ENTERSOFT

# DorkFi

## Smart Contract Audit

DATE: 7th November 2025 | PREPARED BY: Entersoft Singapore

# Contents

# Revision History & Version Control

| Start Date | End Date | Author | Comments/Details |
|---|---|---|---|
| 12 August 2025 | 07 November 2025 | Tharun Bethina | Final Release for the Client |

| Reviewed by | Released by |
|---|---|
| Nishita Palaksha | Nishita Palaksha |

Entersoft was commissioned to review DorkFi's Algorand smart contracts. The review was conducted between August 12, 2025, and November 7, 2025. The report includes:

- **Executive Summary:** A high-level overview of the security audit findings.
- **Technical analysis:** Our detailed analysis of the Smart Contract code

# 1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to: (i) smart contract best coding practices and vulnerabilities in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

# 2.0 Overview

## 2.1 Project Overview

During the period of **12 August 2025 to 07 November 2025**, Entersoft performed smart contract security audits for **DorkFi**.

## 2.2 Scope

The audit analyzed and documented the smart contract codebase for quality, security, and correctness.

**Reviewed file:**

- https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Out-of-scope:** External contracts, oracles, other smart contracts in the repository, or imported smart contracts.

## 2.3 Project Summary

| Project Name | No. of Smart Contract File(s) | Verified | Vulnerabilities |
|---|---|---|---|
| DorkFi | 1 | Yes | As per report. Section 2.6 |

## 2.4 Audit Summary

| Delivery Date | Method of Audit | Consultants Engaged |
|---|---|---|
| 07 November 2025 | Manual Testing Supplemented with Automated Analysis | 1 |

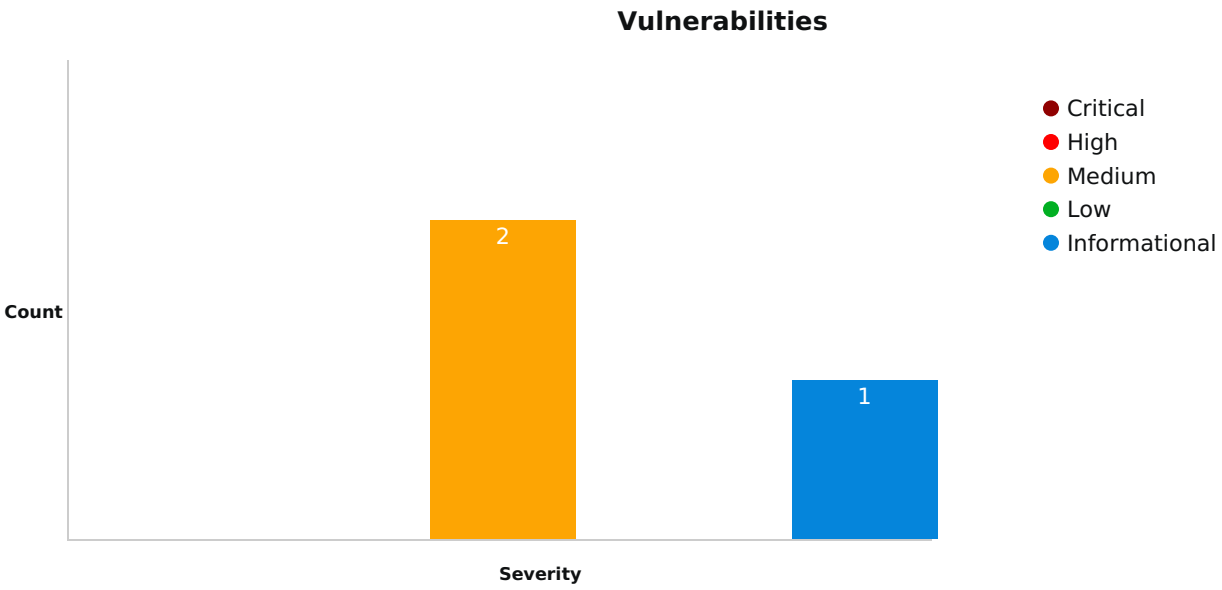
## 2.5 Security Level References

Every vulnerability in this report was assigned a severity level from the following classification table:

| | | Impact | | | | |
|---|---|---|---|---|---|---|
| | | Minimal | Low | Medium | High | Critical |
| Likelihood | Critical | Minimal | Low | Medium | High | Critical |
| | High | Minimal | Low | Medium | High | Critical |
| | Medium | Minimal | Low | Medium | Medium | High |
| | Low | Minimal | Low | Low | Low | Medium |
| | Minimal | Minimal | Minimal | Minimal | Low | |

## 2.6 Vulnerability Summary

| Critical | High | Medium | Low | Informational |
|---|---|---|---|---|
| 0 | 0 | 2 | 0 | 1 |



# 3.0 Executive Summary

Entersoft performed a comprehensive technical audit of the DorkFi Lending Pool smart contract project, implemented in Algorand Python on the Algorand blockchain. The audit aimed to identify vulnerabilities and risks in the codebase, ensuring security, reliability, and performance in line with industry standards. The review focused on prompt identification and resolution of issues to enhance the robustness of the DorkFi protocol. The audit included static analysis, dynamic testing, detailed manual review of the DorkFi Lending Pool and associated ARC200 token contracts, with attention to Algorand-specific standards and DeFi best practices. The DorkFi Lending Pool enables decentralized lending and borrowing of ARC200 tokens, managing market creation, deposits, withdrawals, borrows, repayments, liquidations, and token minting/burning. The contracts implement mechanisms for interest accrual, collateral management, ownership control, pausing, and cross-market liquidation. This audit focused on the security, correctness, and robustness of the Lending Pool's logic, especially in user balance management, interest calculation, access control, market state transitions, and ARC200 token interactions.

