ThunderNFT Audit Report

By Safe Edges

AUDIT

ThunderNFT Smart Contract Audit Report

Introduction

A time-boxed security review of the **ThunderNFT** protocol was performed by **SafeEdges**, focusing on the security aspects of the smart contract's implementation.

Disclaimer

A smart contract security review can never guarantee the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort aimed at identifying as many vulnerabilities as possible. However, I cannot assure 100% security after the review or that all issues will be discovered. For better protection, subsequent security reviews, bug bounty programs, and on-chain monitoring are highly recommended.

About Safe edges

Safedges, or **SafeEdges**, is an smart contract security company. With experience in finding numerous vulnerabilities in various protocols, he is dedicated to contributing to the blockchain ecosystem through security research and audits. Feel free to explore his previous work or reach out on Twitter @piyushshukla__.

About ThunderNFT

The ThunderNFT protocol facilitates NFT auctions and sales by leveraging its unique **ExecutionStrategy** smart contract. It implements a fixed-price sale strategy and allows users to list, bid, and buy NFTs. The system also allows royalties management for NFT creators.

Security Assessment Summary

- **Review commit hash**: 260c9859e2cd28c188e8f6283469bcf57c9347de
- **Fixes implemented**: No fixes implemented at the time of this audit.

Scope

The following smart contracts were included in the audit:

- `ExecutionStrategy`
- `ThunderExchange`
- `Pool`
- `Ownership Management`
- `Royalty Manager`

▼ Focus Area During Audit

1. Reentrancy
2. Integer Overflow
3. Integer Underflow
4. Division by Zero
5. Unauthorized Access Control
6. Insufficient Gas Limit
7. Timestamp Dependency
8. Transaction-Ordering Dependence (TOD)
9. Front-Running
10. Denial of Service (DoS)
11. Logic Errors

12. Unhandled Exceptions
13. Insecure Randomness
14. Improper Error Handling
15. Lack of Input Validation
16. Incomplete Delegation
17. Incorrect Balance Handling
18. Unintended Token Creation
19. Incorrect Token Transfer
20. Misappropriation of Funds
21. Function Visibility Misconfiguration
22. Insecure External Calls
23. Failure to Update State
24. Incorrect Event Handling
25. Unprotected Self-Destruct
26. External Dependency Risks
27. Timestamp Manipulation
28. Delegatecall to Untrusted Contracts
29. Frozen Wallets
30. Incorrect Implementation of Standards (e.g., ERC-20, ERC-721)
31. Insecure Oracle Usage
32. Lack of Multisig Authentication
33. Incorrect Access Control Lists
34. Insecure Parameters
35. Lack of Upgradability Mechanism
36. Improper Token Allowances
37. Race Conditions

38. Difficulty Adjustment Attacks
39. Time Zone Vulnerabilities
40. Gas Limit Vulnerabilities
41. Lack of Proper Documentation
42. Misuse of Low-Level Calls
43. Dependency on External Contracts
44. Insider Attacks
45. Phantom Token Attacks
46. Fallback Function Vulnerabilities
47. Incorrect Constructor Logic
48. Chain Split Attacks
49. Unprotected Private Data
50. Insecure Compiler Version

Detailed Findings

1. Front-Running Vulnerability Due to Protocol Fee Initialization Delay

Brief

The contract is vulnerable to front-running attacks due to the protocol fee not being set during initialization. Attackers can exploit this delay between deployment and the protocol fee being set by submitting multiple fee-free orders.

Impact

Significant business logic impact to the protocol due to potential order flooding without fees.

Proof of Concept

1. Deploy the contract with the protocol fee set to zero.
2. Submit multiple orders immediately after deployment but before the owner can call the `set_protocol_fee` function.
3. Attackers monitoring the mempool can rapidly submit orders without fees, exploiting this gap.

Code Reference

Strategy Fixed Price Sale – Smart Contract Code

```
impl ExecutionStrategy for Contract {
    /// Initializes the contract, sets the owner, and Thunder
Exchange contract
    #[storage(read, write)]
    fn initialize(exchange: ContractId) {
        require(!_is_initialized(), StrategyFixedPriceError
s::Initialized);
        storage.is_initialized.write(true);
        let caller = get_msg_sender_address_or_panic();
        storage.owner.set_ownership(Identity::Address(calle
r));
        storage.exchange.write(Option::Some(exchange));
    }
}
```

Recommendation

Modify the `initialize` function to require the protocol fee as a parameter and limit its value.

```
fn initialize(exchange: ContractId, fee: u64) {
    require(!_is_initialized(), StrategyFixedPriceErrors::Ini
tialized);
    require(fee <= 500, StrategyFixedPriceErrors::FeeTooHig
h);
    // Existing code to set ownership and exchange contract
}
```

---

2. Funds Not Locked During Order Placement

Brief

The `place_order` function in the ThunderExchange contract does not lock user funds during the order placement process. This flaw allows users to place multiple buy orders with the same balance, leading to potential losses.

Impact

Users can place multiple orders without sufficient funds, leading to financial losses for the platform.

Proof of Concept

1. User A has 100 units in payment_asset.
2. They place a buy order for 100 units and quickly place another order before their balance is updated.
3. Both orders are accepted, leading to an overdraft.

