BLOCKAPEX

SMART CONTRACT SECURITY

v 1.0

Date: 08-08-2025

Prepared For: Metapool Near

About BlockApex

Founded in early 2021 BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

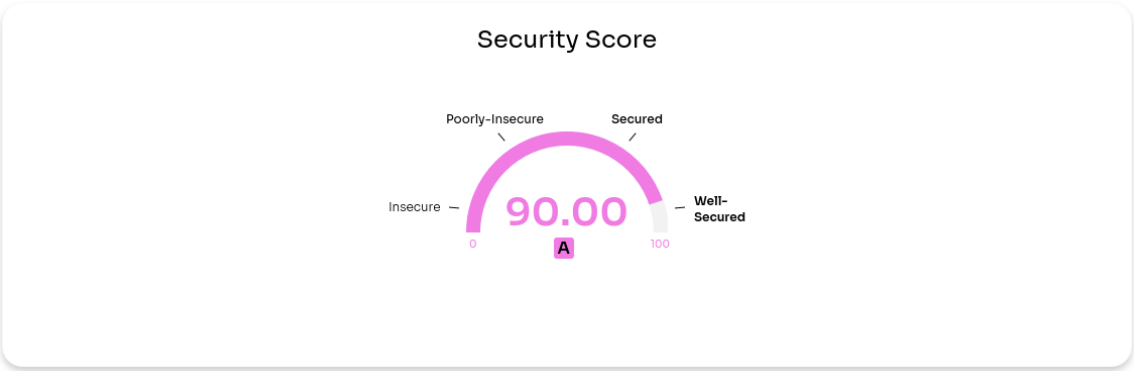
1 Executive Summary **4**
1.1 Recommendation . . . 6
1.2 Scope . . . 7
1.2.1 In Scope . . . 7
1.2.2 Out of Scope . . . 7
1.3 Methodology . . . 7
1.4 Status Descriptions . . . 8
1.5 Summary of Findings Identified . . . 9

2 Findings and Risk Analysis **11**
2.1 Timestamp Calculation Error Causes Catastrophic Interest Accrual Failure . . . 11
2.2 Repay Function Fails to Burn VERDE Tokens Enabling Free Debt Elimination . . . 13
2.3 Collateral contamination risk due to shared debt accounting and flat system parameters 14
2.4 Complete Stable Token Accounting System Failure leading to DOS scenarios . . . 15
2.5 No Incentive for Liquidator Above 90% LTV leading to Bad Debt . . . 19
2.6 USDMP (Verde) peg of 1$ can create Protocol Risks . . . 21
2.7 Flawed Interest Calculation leads to unfair interest charge and gaming . . . 22
2.8 Protocol Absorbs All Storage Costs Instead of Users . . . 24
2.9 Deposit Tokens Whitelisted for Both Collateral and Swap Causes Swap Path to Become Unreachable . . . 26
2.10 Interest Accrual Bypass Enables Interest-Free Loans and Potential DOS . . . 27
2.11 Inverted Swap Fee Validation Logic Allows Unlimited Fees . . . 28
2.12 Missing Price Staleness Validation Enables Stale Oracle Data Usage . . . 29
2.13 liquidation can be done if the protocol is paused . . . 31
2.14 Capital-intensive liquidation mechanism increases risk of insolvency . . . 32
2.15 Missing Pause Controls on Debt Repayment Function . . . 33
2.16 Compound Interest Implementation Overcharge Users . . . 34
2.17 Single Oracle Dependency Creates Manipulation Risk and Protocol Failure Points . . . 35
2.18 Weak Input Validation Allows Invalid Swap Format Values . . . 36

1 Executive Summary

A team of 2 Auditors performed the Filtered Audit,where the codebase was audited individually by both auditors. After a thorough and rigorous manual testing process involving line by line code review for bugs related to Rust and Near Smart Contracts. All the raised flags were manually reviewed and tested to identify any false positives.

The Audit resulted in 4 Critical , 6 Highs , 5 medium , 3 low and 1 Info Severity Issues. The codebase received an overall security score of 90 out of 100 points. Ten points were deducted due to the absence of comprehensive fuzzing testing and a complete test suite, which are recommended best practices for production-ready smart contracts.


Security Score
Poorly-Insecure
Secured
Insecure
90.00
Well-
Secured
0
A
100

1.1 Recommendation

To enhance the security posture the development team should prioritize the following improvements:

1. **Implement Comprehensive Fuzzing Testing** - Design a Fuzzing suite to identify edge cases and potential vulnerabilities that may not be caught through manual testing alone.
2. **Develop Complete Test Suite** - Create extensive unit tests, integration tests, and end-to-end tests that cover all contract functions and potential user interactions.

1.2 Scope

1.2.1 In Scope

Repository - https://github.com/Meta-Pool/stable-verde-near/releases/tag/v0.1.0

Commit Hash - 2397bff46c4d1c741b3750079c4752f575db5da0

1.2.2 Out of Scope

Anything not mentioned in the Scope.

1.3 Methodology

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning 1.5 Weeks. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations and security design patterns, code styles and best practices.