**Algorand Python Testing Method**

Testing followed industry standards, using static analysis, manual review, and functional testing. Contract logic was reviewed for compliance with Algorand's ARC4/ARC200 standards and smart contract safety.

**Findings and Security Posture**

During the assessment, several critical, high, medium, low, and informational vulnerabilities were identified:

- Critical: Incorrect token burn address, lack of oracle integration, pause access control, and a double-spending risk in the
- redeem function.
- High: Issues with interest accumulation during liquidation, Issues with Oracle integration and SToken liquidation logic.
- Medium: Lending pool non-minter access, low utilization, missing upper cap checks, and SToken mint validation.
- Low: Market parameter validation, same-asset borrowing risks, sanity checks and self-liquidation.
- Informational: Liquidate slippage protection and liquidation threshold calculation. Immediate attention is required for
- critical and high-severity issues, particularly those related to access control, Oracle integration, and liquidation logic.
- Addressing these will significantly improve the contract's security, stability, and resilience.

| Smart Contract Audit Testing objectives | Result |
|---|---|
| **OVERALL SECURITY POSTURE** | ● **GOOD** |

## 3.1 Findings

| Vulnerability ID | Contract Name | Severity | Status |
|---|---|---|---|
| 1 | LendingPool | Medium | Identified |
| 2 | LendingPool | Medium | Identified |
| 3 | LendingPool | Informational | Identified |

## 3.2 Recommendations

The audit initially identified critical and high-severity vulnerabilities within the DorkFi Lending Pool smart contract, particularly related to unchecked access control, missing oracle integration, and liquidation logic. While core mechanisms such as ownership and pausing were already in place, these areas required attention to ensure robustness and prevent potential risks such as fund loss or denial of service. All identified findings have since been addressed, and the implemented fixes have significantly strengthened the contract’s overall security, stability, and resilience within the decentralized lending environment.

# 4.0 Technical Analysis

## 4.1 Centralized Price Oracle Manipulation Risk

| Severity | Status | Type of Analysis |
|---|---|---|
| ● Critical | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

Price oracles are critical components in DeFi protocols that provide external market data to smart contracts. General best practices require decentralized, verifiable, and tamper-resistant price feeds to ensure fair and secure protocol operations.

Entersoft has observed that in DorkFi's LendingPool contract, the price oracle functionality is fully centralized and controlled solely by the contract owner via the set_market_price function. The owner can set market prices arbitrarily without validation, time checks, or aggregation from multiple sources.

This absence of checks or safeguards creates a high risk of price manipulation, directly impacting collateral valuation, borrowing limits, and liquidation mechanisms.

**Locations:**

File: https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

To mitigate Centralized Price Oracle Manipulation Risk, the following remediations are recommended

- Implement aggregation of price data from multiple independent and reliable sources to minimize single-point-of-failure risk.
- Introduce mechanisms such as time-weighted average prices (TWAP) or delays in price updates to prevent instant and malicious manipulation.
- Integrate established decentralized oracle networks like Chainlink or Band Protocol to provide secure and tamper-resistant price feeds.

**Impact:**

Centralized Price Oracle Manipulation Risk can impact in multiple detrimental ways, including unfair liquidations, economic manipulation, erosion of user trust, and heightened regulatory exposure.

- **Unfair Liquidations:** The owner can set prices to artificially low values, causing healthy positions to be liquidated and users to lose collateral.
- **Economic Manipulation:** Malicious actors can profit by manipulating prices before executing borrow, repay, or liquidation operations, undermining market fairness.
- **Loss of Trust:** Users may lose confidence in the protocol's security and fairness, leading to reduced adoption and reputational damage.
- **Regulatory Exposure:** Centralized price control may attract regulatory scrutiny and legal risks, especially if users suffer financial losses due to manipulation.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.2 Incorrect Token Burn Address

| Severity | Status | Type of Analysis |
|---|---|---|
| ● Critical | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

An incorrect address reference in a token-burning mechanism allows for the destruction of protocol-owned assets instead of borrower's assets. This type of logical flaw can lead to the direct depletion of a contract's reserves, as the function targets itself rather than the interacting borrower.

Entersoft has observed that the `repay()` function within the `LendingPool` contract is vulnerable due to incorrect burning of STokens from the protocol's own address (`Global.current_application_address`) instead of the borrower's address (`Txn.sender`). When a user repays their debt, this flaw causes the protocol's token balance to be reduced rather than the borrower's. This fundamentally breaks the debt repayment logic and creates a critical vulnerability that can be exploited to drain protocol funds.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Function:** repay()

**Remediation:**

To remediate this issue, the `repay()` function must be modified to burn STokens from the address of the user repaying the debt. The call to `SToken.burn_from` should target `Txn.sender` instead of `Global.current_application_address`.

This correction will ensure that tokens are properly deducted from the borrower's balance, restoring the integrity of the protocol's accounting system.

**Impact:**

The Incorrect Token Burn Address vulnerability can impact protocols in multiple ways, including protocol fund depletion, complete loss of capital, erosion of user trust, and heightened financial and regulatory risk:

- Protocol Fund Depletion: The protocol's token reserves will be systematically drained with each repayment transaction. This will open the door for malicious actors to repeatedly borrow and repay small amounts to burn the protocol's entire SToken holdings.
- Complete Loss of Capital: Attackers will be able to manipulate the protocol's accounting to their advantage. This will ultimately lead to the destabilization of the lending pool and a potential total loss of protocol-managed funds.

- Erosion of User Trust: The incident will severely damage user confidence in the protocol's ability to manage assets securely. This loss of trust will likely result in a mass withdrawal of funds, reputational damage, and a decline in user adoption.
- Financial and Regulatory Risk: The misrepresentation of assets and mismanagement of funds will expose the project to significant financial losses and potential regulatory scrutiny.

