` address always reverts or returns false.
- A self-transfer keeps token balances constant.
- Approving tokens overwrites the previous allowance value.
- Balances are always consistent with the total supply of tokens.
- The circulating token supply is always equivalent to the total supply, deducting splitter, vesting, and private sale contract balances.
- Only the token owner, `TokenDistributor`, can mint tokens.

Configuration

- A call to `FeeSharingSystem.updateRewards()` with the correct preconditions updates the `lastUpdateBlock` and `periodEndBlock`.

G. Incident Response Recommendations

In this section, we provide recommendations around the formulation of an incident response plan.

- **Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
    - Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.
- **Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.**
    - Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.
- **Clearly describe the intended process of contract deployment.**
- **Consider whether and under what circumstances LooksRare will make affected users whole after certain issues occur.**
    - Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.
- **Document how LooksRare plans keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - For each language and component, describe noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.
- **Consider scenarios involving issues that would indirectly affect the system.**
- **Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.) during an incident.**
    - Some issues may require collaboration with external parties to efficiently remediate them.

- **Define contract behavior that is considered abnormal for off-chain monitoring.**
    - Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.
- **Combine issues and determine whether new detection and mitigation scenarios are needed.**
- **Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.**
    - Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

H. Code Quality Recommendations

Exchange

- **Consider defining variables as public and using default Solidity getter functions instead of marking variables as private and using a custom getter function.**
    - `LooksRareExchange`: `isUserOrderNonceExecutedOrCancelled`
    - `RoyaltyFeeRegistry`: `royaltyFeeInfoCollection`
    - `RoyaltyFeeSetter`: `viewProtocolFee`
    - `LooksRareToken`: `SUPPLY_CAP`
- **Use strict Solidity pragmas throughout the codebase.**
- **Define variables for constants.** This will improve the codebase's readability.
    - `LooksRareExchange.sol#L160`
    - `RoyaltyFeeRegistry.sol#L32`
- **Move `require` checks into modifiers for code reuse.**
- **Explicitly define signature hashes in code instead of using hard-coded hashes.** This will improve the codebase's readability.
- **Avoid using `try-catch` statements with empty `catch` statements.**

Token Staking

- **Use the same `PRECISION_FACTOR` across contracts (`TokenDistributor`, `FeeSharingSystem`).** Use fuzz testing to verify that all arithmetic operations execute as expected.
- **Add a check to `withdrawAll()` that ensures the user has sufficient shares to execute the withdrawal.**
- **Add a check of the `mint` function's return value, regardless of whether the operation will lead to state changes.** This will improve code consistency and decrease the likelihood of undefined behavior.

I. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of ERC721 Tokens

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.
- ❏ **Token-specific logic is sound, and the contracts account for unexpected behavior.** Some non-standard ERC721 tokens allow users to own only a single NFT. This means that transfer functions may also burn tokens.