1.4 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.5 Summary of Findings Identified

S.NO	SEVERITY	FINDINGS	STATUS
#1	CRITICAL	Timestamp Calculation Error Causes Catastrophic Interest Accrual Failure	FIXED
#2	CRITICAL	Repay Function Fails to Burn VERDE Tokens Enabling Free Debt Elimination	FIXED
#3	CRITICAL	Collateral contamination risk due to shared debt accounting and flat system parameters	FIXED
#4	CRITICAL	Complete Stable Token Accounting System Failure leading to DOS scenarios	FIXED
#5	HIGH	No Incentive for Liquidator Above 90% LTV leading to Bad Debt	FIXED
#6	HIGH	USDMP (Verde) peg of 1$ can create Protocol Risks	ACKNOWLEDGED
#7	HIGH	Flawed Interest Calculation leads to unfair interest charge and gaming	ACKNOWLEDGED
#8	HIGH	Protocol Absorbs All Storage Costs Instead of Users	OPEN
#9	HIGH	Deposit Tokens Whitelisted for Both Collateral and Swap Causes Swap Path to Become Unreachable	FIXED
#10	HIGH	Interest Accrual Bypass Enables Interest-Free Loans and Potential DOS	ACKNOWLEDGED
#11	MEDIUM	Inverted Swap Fee Validation Logic Allows Unlimited Fees	FIXED
#12	MEDIUM	Missing Price Staleness Validation Enables Stale Oracle Data Usage	OPEN

S.NO	SEVERITY	FINDINGS	STATUS
#13	MEDIUM	liquidation can be done if the protocol is paused	FIXED
#14	MEDIUM	Capital-intensive liquidation mechanism increases risk of insolvency	ACKNOWLEDGED
#15	MEDIUM	Missing Pause Controls on Debt Repayment Function	FIXED
#16	LOW	Compound Interest Implementation Overcharge Users	ACKNOWLEDGED
#17	LOW	Single Oracle Dependency Creates Manipulation Risk and Protocol Failure Points	ACKNOWLEDGED
#18	LOW	Weak Input Validation Allows Invalid Swap Format Values	FIXED

2 Findings and Risk Analysis

2.1 Timestamp Calculation Error Causes Catastrophic Interest Accrual Failure

Severity: Critical

Status: Fixed

Location:

- src/utils.rs
- src/internal.rs

Description: The ‘get_unix_day()“ function contains a critical mathematical error in timestamp conversion that breaks the entire interest accrual mechanism. The function incorrectly divides milliseconds by a seconds-based divisor, causing day calculations to be 1000x smaller than expected.

Root Cause:

`get_current_epoch_millis()` returns milliseconds, but SECONDS_PER_DAY (86,400) is designed for seconds. The division should use 86_400_000 (milliseconds per day) instead of 86_400 (seconds per day).

Current (Broken) Calculation:
env::block_timestamp() = nanoseconds
get_current_epoch_millis() = nanoseconds / 1,000,000 = milliseconds
get_unix_day() = milliseconds / 86,400 = milliseconds / seconds

Expected vs Actual Day Count:
Expected: Day 19,735 (current epoch = January 2024)
Actual: Day = 19.7 (returns fractional days as integers)
Error Factor: 1000x smaller than expected

Proof of Concept: Current broken calculation (approximate values for January 2025):
env::block_timestamp() = 1,735,689,600,000,000,000 ns (Jan 1, 2025)
get_current_epoch_millis() = 1,735,689,600,000 ms
get_unix_day() = 1,735,689,600,000 / 86,400 = 20,088,000 days

Correct calculation should be: get_unix_day() = 1,735,689,600,000 / 86,400,000 = 20,088 days

Realistic timeline:
Day 1 init: last_accrual_day = 20,088,000

Day 2: current unix_day = 20,089,000
time_elapsed_days = 1,000 (should be 1)

With 10% APR (2740 per second rate):
Expected daily interest = borrow_rate_per_second * 1 = 2740
ACTUAL daily interest = borrow_rate_per_second * 1000 = 2,740,000

Result: 1000x interest explosion

Impact:

- Hyperinflated Interest: 1000x intended daily rate
- Instant Liquidations: Debt explodes within hours
- Protocol Insolvency: Impossible debt levels destroy reserves

Recommendation: It is recommended that the calculation for `get_unix_day` be fixed as follows. Near default block_timestamp is in ns , we should get the in ms which are converted to seconds and then divided by 86400.

```
pub(crate) fn get_current_epoch_millis() -&gt; EpochMillis {
   env::block_timestamp_ms() / 1_000_000
}
```

2.2 Repay Function Fails to Burn VERDE Tokens Enabling Free Debt Elimination

Severity: Critical

Status: Fixed

Location:

- src/utils.rs
- src/internal.rs

Description: The repay() function contains a critical bug where it updates all internal debt accounting states but fails to burn the VERDE tokens provided by the user. This allows users to "repay" their debts without actually losing any VERDE tokens, effectively eliminating debt for free while retaining the ability to withdraw their collateral.

Affected Functions:

- repay() in src/lib.rs:296-302
- internal_repay() in src/internal.rs:153-179
- internal_repay_calculations() in src/internal.rs:189-238

Unlike liquidation and swap functions, the repay flow never calls `burn_verde_for()`.