**Code Snippet:**

```
    if market_id == self.stoken_app_id:
arc4.abi_call(
    SToken.burn_from,
    arc4.Address(Global.current_application_address),
    arc4.UInt256(amount),
    app_id=Application(market_id),
)
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.3 Redeem Function Accounting Bypass

| Severity | Status | Type of Analysis |
|---|---|---|
| ● Critical | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

Proper collateral management in DeFi lending protocols depends on accurate position tracking and locked collateral states to prevent double spending or asset misappropriation. Generally, redeem functions must ensure that token burning corresponds exactly to collateral release without allowing accounting inconsistencies.

Entersoft has observed that in DorkFi, the LendingPool contract's redeem function allows users to burn NTokens and redeem underlying assets without updating or checking their position collateralization properly. Although NToken implements transferable and non-transferable assets functionality, the LendingPool contract, as owner, bypasses these checks. This flaw permits users to bypass position tracking, creating a discrepancy between protocol accounting and actual token balances.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 1109-1139
**Function:** redeem() and _redeem()

**Remediation:**

Redeem Function Accounting Bypass vulnerability can be mitigated through the following remediations:

- Owner-Only Redemption: Restrict the redeem function so only the contract owner can execute it, preventing users from bypassing collateral locks.
- Position Tracking: Update the redeem logic to check and update user positions, ensuring that only transferable (unlocked) NTokens can be redeemed.
- Unredeemable for Active Positions: Implement checks that prevent redemption for users with active lending positions or outstanding borrows, enforcing proper collateralization.

**Impact:**

Redeem Function Accounting Bypass can impact protocols in multiple ways, including accounting mismatches, over-borrowing, liquidation failures, and potential protocol drain.

- **Accounting Mismatch:** Users will be able to withdraw assets while the protocol still considers their collateral locked,

leading to inaccurate accounting.

- **Over-Borrowing:** Users may borrow more than their actual collateral allows, increasing the risk of protocol insolvency.
- **Liquidation Failures:** Health and liquidation calculations become unreliable, potentially preventing proper liquidations and risk management.
- **Protocol Drain:** Attackers can repeatedly exploit this flaw to drain protocol reserves, causing financial loss and undermining user trust.

**Code Snippet:**

```
    @arc4.abimethod
def redeem(self, market_id: arc4.UInt64, amount: arc4.UInt256) -> None:
ensure_budget(1400, OpUpFeeSource.GroupCredit)
self._redeem(market_id.native, amount.native)
@subroutine
def _redeem(self, market_id: UInt64, amount: BigUInt) -> None:
self._validate_market_operation(market_id, amount, True)
market = self._accrue_interest(market_id)

arc4.abi_call(
NToken.burn_from,
arc4.Address(Txn.sender),
arc4.UInt256(amount),
app_id=Application(market.ntoken_id.native),
)
scaled_amount = (amount * SCALE) // market.deposit_index.native
assert (
market.total_scaled_deposits.native >= scaled_amount
), "Insufficient market deposits to redeem"

self._decrement_market_total_scaled_deposits(market_id, scaled_amount)

arc4.abi_call(
ARC200TokenInterface.arc200_transfer,
arc4.Address(Txn.sender),
arc4.UInt256(amount),
app_id=Application(market_id),
)
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.4 Unrestricted Pause Function Access

| Severity | Status | Type of Analysis |
|---|---|---|
| ● Critical | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In Lending Protocols, a global pause mechanism is used to halt all protocol operations in emergencies or during maintenance to protect user funds and preserve system stability. Proper implementations restrict control over this functionality to authorized roles only.

Entersoft has observed that in the DorkFi project's LendingPool contract, the pause function is publicly accessible without any access control or role restriction. Any external user, including malicious actors, can call this function to pause or unpause the protocol arbitrarily. This absence of restricted access undermines the protocol's operational security and availability.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 404-410

**Remediation:**

Unrestricted Pause Function Access vulnerability can be mitigated by enforcing strict access controls on the pause function, ensuring only authorized roles can invoke it, and preventing malicious disruptions to protocol operations.

The pause function must be protected by strict access control, allowing only the contract owner or authorized roles to invoke it. This can be implemented by adding an _only_owner() check or equivalent authorization guard at the start of the pause method. Restricting this critical operation will prevent unauthorized users from disrupting the protocol.

**Impact:**

Unrestricted Pause Function Access can impact protocols in several ways, including denial of service, forced liquidations, revenue loss, reputational damage, and even potential permanent shutdown.

- **Denial of Service:** Any user can arbitrarily pause or unpause the protocol, disrupting all lending, borrowing, and liquidation operations.
- **Forced Liquidations and Losses:** Users may be unable to manage their positions, leading to forced liquidations and financial losses.
- **Loss of Protocol Revenue:** Pausing the protocol halts fee generation and revenue streams, impacting sustainability.

- **Reputational Damage:** Users will lose trust in the protocol's reliability and security, potentially leading to permanent loss of user base.
- **Potential for Permanent Shutdown:** If maliciously paused, the protocol may remain inaccessible until the owner intervenes, causing long-term damage.

