certora

Security Assessment and Formal Verification Report

Templar Protocol

September 2025

Prepared for Templar

certora

Table of content

certora

Audit Report

Project Summary

This document describes the manual code review findings of **Templar Protocol**. The work was undertaken from **August 25, 2025,** to **September 19, 2025**.

During this time, Certora's security researchers performed a manual audit of all the Rust contracts and discovered several bugs in the codebase, which are summarized in the subsequent section.

Project Scope

Project Name	Repository (link)	Commit Hash	Final Commit Hash	Platform
Templar Protocol	https://github.com/Templar-Protocol/contracts	a96e44d	08d32ef	NEAR, Rust

The following contract list is included in our audit scope:

```
contract/market/src/impl_helper.rs
contract/market/src/impl_market_external.rs
contract/market/src/impl_token_receiver.rs
contract/market/src/lib.rs
contract/registry/src/lib.rs
common/src/accumulator.rs
common/src/asset.rs
common/src/borrow.rs
common/src/chunked_append_only_list.rs
common/src/event.rs
```

```
common/src/fee.rs
common/src/interest_rate_strategy.rs
common/src/lib.rs
common/src/number.rs
common/src/price.rs
common/src/snapshot.rs
common/src/static_yield.rs
common/src/supply.rs
common/src/time_chunk.rs
common/src/withdrawal_queue.rs
common/src/market/external.rs
common/src/market/configuration.rs
common/src/market/impl.rs
common/src/market/mod.rs
common/src/market/price_oracle_configuration.rs
common/src/oracle/mod.rs
common/src/oracle/price_transformer.rs
common/src/oracle/pyth.rs
```

Protocol Overview

Templar Protocol is a chain-agnostic overcollateralized lending DeFi protocol deployed on the **NEAR** blockchain. The protocol implements a lending market system where users can:

- Supply assets to earn yield from borrower interest payments
- Use assets as collateral to borrow other assets with overcollateralization requirements
- Participate in liquidations of undercollateralized positions to maintain protocol solvency

Key Features:

- Overcollateralized borrowing with configurable collateral ratios (maintenance and liquidation thresholds)
- Time-based interest accumulation through snapshot-based yield distribution

- FIFO withdrawal queue system for managing supply withdrawals
- Insurance fund mechanism for protocol risk management
- Oracle-based pricing with configurable price feeds and maximum age limits
- Flexible asset configuration supporting NEP-141 tokens with customizable parameters

Architecture:

- Market Contract: Core lending logic, position management, and external interactions
- Registry Contract: Market discovery and deployment management
- Common Library: Shared data structures, calculations, and business logic

The protocol uses a snapshot-based system for interest accumulation and yield distribution, enabling efficient batch processing of time-based calculations across all positions.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	2	2	2
High	10	10	8
Medium	16	16	11
Low	5	5	3
Informational	8	8	6
Total	41	41	30

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
C-01	An accounting error leading to protocol insolvency	Critical	Fixed
C-02	Incorrect Liquidation Amount Formula Causes Over-liquidation and Wrong Post-Liquidation CR	Critical	Fixed
H-01	Unfair distribution of yield	High	Fixed
H-02	Borrowers cannot add collateral to restore solvency if flagged for liquidation	High	Fixed
H-03	Large oracle confidence can incorrectly trigger liquidations	High	Acknowledged
H-04	Concurrent borrow operations can bypass the maximum borrow range limits	High	Fixed
H-05	Finalized snapshots record wrong end timestamps, leading to interest durations being miscomputed	High	Fixed
H-06	Untracked Yield Breaks Important Invariant	High	Fixed
H-07	Incorrect Accounting of liquidation_lock Causes Inconsistent State	High	Fixed
H-08	Underwater Liquidation Leaves Residual	High	Partially Fixed

	Debt		
H-09	Continuous Yield Distribution Without Interest Payment Causes Balance Mismatch and Invalid Liquidity Assumptions	High	Fixed
H-10	Incorrect Accounting of Borrowable Liquidity Causes Core Invariant Violation	High	Fixed
M-01	Unsafe CR configurations	Medium	Acknowledged
M-02	market.collateral_asset_deposited is not reduced during liquidation	Medium	Fixed
M-03	Missing maximum borrow cap allows whale positions that cannot be liquidated	Medium	Fixed
M-04	Bad debt not covered by the insurance fund during liquidations	Medium	Acknowledged
M-05	Assets can be borrowed from borrow positions that are about to expire	Medium	Acknowledged
M-06	Immutable borrow_maximum_duration_ms might prove to be problematic in some specific cases	Medium	Acknowledged
M-07	Borrowers might be charged extra interest on repayment	Medium	Fixed
M-08	Incorrect Liquidity Accounting Temporarily Blocks Legitimate Borrows	Medium	Fixed