Proof of Concept: Attack Scenario:

- Alice borrows 1000 VERDE against 200 NEAR collateral
- Alice calls repay(alice, collateral_id, 1000 VERDE)

Result:

- Alice's debt becomes 0 (states updated)
- Alice can withdraw her 200 NEAR collateral
- Alice still has 1000 VERDE in her account (never burned)

Profit: Alice gained 1000 VERDE for free

Impact: The Missing functionality to burn the Verde tokens means Attacker can take away borrowed amount + the collateral from the protocol essentially totally solvency for the protocol.

Recommendation: It is recommended that when the repayment is done , verde tokens are burnt via `burn_verde_for()`.

2.3 Collateral contamination risk due to shared debt accounting and flat system parameters

Severity: Critical

Status: Fixed

Location:

- src/deposit.rs:18-26

Description: Because all collateral types share a single interest rate, same LTV thresholds, and are accounted for in a single credit share pool, risk from one asset type bleeds into and contaminates the health of the entire system. If a single collateral asset is weak, volatile, or exploited, all borrowers (even those using high-quality collateral) are exposed.

Impact:

No Risk Differentiation: All assets, regardless of volatility or liquidity, are treated equally. This means highly volatile or illiquid assets (e.g., MEME tokens) can be used to borrow as much as stable, liquid assets (e.g., USDC), even though their risk profiles are vastly different.

Increased Systemic Risk: If a volatile asset's price collapses, the system may not recover enough value during liquidation to cover outstanding loans, leading to bad debt. Since parameters are uniform, the system cannot compensate for this by requiring higher collateral or charging higher interest for riskier assets

Poor Risk Management: Modern DeFi platforms use asset-specific collateral factors and dynamic rates to manage risk. Uniform parameters ignore these best practices, making the system more vulnerable to manipulation and cascading liquidations during market stress

Recommendation: We recommend defining LTV, Interest, and other parameters per collateral type for more comprehensive risk management.

2.4 Complete Stable Token Accounting System Failure leading to DOS scenarios

Severity: Critical

Status: Fixed

Location:

src/swaps.rs

Description: The protocol's stable token accounting system contains two critical bugs that completely corrupt financial tracking. Both add_stable_total_amount() and remove_stable_total_amount() incorrectly operate on in_transit_amount instead of total_amount, creating systematic accounting failures across all swap operations.

Bug 1: add_stable_total_amount

```
1  pub(crate) fn add_stable_total_amount(&amp;mut self, stable_id: &amp;AccountId,
       stable_amount: u128) {
2      let stable = self.get_mut_stable_asset(stable_id);
3      stable.in_transit_amount += stable_amount;  // @audit Should be: stable.total_amount
           += stable_amount
4  }
```

Bug 2: remove_stable_total_amount

```
1  pub(crate) fn remove_stable_total_amount(&amp;mut self, stable_id: &amp;AccountId,
       stable_amount: u128) {
2      let stable = self.get_mut_stable_asset(stable_id);
3      require!(stable.in_transit_amount &gt;= stable_amount, &quot;NotEnoughStableLiquidity&
           quot;);
4      stable.in_transit_amount -= stable_amount;  // @audit Should be: stable.total_amount
           -= stable_amount
5  }
```

Both functions have names suggesting they operate on total_amount but actually modify in_transit_amount, creating dangerous semantic inconsistency.

Complete Accounting System Breakdown Current (Broken) State:

- add_stable_total_amount() → increases in_transit_amount
- remove_stable_total_amount() → decreases in_transit_amount total_amount NEVER CHANGES despite tokens entering/leaving protocol

Expected (Correct) State:

- add_stable_total_amount() → increases total_amount
- remove_stable_total_amount() → decreases total_amount

`stable.total_amount` is NEVER updated in production code - it remains 0 forever after initialization.

Impact: The protocol's accounting is severely impacted and broken which can lead to DOS on legitimate operations.

Proof of Concept: POC 1: Capacity Control Bypass Leading to Infinite Deposits

Scenario:

Protocol configured with 100,000 USDC maximum capacity to prevent over-exposure. Initial State:

- Protocol reserves (total_amount): 0 USDC (never changes)
- Configured maximum capacity: 100,000 USDC
- Temporary in-transit funds: 0 USDC

Bypass Sequence:

1. User 1 deposits 99,000 USDC for Stable→Verde swap
    - Capacity check: “Is 100,000 < 0 + 99,000?”
    - Answer: NO, swap allowed
    - Protocol receives 99,000 USDC but total_amount remains 0
2. User 2 deposits another 99,000 USDC for Stable -> Verde swap
    - Capacity check: “Is 100,000 < 0 + 99,000?”
    - Answer: NO, swap allowed again
    - Protocol now holds 198,000 USDC but still shows total_amount = 0
3. Process repeats indefinitely

Result: Users can deposit unlimited amounts, completely bypassing all capacity controls. Protocol could accumulate millions while showing zero reserves.

POC 2: Progressive Accounting Corruption and Secondary Dos

Scenario:

Mixed protocol operations showing progressive system degradation.