**Code Snippet:**

```
    @arc4.abimethod
def pause(self, pause: arc4.UInt64) -> None:
self._pause(pause.native)
@subroutine
def _pause(self, pause: UInt64) -> None:
self.paused = pause
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.5 Missing SToken-Specific Liquidation Logic

| Severity | Status | Type of Analysis |
|---|---|---|
| High | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

DeFi lending protocols often involve distinct token types with unique behaviors requiring specialized liquidation processes. SToken, representing a borrow-only market with specific interest and reserve mechanisms, requires custom handling during liquidation to ensure accurate debt settlement and collateral seizure.

Entersoft has observed that in the DorkFi project's LendingPool contract, the _liquidate_cross_market function treats SToken debts identically to other market debts, ignoring SToken's unique characteristics. This omission causes SToken debts to not be properly liquidated, breaking liquidation flows and compromising debt recovery.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 1339
**Function:** _liquidate_cross_market

**Remediation:**

Missing SToken-Specific Liquidation Logic vulnerability can be mitigated by implementing SToken-specific liquidation logic and ensuring proper debt settlement that aligns with SToken's accounting model.

- Implement SToken-Specific Liquidation Logic: Update the _liquidate_cross_market function to detect when the debt market is the SToken market (market_id == self.stoken_app_id) and apply tailored liquidation rules.
- Ensure Proper Debt Settlement: Design the liquidation flow to correctly burn SToken, update reserves, and seize collateral in accordance with SToken's unique accounting.

**Impact:**

Missing SToken-Specific Liquidation Logic can impact protocols in multiple ways, including broken liquidation mechanisms, protocol insolvency, bad debt accumulation, and loss of user funds.

- **Broken Liquidation Mechanism:** SToken debts may not be liquidated, leaving unhealthy positions unresolved.
- **Protocol Insolvency:** Failure to recover SToken debt and seize collateral can lead to bad debt accumulation and threaten protocol solvency.

- **Bad Debt:** Unrecoverable SToken positions increase the risk of bad debt, undermining user trust and protocol stability.
- **Loss of User Funds:** Users may be unable to recover collateral or settle debts, leading to financial losses and reputational damage.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.6 Omission of Accrued Interest in Liquidation Calculations

| Severity | Status | Type of Analysis |
|---|---|---|
| High | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, accurate liquidation risk assessment depends on considering both principal borrow amounts and the accrued interest over time. Failure to factor in accrued interest leads to incorrect health factor calculations, allowing unhealthy positions to evade liquidation.

Entersoft has observed that in the DorkFi project's LendingPool contract, the liquidation health factor computation only considers the total collateral and borrow principal values but omits interest accrued on borrow positions. Consequently, borrowers with significant accrued interest may appear healthy and avoid liquidation, weakening the protocol's risk management.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 1369
**Function:** _liquidate_cross_market

**Remediation:**

Omission of Accrued Interest in Liquidation Calculations vulnerability can be mitigated by ensuring interest is always accrued and synchronized before liquidation checks, thereby maintaining accurate risk management and collateral recovery

- **Accrue Interest Before Liquidation:** Ensure that interest is always accrued and borrow values are updated before performing health factor calculations and liquidation checks.
- **Update User and Market State:** Call interest accrual routines for both the user and the market before checking liquidation eligibility.
- **Automated Interest Sync:** Consider implementing automatic interest synchronization for all users before any critical protocol operation.

**Impact:**

Omission of Accrued Interest in Liquidation Calculations can impact protocols in several ways, including creation of non-liquidatable positions, accumulation of bad debt, increased risk to protocol funds, and loss of user trust.

- **Non-Liquidatable Positions:** Borrowers with large interest accumulation may never be liquidated, even when their positions are underwater.
- **Bad Debt Accumulation:** The protocol may accumulate bad debt as unhealthy positions persist, threatening solvency.
- **Risk to Protocol Funds:** The protocol's reserves and user funds are at risk due to inaccurate risk management and the inability to recover collateral from delinquent borrowers.
- **Loss of Trust:** Users and integrators may lose confidence in the protocol's ability to manage risk and maintain solvency.

**Code Snippet:**

```
        health = (
    global_user.total_collateral_value.native
    * BigUInt(liquidation_threshold)
    * BigUInt(10000)
) // (global_user.total_borrow_value.native * BigUInt(10000))
assert health < 10000, "Position is not liquidatable"
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.7 Oracle Price Manipulation Attack

| Severity | Status | Type of Analysis |
|---|---|---|
| High | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

Price oracle manipulation represents one of the most severe attack vectors in DeFi lending protocols. An attacker with control over price updates can artificially inflate or deflate asset prices without any deviation limits, allowing them to trigger mass liquidations, extract collateral at favorable rates, or borrow against inflated collateral values. This type of vulnerability can lead to the complete drainage of protocol reserves, systematic liquidation of healthy positions, and total loss of user funds. The absence of price validation mechanisms, rate limiting, and deviation checks creates a direct pathway for attackers to extract unlimited value from the protocol.

Entersoft has observed that the _set_market_price() function within the LendingPool contract lacks comprehensive price validation controls. While the PriceOracle contract contains _validate_price_deviation() functionality, the LendingPool completely bypasses these protections when the owner directly calls set_market_price(). The contract accepts any positive price value without validating against the previous price, enforcing maximum percentage changes, implementing time-based rate limiting, or checking price freshness. This allows a malicious or compromised oracle operator to manipulate prices by any magnitude (e.g., setting ETH from $2000 to $500 instantly), enabling coordinated attacks to trigger cascading liquidations, extract liquidation bonuses on artificially-created positions, or borrow maximum amounts against inflated collateral values. The vulnerability is amplified across multiple markets, where an attacker can manipulate prices in coordinated sequences to drain protocol collateral systematically.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Function:** _set_market_price()

**Remediation:**

To mitigate Oracle Price Manipulation Attack vulnerability, implement comprehensive multi-layer price validation to prevent manipulation attacks:

- Add Price Deviation Limits
- Implement Time-Based Rate Limiting
- Implement Circuit Breaker System

- Enforce Multi-Signature Approval for Large Changes

**Impact:**

An Oracle Price Manipulation Attack vulnerability can have the following impact:

- **Flash Price Manipulation:** Set ETH from $2000 to $500, trigger liquidations at bonus rates, then revert the price
- **Liquidation Cascade:** Manipulate multiple market prices simultaneously to create correlated liquidation events
- **Over-Borrowing:** Inflate collateral prices to allow borrowing far beyond actual collateral value
- **MEV Exploitation:** Front-run price updates to liquidate positions at advantageous rates