M-09	Borrow / Withdrawal Denial via NEP-141 Storage Opt-Out (In-Flight Inflation Attack)	Medium	Fixed
M-10	Amortized Interest Not Cleared on Full Repayment Allows Borrowers to Underpay Interest	Medium	Fixed
M-11	Borrowers Charged Full Snapshot Interest Even for Late Loans	Medium	Partially Fixed
M-12	Incorrect Interest Calculation After Partial Repayment	Medium	Fixed
M-13	Delayed Activation of Deposits Causes Unfair Yield Accrual	Medium	Fixed
M-14	Under-liquidation when full liquidation is required	Medium	Fixed
M-15	Delayed Liability Reduction Causes Temporarily Inflated Borrowed Amount	Medium	Fixed
M-16	DoS on liquidations	Medium	Fixed
L-01	Big withdrawal requests might clog the withdrawal queue	Low	Fixed
L-02	Storage is not being cleaned up after liquidation	Low	Fixed
L-03	Withdrawal-fee bypass by leaving a 1-token	Low	Acknowledged
L-04	Collateral Loss Due to Double Withdrawal	Low	Fixed

	and Premature Position Removal		
L-05	Low minimum withdrawal limits can lead to temporary withdrawal DoS	Low	Acknowledged
I-01	Suppliers Do Not Receive Yield from the Current Running Snapshot on Withdrawal	Informational	Acknowledged
I-02	Denial of Service on Liquidations via Locking Mechanism and Storage Opt-Out	Informational	Fixed
I-03	Snapshot not updated when market.collateral_asset_deposited is updated	Informational	Fixed
I-04	Unaccounted Current Snapshot Interest During Liquidation	Informational	Fixed
I-05	Incorrect value of deposited_incoming assigned to new snapshot	Informational	Fixed
I-06	Precision Loss in Time-Based Fee Calculation (Division Before Multiplication)	Informational	Fixed
I-07	Redundant MCR Checks in Collateral Withdrawal Validation	Informational	Acknowledged
I-08	Lack of slippage protection for liquidations	Informational	Fixed

Critical Severity Issues

C-01. An accounting error leading to protocol insolvency

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: impl.rs	Status: Fixed	

Summary

When finalizing snapshots, the protocol incorrectly pushes a snapshot that does not reflect newly activated deposits. This stale state causes a mismatch between the `borrow_asset_deposited_active` in snapshot(**denominator**) and the user positions that already account for activated deposits(**nominator**), resulting in the yield being calculated with a smaller denominator. Consequently, suppliers can claim more yield than what was actually generated, leading to protocol insolvency.

Description:

The problem arises during snapshot finalization (`snapshot_with_yield_distribution`):

Rust

```
            } else {
                // Otherwise, finalize the current snapshot and create a new one.
                let deposited_incoming = self
                    .borrow_asset_deposited_incoming
                    .remove(&self.finalized_snapshots.len())
                    .unwrap_or(0.into());
                asset_op!(self.borrow_asset_deposited_active += deposited_incoming);
                let mut snapshot = Snapshot::new(time_chunk);
                snapshot.set_yield_distribution(yield_distribution);
                snapshot.set_borrow_asset_deposited_incoming(deposited_incoming);
                snapshot.update_active(
                    self.borrow_asset_deposited_active,
```

```
                self.borrow_asset_borrowed,
                self.collateral_asset_deposited,
                &self.configuration.borrow_interest_rate_strategy,
            );
            std::mem::swap(&mut snapshot, &mut self.current_snapshot);
            MarketEvent::SnapshotFinalized {
                index: self.finalized_snapshots.len(),
                snapshot: snapshot.clone(),
            }
            .emit();
            self.finalized_snapshots.push(snapshot);
        }


        self.finalized_snapshots.len()
    }
```

- `deposited_incoming` is activated and added to `borrow_asset_deposited_active` (e.g., 11000 + 1100 = 12100).
- A new snapshot is created with this updated value.
- But after `std::mem::swap`, the **old current snapshot** (with stale `borrow_asset_deposited_active = 1100`) is pushed to `finalized_snapshots`.

This ordering means that:

- `current_snapshot.borrow_asset_deposited_active` is updated (12100 in the example).
- But the snapshot **pushed into finalized_snapshots** reflects the stale pre-activation value (1100).