Phase 1 - Stable→Verde Operations (Appear to Work):

1. User A: Deposits 50,000 USDC, receives VERDE tokens
    - Incorrect accounting: Adds 50,000 to temporary in-transit instead of reserves
    - State: total_amount = 0, in_transit_amount = 50,000
2. User B: Deposits 30,000 USDC, receives VERDE tokens
    - Incorrect accounting: Adds 30,000 to temporary in-transit
    - State: total_amount = 0, in_transit_amount = 80,000

Phase 2 - Verde→Stable Operations (Temporarily Work):

1. User C: Swaps VERDE for 25,000 USDC
    - Check: “Is 80,000 >= 25,000?”
    - Answer: YES, swap proceeds
    - Incorrect accounting: Removes 25,000 from in-transit instead of reserves
    - State: total_amount = 0, in_transit_amount = 55,000
2. User D: Swaps VERDE for 40,000 USDC
    - Check: “Is 55,000 >= 40,000?”
    - Answer: YES, swap proceeds
    - State: total_amount = 0, in_transit_amount = 15,000

Phase 3 - Secondary DoS Trigger:

1. User E: Attempts to swap VERDE for 20,000 USDC
    - Check: “Is 15,000 >= 20,000?”
    - Answer: NO
    - Transaction fails with “NotEnoughStableLiquidity”

Result: Permanent DoS - no more Verde→Stable swaps possible.

Reality Check: Protocol actually holds 55,000 USDC in real tokens (50,000 + 30,000 - 25,000) but accounting shows 0 reserves and 15,000 phantom in-transit.

POC 3: Legitimate Operations Blocked by Phantom Accounting

Scenario:

Normal protocol usage blocked by corrupted accounting.

Setup:

- Protocol holds 200,000 USDC in actual reserves
- Accounting shows: total_amount = 0, in_transit_amount = 30,000 (from previous ops)
- User has pending 15,000 USDC Stable→Verde swap (legitimately in-transit)

Failure Sequence:

- New user wants Verde→Stable swap for 40,000 USDC
- Protocol has 200,000 USDC available (more than enough)
- But an accounting check: “Is 30,000 >= 40,000?”
- Answer: NO
- Swap fails despite abundant real liquidity

Result: Legitimate operations blocked by phantom accounting.

Recommendation: It is recommended to apply fixes to the function and update the stable.total_amount correctly.

```
1   pub(crate) fn add_stable_total_amount(&amp;mut self, stable_id: &amp;AccountId,
        stable_amount: u128) {
2       let stable = self.get_mut_stable_asset(stable_id);
3       stable.total_amount += stable_amount;
4    }
5
6    pub(crate) fn remove_stable_total_amount(&amp;mut self, stable_id: &amp;AccountId,
        stable_amount: u128) {
7       let stable = self.get_mut_stable_asset(stable_id);
8       require!(stable.in_transit_amount &gt;= stable_amount, &quot;
            NotEnoughStableLiquidity&quot;);
9       // @audit Critical , this is wrong , it should be removed from the total amount.
10      stable.total_amount -= stable_amount;
11   }
```

2.5 No Incentive for Liquidator Above 90% LTV leading to Bad Debt

Severity: High

Status: Fixed

Location:

- src/lib.rs
- src/internal.rs

Description: The protocol's Tier 3 liquidation mechanism (LTV >= 90%) creates a fundamental economic disincentive for liquidators when positions exceed 90% LTV. Liquidators are required to pay the full debt amount but only receive collateral worth less than the debt, resulting in guaranteed losses that prevent liquidations and accumulate bad debt.

```
pub(crate) fn internal_get_liquidation_deal(
        &amp;amp;self,
        debt: u128,
        collateral: u128,
        collat_price_usd: u128
    ) -&amp;gt; (u128, u128) {
        let required_verde;
        let collateral_out;

        let real_ltv_bp = self.internal_get_loan_to_value_ratio_bp(
            debt,
            collateral,
            collat_price_usd
        );
        log!(&amp;quot;before liquidation LTV={}&amp;quot;, real_ltv_bp);

        // Tier 3: LTV greater than 90.00%.
        if real_ltv_bp &amp;gt;= 9000 {
            required_verde = debt;
            collateral_out = collateral;

        // Tier 2: LTV between 80.00% and 89.99%.
        } else if real_ltv_bp &amp;gt;= 8000 {
            required_verde = debt;
            // multiply debt by 110%. Collateral out equals: debt + (debt * 10%).
            collateral_out = self.from_verde_to_collat(
                collat_price_usd,
                proportional(debt, 11000, BASIS_POINTS as u128)
            );

        // Tier 1: LTV between `liquidationLtvBp` and 79.99%.
        } else if real_ltv_bp &amp;gt;= self.liquidation_ltv_bp as u128 {
            required_verde = self.get_liquidation_required_verde(debt, collateral,
                collat_price_usd);
            log!(&amp;quot;required_verde={} ** Must be ~: 196.4&amp;quot;, required_verde);
            collateral_out = self.from_verde_to_collat(
                collat_price_usd,
                proportional(
                    required_verde,
                    (BASIS_POINTS + self.liquidation_penalty_bp) as u128,
                    BASIS_POINTS as u128
                )
```

```
            );
        } else { panic!(&quot;NonLiquidatableLoan&quot;); }

        (required_verde, collateral_out)
    }
```

The liquidation design assumes collateral value >= debt value, but provides no incentive adjustment when debt exceeds collateral value, creating negative liquidation rewards above 90% LTV.