**Code Snippet:**

```
    @subroutine
def _set_market_price(self, token_id: UInt64, price: BigUInt) -> None:
assert not self._is_paused(), "Protocol is paused"
assert token_id in self.markets, "Market does not exist"
# assert token_id not in self.price_feeds, "Price feed already attached"
assert price > 0, "Price must be positive"
market = self.markets[token_id].copy()
# Unpause the market on first price set
if market.paused and market.price == 0:
market.paused = arc4.Bool(False)
# Scale the price by SCALE internally for storage
market.price = arc4.UInt256(price * SCALE)
self.markets[token_id] = market.copy()
# arc4.emit(MarketPriceSet(arc4.UInt64(token_id), arc4.UInt256(price)))
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.8 Oracle Price Staleness

| Severity | Status | Type of Analysis |
|---|---|---|
| ● High | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

Lending protocols depend on current, accurate price data to make critical decisions regarding collateral valuation, liquidation eligibility, and borrowing limits. When price feeds become stale, outdated by minutes or hours, the protocol operates on incorrect information, enabling risk-free arbitrage, incorrect liquidations, and market manipulation. Stale prices create a temporal window where attackers can exploit discrepancies between real market prices and protocol-recorded prices, borrowing against inflated values or avoiding legitimate liquidations through price lag. This vulnerability is particularly severe during periods of high volatility or oracle downtime when price divergence can exceed 10-20% before updates occur.

Entersoft has observed that the LendingPool contract's _set_market_price() and _fetch_price_feed() functions completely lack timestamp validation and price freshness checks. While the PriceOracle correctly stores price timestamps via PriceWithTimestamp, the LendingPool ignores these timestamps and uses stale prices without validation. Users can deposit collateral at inflated (stale) prices, borrow maximum amounts, and realize profits when prices eventually update. Additionally, the contract stores no last_update_time in MarketData, preventing any staleness validation. Market prices can be hours old without triggering any warnings or blockers. During oracle downtime or network delays, the protocol continues operating with outdated prices, potentially accumulating bad debt that becomes apparent only after the next price update, when positions suddenly become undercollateralized.

**Locations:**

- **File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

To mitigate Oracle Price Staleness, implement comprehensive price freshness validation with multiple layers of protection:

- Add Timestamp Tracking to MarketData
- Validate Price Freshness Before Usage
- Update _set_market_price with Freshness Check
- Implement Time-Weighted Average Price (TWAP)

**Impact:**

Oracle Price Staleness can have the following impact:

- **Stale Price Arbitrage:** Deposit at $2000 price (real: $1800), borrow maximum, realize $200/asset profit
- **Oracle Downtime Cascade:** Price stale 1 hour, market crashes 15%, protocol unaware until update
- **Flash Crash Exploitation:** 4.5-minute window between real crash and oracle update for arbitrage
- **Liquidation Bypass:** User maintains "appears liquidatable" state during staleness, gaining time

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.9 Lack of Low Utilization Safeguards

| Severity | Status | Type of Analysis |
| --- | --- | --- |
| Medium | Identified | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, the utilization rate, the ratio of borrowed assets to supplied assets, is crucial for capital efficiency and attractiveness to lenders. Without explicit safeguards or incentives to manage low utilization periods, protocols risk idle capital and reduced yields, impairing overall performance.

Entersoft has observed that in the DorkFi project's LendingPool contract, dynamic interest rate adjustments exist, but no other mechanisms address low utilization scenarios. This absence results in inefficient capital usage, minimal returns for lenders, and diminished protocol competitiveness compared to platforms with active utilization management.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

The lack of Low Utilization Safeguards vulnerability can be mitigated by introducing utilization safeguards such as borrower incentives, dynamic parameters, alternative yield strategies, and enhanced transparency for users.

- Implement additional incentives for borrowers during low utilization periods, such as temporary interest rate reductions or rewards.
- Restrict additional deposits when utilization is low.
- Introduce alternative yield strategies, such as integrating with external protocols to deploy idle assets.
- Set minimum utilization thresholds and dynamically adjust protocol parameters to encourage activity.
- Provide transparency and analytics to users about utilization rates and yield optimization.

**Impact:**

Lack of Low Utilization Safeguards can impact protocols in several ways, including inefficient capital allocation, lower yields for lenders, and reduced competitiveness.

- **Inefficient Capital Allocation:** Large amounts of capital may remain unused, reducing overall protocol efficiency.
- **Lower Yields for Lenders:** Lenders may earn less interest, making the protocol less attractive and potentially leading to capital outflows.

- **Reduced Competitiveness:** The protocol may lose market share to platforms that actively optimize for utilization and yield.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.10 Missing Minter Approval Check for SToken

| Severity | Status | Type of Analysis |
|---|---|---|
| Medium | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi token ecosystems, minting and burning permissions are typically managed via minter approvals to ensure only authorized contracts can modify token supply. General best practices require verifying minter approval before initiating mint or burn actions to prevent operational failures.

Entersoft has observed that in the DorkFi project's LendingPool contract, the set_stoken_app_id function does not verify if LendingPool is authorized as a minter on the SToken contract. Since SToken restricts minting and burning to approved minters only, missing this approval check risks transaction failures during borrowing and repayment involving SToken.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 435-438
**Function:** set_stoken_app_id

**Remediation:**

Missing Minter Approval Check for SToken can be mitigated by adding a validation in the set_stoken_app_id function to ensure the LendingPool contract is approved as a minter before assigning the app ID.

**Impact:**

Missing Minter Approval Check for SToken can impact protocols in several ways, including SToken market failure, protocol downtime, loss of user trust, and increased operational risk.

- **SToken Market Failure:** All minting and burning operations for SToken will fail, breaking borrowing and repayment for the SToken market.
- **Protocol Downtime:** Users will be unable to interact with the SToken market, leading to loss of functionality and potential user funds locked in the protocol.
- **Loss of Trust:** Users may lose confidence in the protocol's reliability and security due to unexpected failures.
- **Operational Risk:** The protocol may require emergency upgrades or interventions to restore SToken market functionality.

**Code Snippet:**

```
    @arc4.abimethod

def set_stoken_app_id(self, stoken_app_id: arc4.UInt64) -> None:

self._only_owner()

self.stoken_app_id = stoken_app_id.native
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.11 Missing Transaction and User Caps (Unbounded Operations)

| Severity | Status | Type of Analysis |
| --- | --- | --- |
| Medium | Identified | Dynamic |

**Contract Name:**

LendingPool

**Description:**

DeFi protocols typically enforce upper limits per transaction and per user to maintain operational stability, prevent liquidity draining, and avoid market dominance by single entities. Without such constraints, malicious users can perform excessively large operations, destabilizing the market and risking protocol insolvency.