Code Reference

Place Order Function – Thunder Exchange

```
fn place_order(order_input: MakerOrderInput) {
    _validate_maker_order_input(order_input);
    let strategy = abi(ExecutionStrategy, order_input.strateg
y.bits());
    let order = MakerOrder::new(order_input);
    match order.side {
        Side::Buy => {
            let pool_balance = _get_pool_balance(order.maker,
order.payment_asset);
            require(order.price <= pool_balance, ThunderExcha
ngeErrors::AmountHigherThanPoolBalance);
        },
        Side::Sell => {
            // Validations
        },
    }
    strategy.place_order(order);
    log(OrderPlaced { order });
}
```

Recommendation

Introduce a mechanism to lock user funds before processing the order.

```
fn place_order(order_input: MakerOrderInput) {
    _validate_maker_order_input(order_input);
    let strategy = abi(ExecutionStrategy, order_input.strateg
y.bits());
    let order = MakerOrder::new(order_input);

    match order.side {
        Side::Buy => {
            let pool_balance = _get_pool_balance(order.maker,
order.payment_asset);
            require(order.price <= pool_balance, ThunderExcha
ngeErrors::AmountHigherThanPoolBalance);
            _lock_funds(order.maker, order.payment_asset, ord
er.price); // Lock the funds
```

```
        },
        // Side::Sell logic
    }
    strategy.place_order(order);
    log(OrderPlaced { order });
}
```

3. Cached Balance Vulnerability in Transfer Function

Brief

The `_transfer` function in the pool contract relies on cached balances for both the sender and recipient. This can lead to a double-spend or token duplication vulnerability.

Impact

Attackers can exploit cached balances, leading to token duplication and financial drain from the contract.

Proof of Concept

When the sender and recipient addresses are identical, the cached balance is incremented twice, causing the user's balance to double.

Code Reference

Transfer Function Code

```
fn _transfer(from: Identity, to: Identity, asset: AssetId, am
ount: u64) {
    let from_balance = _balance_of(from, asset);
    let to_balance = _balance_of(to, asset);
    require(from_balance >= amount, PoolErrors::AmountHigherT
hanBalance);

    storage.balance_of.insert((from, asset), from_balance - a
mount);
```

```
    storage.balance_of.insert((to, asset), to_balance + amoun
t);

    log(Transfer { from, to, asset, amount });
}
```

Recommendation

Check if the sender and recipient are the same and avoid balance update in that case.

```
fn _transfer(from: Identity, to: Identity, asset: AssetId, am
ount: u64) {
    if from == to {
        let balance = _balance_of(from, asset);
        require(balance >= amount, PoolErrors::AmountHigherTh
anBalance);
        log(Transfer { from, to, asset, amount });
        return;
    }
    // Proceed with regular transfer logic if from != to
}
```

4. Missing Logging in Ownership Functions

Brief

The `transfer_ownership` and `renounce_ownership` functions do not log events. This can lead to ownership changes being untracked, making it difficult to audit the contract.

Impact

Lack of auditability and security around ownership changes.

Code Reference

Transfer Ownership Function

```
fn transfer_ownership(new_owner: Identity) {
    storage.owner.only_owner();
    storage.owner.transfer_ownership(new_owner);
}
```

Recommendation

Use the ownable library or manually add logging functionality for ownership transfers.

---

5. Zero Balance Storage Retention in Withdrawals

Brief

The `withdraw` function retains entries in storage for accounts with zero balances, leading to increased storage costs.

Impact

Unnecessary storage costs due to the retention of zero-balance entries.

Recommendation

Modify the withdraw function to remove zero-balance entries from storage.

```
if new_balance == 0 {
    storage.balance_of.remove(&(sender, asset));
} else {
    storage.balance_of.insert((sender, asset), new_balance);
}
```

---

6. Ownership Assignment to Zero Address

Brief

The contract allows ownership transfer to a zero address, which can lead to loss of control over the contract.

Impact

Loss of contract functionality due to an invalid owner.

Recommendation

Add a validation to prevent transferring ownership to the zero address.

```
require(!Ownership::is_zero_address(new_owner), AccessError::
CannotReinitialized);
```

---

7. Royalty Fee Governance Flaw

Brief

The contract allows the owner to set the royalty fee limit, which should be restricted to the Royalty Manager or NFT collection owner to ensure proper governance.

Impact

Misalignment between contract owners and NFT collection stakeholders.

Recommendation

Limit the ability to set royalty fees to the Royalty Manager or collection owner.

This report provides a comprehensive review of the identified issues and recommended solutions for your smart contracts. Addressing these vulnerabilities is critical to ensuring the security and efficiency of the platform.

Previous Security Audit Reports

No previous audits for ThunderNFT have been publicly disclosed.

Observations

The `ThunderNFT` protocol's architecture allows for a flexible implementation, but care should be taken with regards to execution strategies and fund management. The lack of fund locking and delayed fee setup create vulnerabilities that must be addressed to ensure the integrity of the protocol.

Here's an updated report using the original structure, but now focused on the MiraAMM project:

MiraAMM Smart Contract Audit Report