Proof of Concept: Position State:

- Collateral: 1 ETH = $2,000
- Debt: $2,200 (110% LTV)
- Status: Liquidatable (LTV > 90%)

Liquidation Economics:

- Liquidator Required Payment: $2,200 VERDE (full debt)
- Liquidator Receives: 1 ETH = $2,000 (all collateral)
- Net Result: -$200 loss for liquidator

Outcome: NO rational actor will liquidate this position

Impact:

- This protocol design would lead to bad debt Accumulation and Underwater positions become unliquidatable due to economic disincentives.
- Protocol have to absorb 100% loss on positions above 100% LTV.
- A large crash can lead to protocol insolvency.

Recommendation: - Cap the payment at 95% of the collateral value , 5% as bonus to the liquidator and protocol absorbs the bad debt difference - Create an Insurance fund which would be used to subsidize the underwater liquidations

2.6 USDMP (Verde) peg of 1$ can create Protocol Risks

Severity: High

Status: Acknowledged

Location:

- src/price_feed.rs
- src/internal.rs

Description: The protocol is built on a hardcoded assumption that VERDE tokens maintain a $1 USD peg, embedded throughout all price conversion functions. This assumption is never validated and creates systemic risk if VERDE depegs from $1, leading to incorrect collateral valuations, liquidation thresholds, and potential protocol insolvency.

```
// src/price_feed.rs:65-82
pub(crate) fn from_collat_to_verde(&amp;self, collat_price_usd: u128, amount: u128) -&gt; u128 {
    proportional(amount, collat_price_usd, self._collat_to_verde_conversion_constant)
    // Wrong , Assumes VERDE = $1 USD (no verde_price_usd parameter)
}

pub(crate) fn from_stable_to_verde(&amp;self, stable_price_usd: u128, amount: u128) -&gt; u128 {
    proportional(amount, stable_price_usd, self._stable_to_verde_conversion_constant)
    // Wrong Direct conversion assuming VERDE = stable price = $1 USD
}
```

Impact: Verde token depegging can create arbitrage opportunities and leads to potential protocol solvency as in all calculations verde token is considered as 1$ pegged.

Recommendation:
- Implement a Verde Token Oracle which reflects the actual price of verde token as per open market - Implement a Depeg Circuit Breaker.

2.7 Flawed Interest Calculation leads to unfair interest charge and gaming

Severity: High

Status: Acknowledged

Location:

- src/internal.rs

Description: The protocol's interest accrual mechanism operates on a daily basis with a binary threshold that creates significant fairness issues and gaming opportunities. Interest is only charged when time_elapsed_days >= 1, leading to scenarios where users can be charged for full days of interest despite borrowing for only hours, or conversely, borrow for nearly a full day without any interest charges and gaming of interest mechanism.

```
// src/internal.rs:12-25 and 29-42
pub(crate) fn internal_get_interest(&amp;self) -&gt; u128 {
    let mut interest = 0_u128;
    let unix_day = get_unix_day();
    let time_elapsed_days = u128::from(unix_day - self.last_accrual_day);
    if time_elapsed_days &gt;= 1 {  //  Binary threshold creates unfairness
        if self.total_debt_verde &gt; 0 {
            interest = proportional(
                self.borrow_rate_per_second * time_elapsed_days,  //  Multiplies by full
                    days
                self.total_debt_verde,
                10_u128.pow(24)
            );
        }
    }
    interest
}

pub(crate) fn internal_accrue(&amp;mut self) {
    let unix_day = get_unix_day();
    let time_elapsed_days = u128::from(unix_day - self.last_accrual_day);
    if time_elapsed_days &gt;= 1 {  //  Same binary logic
        // ... applies interest for full day increments only
        self.last_accrual_day = unix_day;  //  Updates to current day
    }
}
```

Root Cause Analysis:

1. Binary Threshold: Interest only accumulates when >= 1 full day has passed
2. Day Quantization: get_unix_day() truncates time to daily boundaries
3. All-or-Nothing: Users either pay 0% or 100% of daily interest rate
4. Gaming Window: 24-hour timing manipulation opportunities

Proof of Concept: Scenario: Unfair Maximum Charge

Borrow at Day 1, 23:59 UTC → Repay at Day 2, 00:01 UTC (2 minutes) - Charged: Full 24-hour interest for 2 minutes of borrowing

Scenario: Unfair Zero Charge

- Borrow at Day 1, 00:01 UTC → Repay at Day 1, 23:59 UTC (23h 58m)
- Charged: 0% interest for nearly 24 hours of borrowing

Scenario: Strategic Gaming

- Borrow after midnight, repay before midnight → ~24 hours free borrowing
- Repeat daily for indefinite interest-free loans

Impact:

- Flash-loan style attacks exploiting interest-free periods
- Unpredictable liquidations when interest suddenly accrues