Entersoft has observed that in the DorkFi project's LendingPool contract, no caps exist on amounts deposited, withdrawn, borrowed, liquidated, or redeemed in a single transaction. Moreover, there are no per-user caps on total deposit or borrow amounts. This leaves the protocol vulnerable to liquidity drains, market manipulation, and increased attack surface.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

Missing Transaction and User Caps (Unbounded Operations) can be mitigated by implementing per-transaction and per-user limits, adding dynamic cap adjustments, and monitoring for unusually large operations.

- Implement Per-Transaction Caps: Set reasonable maximum limits for withdraw, redeem, and liquidate operations to prevent draining liquidity in a single transaction.
- Add Per-User Caps: Restrict the total amount each user can deposit or borrow, reducing the risk of market domination and concentration.
- Dynamic Cap Adjustment: Consider adjusting caps based on market conditions, protocol liquidity, or governance decisions.
- Monitor and Alert: Add monitoring and alerting for unusually large transactions to enable rapid response to potential attacks.

**Impact:**

Missing transactions and User Caps (Unbounded Operations) can impact protocols in several ways, including market domination, liquidity drain, interest rate manipulation, and an increased attack surface.

- **Market Domination:** A single user or entity can control a disproportionate share of deposits or borrows, undermining

decentralization and fairness.

- **Liquidity Drain:** Large withdrawals, redemptions, or liquidations can deplete protocol reserves in a single transaction, causing operational disruptions and user losses.
- **Rate Manipulation:** Sudden changes in utilization from large transactions can distort interest rates and market dynamics, impacting all users.
- **Increased Attack Surface:** The protocol becomes more vulnerable to flash loan attacks and other forms of economic manipulation.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.12 Missing Liquidation Threshold vs Collateral Factor Check

| Severity | Status | Type of Analysis |
| --- | --- | --- |
| Low | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, the collateral_factor and liquidation_threshold parameters must maintain logical consistency to ensure fair risk management. The collateral_factor defines the required collateral ratio for borrowing, while the liquidation_threshold determines when positions become eligible for liquidation. The liquidation_threshold should be equal to or higher than the collateral_factor to prevent unfair liquidations.

Entersoft has observed that in the DorkFi project's LendingPool contract, the create_market function lacks validation to enforce liquidation_threshold >= collateral_factor. Consequently, markets can be deployed with illogical parameters where users might be liquidated despite having adequate collateral.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 546-588
**Function:** create_market

**Remediation:**

Missing Liquidation Threshold vs Collateral Factor Check can be mitigated by enforcing a check in the create_market function to ensure liquidation thresholds are always greater than or equal to collateral factors.

- Add an assertion in the create_market function to ensure that liquidation_threshold >= collateral_factor before creating a new market.
- This simple check enforces logical consistency between collateral requirements and liquidation rules, protecting users from unfair liquidations.

**Impact:**

Missing Liquidation Threshold vs Collateral Factor Check can impact protocols by enabling unfair liquidations and eroding user trust in the protocol's fairness and risk management.

- **Unfair Liquidations:** Users may be liquidated even when their collateral is sufficient, resulting in unexpected losses.
- **Loss of Trust:** Users may lose confidence in the protocol's fairness and risk management, leading to reduced adoption.

**Code Snippet:**

```
    @arc4.abimethod
def create_market(
self,
token_id: arc4.UInt64,
collateral_factor: arc4.UInt64,
liquidation_threshold: arc4.UInt64,
reserve_factor: arc4.UInt64,
borrow_rate: arc4.UInt64,
slope: arc4.UInt64,
max_total_deposits: arc4.UInt256,
max_total_borrows: arc4.UInt256,
liquidation_bonus: arc4.UInt64,
close_factor: arc4.UInt64,
) -> arc4.UInt64:
self._only_owner()
assert not self._is_paused(), "Protocol is paused"
return arc4.UInt64(
self._create_market(
token_id.native,
MarketData(
paused=arc4.Bool(True), # Start paused until price is set
max_total_deposits=max_total_deposits,
max_total_borrows=max_total_borrows,
liquidation_bonus=liquidation_bonus,
collateral_factor=collateral_factor,
liquidation_threshold=liquidation_threshold,
reserve_factor=reserve_factor,
borrow_rate=borrow_rate,
```

```
slope=slope,
total_scaled_deposits=arc4.UInt256(0),
total_scaled_borrows=arc4.UInt256(0),
deposit_index=arc4.UInt256(SCALE), # Start at SCALE (1.0)
borrow_index=arc4.UInt256(SCALE), # Start at SCALE (1.0)
last_update_time=arc4.UInt64(Global.latest_timestamp),
reserves=arc4.UInt256(0),
price=arc4.UInt256(
0
), # Price must be set before market can be used
ntoken_id=arc4.UInt64(0), # Will be set during token initialization
close_factor=close_factor,
),
)
)
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.13 Same-Asset Borrowing Problem

| Severity | Status | Type of Analysis |
|---|---|---|
| Low | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In lending protocols, borrowing the same asset as supplied collateral typically offers no financial benefit and exposes users to unnecessary fees and risks. Best practices advise restricting same-asset borrowing to prevent inefficient and risk-irrelevant positions that degrade user experience.

Entersoft has observed that in the DorkFi project's LendingPool contract, users can borrow from the same market where they supply collateral, creating price-invariant positions with no leverage advantages. This may confuse users and automated strategies, resulting in wasted capital and increased liquidation risks without protocol-level harm.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

The Same-Asset Borrowing Problem can be mitigated by enforcing validation checks in the borrow and deposit logic, and by providing user-facing warnings when such actions are attempted

- Enforce a Check: Add a validation in the borrow and deposit logic to prevent users from borrowing against collateral in the same market.
- User-Facing Warnings: Optionally, provide clear warnings or error messages to users attempting same-asset borrowing, explaining why it is disallowed.

**Impact:**

The Same-Asset Borrowing Problem can impact users through unnecessary losses, poor user experience, and reduced trust, even though the protocol itself faces no direct financial risk.

- **User Losses:** Users may unknowingly enter positions that only generate fees and interest without any benefit.
- **Poor User Experience:** Automated flows or bots may create inefficient positions, leading to confusion and dissatisfaction.
- **No Protocol Risk:** The protocol itself is not directly harmed, but user trust and satisfaction may be negatively impacted.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.14 Self Liquidation

| Severity | Status | Type of Analysis |
|---|---|---|
| Low | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, liquidation functions are intended to allow third-party liquidators to repay undercollateralized borrowers and seize collateral. Allowing users to liquidate their own positions generally causes no protocol harm but leads to unnecessary penalties and loss for the user when self-liquidating.

Entersoft has observed that the DorkFi project's LendingPool contract currently permits self-liquidation by allowing users to call the liquidation function targeting themselves. This can cause users to incur liquidation bonuses, fees, or collateral loss unnecessarily, negatively affecting user experience and wallet balances without any benefit to the protocol.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py

**Remediation:**

Self Liquidation vulnerability can be mitigated by restricting liquidation logic to disallow self-liquidation and by providing user-facing warnings when such attempts occur.

- Restrict Self-Liquidation: Add a check in the liquidation logic to prevent users from liquidating their own positions. The protocol should assert that the liquidator and the target user are not the same.
- User-Facing Warnings: Optionally, provide clear error messages to users attempting self-liquidation, explaining why it is disallowed.

**Impact:**

Self Liquidation vulnerability can impact users through unnecessary losses, poor user experience, and reduced trust, even though the protocol itself faces no direct financial risk

- **User Losses:** Users may unknowingly liquidate their own positions, incurring unnecessary penalties and loss of collateral.
- **Poor User Experience:** Automated flows or bots may trigger self-liquidation, leading to confusion and dissatisfaction.
- **No Protocol Risk:** The protocol itself is not directly harmed, but user trust and satisfaction may be negatively impacted.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.15 Liquidate Slippage Protection

| Severity | Status | Type of Analysis |
|---|---|---|
| Informational | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, slippage protection during liquidation safeguards liquidators from receiving less collateral than expected due to rapid price fluctuations between transaction initiation and execution. Typically, liquidators specify a minimum collateral amount they expect to receive, and transactions revert if this threshold is unmet.

Entersoft has observed that in the DorkFi project's LendingPool contract, the _liquidate_cross_market function lacks a min_collateral_received parameter or equivalent mechanism. Consequently, liquidators are exposed to collateral value slippage, potentially causing unexpected losses during volatile market conditions.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 1339
**Function:** _liquidate_cross_market

**Remediation:**

Liquidate Slippage Protection can be implemented by allowing liquidators to specify a minimum collateral parameter in liquidation calls, ensuring fair execution without affecting protocol solvency.

- Add Slippage Protection: Allow liquidators to specify a min_collateral_received parameter when calling the liquidation function. The protocol should assert that the actual collateral transferred meets or exceeds this minimum.
- Optional Implementation: This is an optional feature for user protection and does not impact protocol security or solvency.

**Impact:**

Liquidate Slippage Protection primarily impacts liquidators by exposing them to unexpected collateral shortfalls, which can discourage participation and reduce market efficiency, though it does not pose direct protocol risk

- **Liquidator Risk:** Liquidators may receive less collateral than expected due to price slippage, discouraging participation.
- **No Direct Protocol Risk:** The absence of this feature does not directly harm protocol solvency or user funds, but it may affect user experience and market efficiency.

**Code Snippet:**

NA

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.16 Liquidation threshold calculation

| Severity | Status | Type of Analysis |
|---|---|---|
| Informational | Closed | Dynamic |

**Contract Name:**

LendingPool

**Description:**

In DeFi lending protocols, liquidation thresholds govern when positions become eligible for liquidation, typically based on the risk profile of the collateral asset being seized. Industry best practices recommend using the collateral market's liquidation threshold rather than combining thresholds.

Entersoft has observed that in the DorkFi project's LendingPool contract, the _liquidate_cross_market function determines the liquidation threshold as the minimum of the debt market's and the collateral market's thresholds. This method can cause inconsistent or overly conservative liquidation conditions, resulting in unexpected liquidation timings that do not accurately reflect collateral risk.

**Locations:**

**File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
**Lines:** 1365
**Function:** _liquidate_cross_market

**Remediation:**

Liquidation Threshold Calculation can be corrected by using only the collateral market's liquidation threshold, aligning liquidation logic with industry norms, and improving consistency.

Use Collateral Market Liquidation Threshold: Replace the calculation to use only the collateral market's liquidation threshold. This ensures that the risk and liquidation logic are based on the asset being seized, which is the industry norm.

**Impact:**

Liquidation Threshold Calculation vulnerability can impact protocols by creating inconsistent liquidation behavior and user confusion, though they do not introduce direct protocol solvency risk.

- **Inconsistent Liquidation Behavior:** Positions may be liquidated based on thresholds that do not match the risk profile of the collateral, leading to unpredictable outcomes.
- **User Confusion:** Users may not understand why their positions are liquidated or protected, reducing protocol transparency.

- **No Direct Protocol Risk:** This is primarily a user experience and consistency issue, not a direct security or solvency risk.

**Code Snippet:**