Recommendation: It is recommended that a per-second interest rate is applied across the protocol which would cater against the unfair advantages to the users and prevent the loss of revenue to the protocol.

```
pub(crate) fn internal_accrue(&amp;mut self) {
    let current_timestamp = env::block_timestamp();
    let time_elapsed_seconds = (current_timestamp - self.last_accrual_timestamp) / 1
        _000_000_000;

    if time_elapsed_seconds &gt; 0 &amp;&amp; self.total_debt_verde &gt; 0 {
        let interest = proportional(
            self.borrow_rate_per_second * time_elapsed_seconds,
            self.total_debt_verde,
            10_u128.pow(24)
        );

        if interest &gt; 0 {
            self.mint_verde_for(&amp;self.treasury_id.clone(), interest);
        }
        self.total_debt_verde += interest;
    }

    self.last_accrual_timestamp = current_timestamp;
}
```

Developer's Response

We plan to introduce familiar amortized payment plans (e.g. weekly or monthly fixed installments) to eliminate intra-day timing effects without per-second on-chain computation.

2.8 Protocol Absorbs All Storage Costs Instead of Users

Severity: High

Status: Open

Location:

- src/internal.rs:89-91
- src/internal.rs:129
- src/admin.rs:251,271,294

Description: The protocol violates NEAR blockchain best practices by absorbing all storage costs for user operations instead of requiring users to pay for their own storage usage. Throughout the codebase, user actions that create new storage entries (borrower data, lost funds tracking, active loans, etc.) are performed without charging users for the associated storage costs, placing an unsustainable financial burden on the protocol contract.

Root cause analysis

NEAR Storage Cost Model: In NEAR, every byte of storage costs approximately 0.0001 NEAR per byte per year. When users interact with contracts and create new storage entries, the storage cost should be paid by the user, not absorbed by the contract.

Current Implementation: The protocol uses #[payable] annotations with assert_one_yocto() checks, which only require 1 yoctoNEAR attachment for security purposes, but don't cover actual storage costs.

Impact:

- Every new borrower costs ~10-20KB storage = 0.001-0.002 NEAR per user
- With 10,000 users, protocol pays ~10-20 NEAR just for basic storage
- Storage costs accumulate over time

Recommendation: It is recommended that user should bear the storage cost as per the near's best practices

```
#[payable]
pub fn borrow(&amp;mut self, collateral_id: AccountId, verde_amount: U128) -&gt; Promise {
    let initial_storage = env::storage_usage();

    // ... existing logic ...

    let final_storage = env::storage_usage();
    let storage_cost = Balance::from(final_storage - initial_storage) * env::
        storage_byte_cost();

    let attached = env::attached_deposit();
    require!(attached &gt;= storage_cost + ONE_YOCTO, &quot;InsufficientStorageDeposit&
        quot;);
```

```
12
13      if attached &gt; storage_cost + ONE_YOCTO {
14          Promise::new(env::predecessor_account_id()).transfer(attached - storage_cost -
                ONE_YOCTO);
15      }
16
17      // ... rest of logic
18  }
```

Auditor's Response

The protocol now allocates 20 Near for Storage COverage , but the issue still lies , we believe that the best practices recommended by Near should be followed where the user pay for storage as it would save against the Potential DOS in case when the Protocols 20 Near depleats

2.9 Deposit Tokens Whitelisted for Both Collateral and Swap Causes Swap Path to Become Unreachable

Severity: High

Status: Fixed

Location:

- src/deposit.rs:18-26

Description: The ft_on_transfer function handles two possible actions:

1. Adding the deposited tokens as collateral.
2. Swapping the deposited tokens.

User intent is determined implicitly by checking:

1. If the token is in self.is_collat_available, it is treated as collateral.
2. Else, if the token is in self.is_swap_available, it is processed as a swap.

Issue: If a deposited token is whitelisted for both collateral and swap, the swap path becomes unreachable because the collateral check (self.is_collat_available) takes precedence. As a result, the swap functionality is effectively disabled for such tokens.

Impact: Users intending to swap the stablecoin will instead have their tokens added as collateral, leading to unintended behavior and a broken swap feature.

Recommendation: We recommend explicitly checking the intent of user by parsing the `msg` field to check if is_swap or not.

2.10 Interest Accrual Bypass Enables Interest-Free Loans and Potential DOS

Severity: High

Status: Acknowledged

Location:

- src/deposit.rs:18-26

Description: Location: src/deposit.rs in internal_borrow() function Vulnerable Code:

```
require!(self.total_debt_verde + total_borrow_amount &lt;= self.debt_cap, &quot;
    LargerThanDebtCap&quot;);
```

Attack Scenario:

Interest-Free Borrowing & Debt Cap Manipulation

1. Borrow Timing Exploit:
    - A user takes a loan immediately after daily interest accrual occurs.
    - Since interest only compounds once per day, no interest is initially applied.
2. Repay Before Next Accrual:
    - The user repays the full principal amount just before the next 24-hour accrual period ends (e.g., at 23:59).
    - Since the 24-hour window hasn't passed, no interest is charged.
3. Repeated Borrow-Repay Cycle:
    - The attacker repeats this process in every block, effectively borrowing for free.
    - If done at scale, this artificially inflates total_debt_verde near the debt cap, blocking legitimate borrowing.

Impact:

- **Interest Avoidance:** Attackers can borrow indefinitely without paying interest, breaking protocol economics.
- **Debt Cap Manipulation:** Repeated borrow-repay cycles can trigger false debt cap violations, causing a temporary denial-of-service (DOS) for other users.

2.11 Inverted Swap Fee Validation Logic Allows Unlimited Fees

Severity: Medium

Status: Fixed

Location:

- src/utils.rs
- src/internal.rs

Description: The validate_swap_fee() function has inverted validation logic that only allows swap fees greater than the maximum allowed threshold, effectively permitting unlimited swap fees while rejecting valid ones.

```
1    pub(crate) fn validate_swap_fee(&amp;self, swap_fee_bp: BasisPoints) {
2         // @audit this check is inverted , it mean the swap fee should be greater than the
              max swap fee.
3         require!(swap_fee_bp &gt; MAX_SWAP_FEE_BP, &quot;InvalidFee&quot;);
4     }
```

Issue Analysis

- Current Logic: require!(swap_fee_bp > MAX_SWAP_FEE_BP, "InvalidFee");
- Effect: Only allows fees greater than 15% (MAX_SWAP_FEE_BP = 1500)
- Correct Logic: Should be require!(swap_fee_bp <= MAX_SWAP_FEE_BP, "InvalidFee");

Impact:

- Admin can set unlimited swap fees (e.g., 50%, 100%, 1000%)
- All reasonable fees (0-15%) are rejected as "invalid"
- Users can be charged excessive swap fees without protocol limits

Recommendation: It is recommended to change the require statement so current logic is enforced.

```
1   require!(swap_fee_bp &lt; MAX_SWAP_FEE_BP, &quot;InvalidFee&quot;);
```

2.12 Missing Price Staleness Validation Enables Stale Oracle Data Usage

Severity: Medium

Status: Open

Location:

- src/price_feed.rs

Description: The after_get_asset_price_callback() function retrieves price data from the oracle but fails to validate whether the price is stale, potentially allowing critical protocol operations to execute with outdated price information.

The PriceData struct contains staleness validation fields that are completely ignored:

```
pub(crate) struct PriceData {
   pub timestamp: Timestamp,
   pub recency_duration_sec: DurationSec,
   pub prices: Vec&lt;AssetOptionalPrice&gt;,
}
```

Impact:

- Stale price can have following impact on the user's positions and operations:
- Users liquidated based on outdated prices when positions are actually safe
- Legitimate positions seized due to oracle staleness

Recommendation: It is recommended to add a staleness validation check.

```
pub fn after_get_asset_price_callback(&amp;self) -&gt; U128 {
   match env::promise_result(0) {
       PromiseResult::Successful(result) =&gt; {
           let price_data = near_sdk::serde_json::from_slice::&lt;PriceData&gt;(&amp;
               result).unwrap();

           // Validate price freshness
           let current_timestamp = env::block_timestamp_ms();
           let price_age_ms = current_timestamp - price_data.timestamp;
           let max_age_ms = price_data.recency_duration_sec as u64 * 1000;

           require!(
               price_age_ms &lt;= max_age_ms,
               &quot;PriceDataStale&quot;
           );

           let price = price_data
               .prices
               .get(0)
               .expect(&quot;PriceNotFound&quot;)
               .price
               .clone()
               .expect(&quot;PriceNotFound&quot;);
           price.multiplier
       }
       PromiseResult::Failed =&gt; {
```

```
26              panic!(&quot;PriceNotFound&quot;);
27          }
28      }
29  }
```

Auditor's Response

The `stale_price_threshold` is set to `MILLIS_PER_DAY` which is 86_400_000 i.e 24 hrs , We advise that the threshold should be less.

2.13 liquidation can be done if the protocol is paused

Severity: Medium

Status: Fixed

Location:

Description: The liquidation functionality lacks pause checks, allowing liquidations to proceed even when the protocol has paused critical user operations like deposits, withdrawals, and borrowing. This creates an unfair advantage for liquidators and exposes users to forced liquidations during operational emergencies (e.g price oracle are paused and stale price is not checked) when they cannot defend their positions.

Proof of Concept: Unfair Liquidation Scenarios

Scenario: Emergency Deposit Pause

- Protocol pauses deposits due to oracle issues or market volatility
- User positions become liquidatable due to price movements
- Users cannot add collateral to save their positions (deposits paused)
- Liquidators can still liquidate user positions (no liquidation pause)

Result: Forced liquidations when users have funds but cannot use them

Recommendation: It is recommended to add Liquidation protection in case the protocol is paused as it would be fair for the borrowers.