```
        liquidation_threshold = self._min64(
    debt_market.liquidation_threshold.native,
    collateral_market.liquidation_threshold.native,
)
```

**Reference:**

**Proof of Vulnerability:**

N.A.

## 4.17 Sanity Checks

| Severity | Status | Type of Analysis |
|---|---|---|
| Informational | Identified | Dynamic |

**Contract Name:**

LendingPool

**Description:**

Market creation in lending protocols requires strict validation of all economic parameters to ensure the protocol operates within safe bounds and prevents configuration errors that could lead to liquidation failures, extreme interest rates, or accounting inconsistencies. Sanity checks on collateral factors, liquidation thresholds, reserve factors, close factors, and interest rate parameters are standard industry practice to catch misconfigurations before they cause harm. Without these checks, administrative errors or intentional misuse can result in markets with impossible parameter combinations (e.g., liquidation threshold < collateral factor, close factor = 0, interest rates exceeding 1000% APR, or negative reserve factors).

Entersoft has observed that the create_market() function in the LendingPool contract lacks comprehensive parameter validation. While a minimal check exists for liquidation_threshold > collateral_factor, the function is missing critical sanity checks on close factor (allowing 0), reserve factor (allowing values > 100%), liquidation bonus (allowing > 100%), interest rates (allowing extreme APR values), and supply/borrow caps. The function accepts close_factor as UInt64(0), which would make liquidations impossible. Reserve factor can exceed 10000 basis points (100%), causing incorrect interest distribution calculations. Liquidation bonus can exceed reasonable bounds, resulting in liquidators receiving more collateral than the debt repaid. Interest rates (borrow_rate + slope) can accumulate to thousands of percent APR without validation. Supply and borrow caps can be set to 0, immediately freezing market operations. These missing checks allow administrators—whether through honest mistakes or malicious intent—to create broken markets that cannot function correctly.

**Locations:**

- **File:** https://github.com/DorkFi/os-dorkfi/blob/beta/src/contract.py
  - **Function:** _create_market()

**Remediation:**

Implement comprehensive sanity checks for all market parameters:

- Collateral Factor: Must be > 0 and ≤ 10000 (≤ 100%)
- Liquidation Threshold: Must be > 0, ≤ 10000, and > Collateral Factor
- LT-CF Gap: Must be ≥ 500 (≥ 5%) for safe liquidations
- Close Factor: Must be > 0 and ≤ 10000 (≤ 100%), and ≤ Liquidation Threshold
- Reserve Factor: Must be ≥ 0 and ≤ 10000 (≤ 100%)

- Liquidation Bonus: Must be > 0 and ≤ 5000 (≤ 50%)
- Borrow Rate: Must be ≥ 0 and ≤ 50000 (≤ 500% APR)
- Slope: Must be ≥ 0 and ≤ 100000 (≤ 1000%)
- Max Interest Rate: (Borrow Rate + Slope) ≤ 200000 (≤ 2000% APR)

**Impact:**

Lack of Sanity Checks can have the following impact:

- Close factor = 0: Liquidations become impossible (liquidators can't repay debt)
- Reserve factor = 15000: 150% of interest claimed as reserves (negative return to depositors)
- Close factor = 100%: All debt must be liquidated (unrealistic for small debts)
- Max deposits = 0: Market frozen immediately upon creation
- Interest rate = 1000000 bps: 10,000% APR (economically nonsensical)

**Code Snippet:**

```
    @subroutine
def _create_market(self, token_id: UInt64, market_data: MarketData) -> UInt64:
assert token_id not in self.markets, "Market already exists"
payment = require_payment(Txn.sender)
assert payment >= 2000000, "insufficient payment for create market"
compiled = compile_contract(NToken, extra_program_pages=0) # no extra pages
base_app = arc4.arc4_create(NToken, compiled=compiled).created_app
itxn.Payment(
receiver=base_app.address, amount=1000000, fee=0 # min 108900 sends 1000000
).submit()
arc4.abi_call(
NToken.bootstrap64,
arc4.UInt64(token_id),
app_id=base_app,
)
market_data.ntoken_id = arc4.UInt64(base_app.id)
self.markets[token_id] = market_data.copy()
arc4.emit(MarketCreated(arc4.UInt64(token_id)))
return base_app.id
```

**Reference:**

**Proof of Vulnerability:**

N.A.

# 5.0 Auditing Approach and Methodologies applied

Throughout the audit of the smart contract, care was taken to ensure:

- Overall quality of code
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Mathematical calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of token standards.
- Efficient use of gas.
- Code is safe from Re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

## 5.1 Structural Analysis

In this step we have analysed the design patterns and structure of all smart contracts. A thorough check was completed to ensure all Smart contracts are structured in a way that will not result in future problems.

## 5.2 Static Analysis

Static Analysis of smart contracts was undertaken to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

## 5.3 Code Review / Manual Analysis

Manual Analysis or review of done to identify new vulnerabilities or to verify the vulnerabilities found during the Static Analysis. The contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. It should also be noted that the results of the automated analysis were verified manually.

## 5.4 Gas Consumption

In this step, we checked the behaviour of all smart contracts in production. Checks were completed to understand how much gas gets consumed, along with the possibilities of optimisation of code to reduce gas consumption.

## 5.5 Tools & Platforms Used For Audit

Tealer, Chai, Mocha, Algokit

# 6.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of DorkFi and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Smart Contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of DorkFi and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and DorkFi governs the disclosure of this report to all other parties including product vendors and suppliers.