2.14 Capital-intensive liquidation mechanism increases risk of insolvency

Severity: Medium

Status: Acknowledged

Location:

- src/liquidation.rs:74

Description: The protocol's get_liquidation_required_verde function enforces that liquidators must repay enough debt to restore a borrower's position fully to the safe LTV (e.g., 50%) rather than allowing partial or incremental repayments. This means, for example, if a borrower has $20M collateral and $14M debt (70% LTV), the liquidator must repay $4M to bring the LTV back to 50%. Partial repayments that reduce LTV only slightly (e.g., from 70% to 65%) are not permitted.

```
proportional(
            debt - safe_collateral_value,
            BASIS_POINTS as u128,
            self.get_liquidation_price_denominator(
                self.safe_ltv_bp as u128, /
                self.liquidation_penalty_bp as u128
            )
        )
```

Impact: The rigid liquidation model requires large capital outlay to liquidate big positions, which discourages smaller liquidators and reduces competition. This favors attackers who open large positions and let them approach or cross liquidation thresholds, knowing they can strategically self-liquidate to avoid penalties if liquidation attempts appear. If a liquidator finally steps in to repay the full debt, the attacker can front-run the transaction and protect their capital, effectively gaming the system. This behavior creates asymmetry where attackers face limited downside while pushing the protocol toward bad debt and potential insolvency.

Recommendation: We recommend refactoring the liquidation mechanism to allow partial liquidations, where liquidators can repay a portion of the borrower's debt sufficient to improve their health factor incrementally (e.g., reduce LTV from 70% to 65%).

2.15 Missing Pause Controls on Debt Repayment Function

Severity: Medium

Status: Fixed

Location:

- src/internal.rs

Description: The repay() function lacks any pause mechanism checks, creating an asymmetric pause control system where users can be prevented from managing their debt positions during protocol emergencies.

Proof of Concept: Primary Scenario:

Borrowing Pause Traps Users Protocol pauses borrowing due to emergency (oracle issues, market volatility, system maintenance): - Users cannot obtain new VERDE tokens to repay existing loans - Existing debt continues accruing interest while users are blocked from debt management - Users become trapped in deteriorating positions they cannot escape

Example Timeline:

Day 1: Admin pauses borrowing due to oracle malfunction Day 2: User's debt grows from interest accrual Day 3: User wants to repay but has no VERDE tokens Day 4: User tries to borrow VERDE to repay → BLOCKED by pause Day 5: User's position becomes liquidatable due to accumulated interest

Result: Forced liquidation that could have been prevented

Recommendation: It is recommended to add Repay Pause Control. Another approach can be to introduce a protocol wise pause "pause_all_operations" in case of extreme emergency.

2.16 Compound Interest Implementation Overcharge Users

Severity: Low

Status: Acknowledged

Location:

src/internal.rs:41-26

Description: In the current implementation, the `internal_accrue` function compounds interest daily by adding accrued interest to the `total_debt_verde` before calculating the interest for the next period. This means that each day, interest is calculated not just on the principal debt, but also on previously accumulated interest.

‘self.total_debt_verde += interests;“

As a result, the actual borrowing cost exceeds the stated x% annual rate due to compounding effects.

Mathematical Impact Suppose the advertised 15% APR (Annual Percentage Rate) assumes simple interest, where:

```
Interest = Principal x Rate x Time
```

However, with daily compounding, the effective annual rate (EAR) becomes higher:

```
EAR = (1 + (Rate / Days))^Days - 1
```

If `borrow_rate_per_second` scales to 15% APR, daily compounding would yield an actual rate of 16.18% (assuming 365 days).

Recommendation: We recommend either updating the docs to reflect the compounding interest rates or modify the inteset accrual to apply on principle debt.

2.17 Single Oracle Dependency Creates Manipulation Risk and Protocol Failure Points

Severity: Low

Status: Acknowledged

Location:

- src/price_feed.rs
- src/lib.rs

Description: The protocol exhibits over-reliance on a single oracle source (priceoracle.near by NearDefi) for all price determinations. This architectural decision creates multiple attack vectors and systemic failure risks that could result in protocol insolvency or user fund loss.

Impact:

1. Liquidation Triggers: All liquidation decisions depend on oracle prices
2. Borrowing Limits: Collateral valuations determine maximum borrowable amounts
3. Swap Operations: Stable token exchange rates rely on oracle data
4. Risk Management: All LTV calculations use single price source

Recommendation: It is recommended that multi-oracle approach should be taken , pyth is a reliable and reputable oracle which is implemented on Near Blockchain , it should also be used in parallel to NearDefi's oracle.

2.18 Weak Input Validation Allows Invalid Swap Format Values

Severity: Low

Status: Fixed

Location:

- src/deposit.rs

Description: The parse_swap_to_verde_args() function accepts any numeric value greater than 0 as a valid swap format identifier, rather than strictly validating the documented formats [0] and [1]. This allows invalid formats like [5,receiver_id,amount] to be processed as valid input.

Current vs Expected Validation

Expected

Format [0,receiver_id,min_verde_out] → is_verde_amount = false
Format [1,receiver_id,verde_amount] → is_verde_amount = true

Current Validation (any value > 0):
Format [5,receiver_id,amount] → is_verde_amount = true (incorrectly accepted)
Format [99,receiver_id,amount] → is_verde_amount = true (incorrectly accepted)

Impact: Invalid format values trigger the wrong swap processing path.

Recommendation: Apply Strict Format Validation whereby only 0 and 1 are considered as valid msg format and rest are discarded.

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

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

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

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