Example:

- **Initial state**
  - Finalized snapshot length = 8
  - `borrow_asset_deposited_active = 1100` (Alice's original deposit)

- `borrow_asset_borrowed = 1000` (Bob's borrow)

- **Alice supplies again**
  - She deposits **11000**.
  - This is recorded as *incoming* for snapshot index **9**:

```
None
borrow_asset_deposited_incoming(9) = 11000
```

- **Bob repays**
  - Repayment triggers snapshot progression.
  - `deposited_incoming = 11000` is removed and added to `borrow_asset_deposited_active`:

```
None
borrow_asset_deposited_active = 1100 + 11000 = 12100
```

- **A new snapshot is created**
  - A new `Snapshot` is initialized with:
    - `borrow_asset_deposited_active = 12100` (from updated market state)
- **Swap and push**
  - `std::mem::swap` exchanges the new snapshot with `current_snapshot`.
  - Then the old snapshot (with **1100** active, not 12100) is pushed into `finalized_snapshots[9]`.
  - Result:
    - `current_snapshot.borrow_asset_deposited_active = 12100` (correct, used in positions)
    - `finalized_snapshots[9].borrow_asset_deposited_active = 1100` (stale, incorrect)
- **Alice accumulates yield**
  - Her position sees `amount = 12100` (including her new deposit).
  - Snapshot 9 reports `borrow_asset_deposited_active = 1100`.

- Yield distribution for snapshot 9 = 81.
- Calculation:

```
None
accumulated = (12100 * 81) / 1100 = 891
```

Only **81 yield** was actually distributed, but Alice records **891 yield**. The protocol overpays suppliers, leading to insolvency.

PoC: Paste the test in happy_path.rs

```rust
Rust
#[rstest]
#[case(false, false)]
#[case(false, true)]
#[case(true, false)]
#[case(true, true)]
#[allow(clippy::too_many_lines)]
#[tokio::test]
async fn test_insolvency(#[case] borrow_mt: bool, #[case] collateral_mt: bool) {
    setup_test!(
        extract(c, protocol_yield_user, insurance_yield_user)
        accounts(borrow_user, supply_user, supply_user_2)
        config(|c| {
            c.time_chunk_configuration = TimeChunkConfiguration::BlockTimestampMs {
                divisor: 500.into(), // 0.5 seconds
            };
            c.borrow_interest_rate_strategy =
                InterestRateStrategy::linear(dec!("2"), dec!("6")).unwrap();
            if borrow_mt {
                c.borrow_asset =
                    FungibleAsset::nep245(
                        c.borrow_asset.clone().into_nep141().unwrap(),
                        "mt_borrow".into(),
                    );
            }
            if collateral_mt {
```

```
            c.collateral_asset =
                FungibleAsset::nep245(
                    c.collateral_asset.clone().into_nep141().unwrap(),
                    "mt_collateral".into(),
                );
        }
    })
);

let configuration = c.get_configuration().await;

assert!(configuration.borrow_mcr_liquidation.near_equal(dec!("1.2")));

let bounds = c.storage_balance_bounds().await;

assert!(!bounds.min.is_zero());

let snapshots_len = c.get_finalized_snapshots_len().await;
assert_eq!(snapshots_len, 1, "Should generate single snapshot on init");

let snapshots = c.list_finalized_snapshots(None, None).await;
assert_eq!(snapshots.len(), 1);
assert!(snapshots[0].yield_distribution().is_zero());
assert!(snapshots[0].borrow_asset_deposited_active().is_zero());
assert!(snapshots[0].borrow_asset_borrowed().is_zero());

// Step 1: Supply user sends tokens to contract to use for borrows.
println!("=======User1 supplying=======");
c.supply(&supply_user, 1100).await;

let supply_position = c.get_supply_position(supply_user.id()).await.unwrap();

let mut snap = c.get_finalized_snapshots_len().await;
println!("Snapshots len after supplying: {}", snap);
assert_eq!(
    u128::from(supply_position.total_incoming()),
    1100,
    "Supply position should match amount of tokens supplied to contract",
);

// Wait for activation.
```

```
while !c
    .get_supply_position(supply_user.id())
    .await
    .unwrap()
    .get_deposit()
    .incoming
    .is_empty()
{
    c.harvest_yield(&supply_user, None, None).await;
}

snap = c.get_finalized_snapshots_len().await;
println!("Snapshots len after supply activation: {}", snap);

let supply_position = c.get_supply_position(supply_user.id()).await.unwrap();

assert_eq!(
    u128::from(supply_position.get_deposit().active),
    1100,
    "Supply position should match amount of tokens supplied to contract",
);

// Step 2: Borrow user deposits collateral
println!("=======Borrower collateralizing=======");
c.collateralize(&borrow_user, 2000).await;

let borrow_position = c.get_borrow_position(borrow_user.id()).await.unwrap();

assert_eq!(
    u128::from(borrow_position.collateral_asset_deposit),
    2000,
    "Collateral asset deposit should be equal to the number of collateral tokens sent",
);

let borrow_status = c
    .get_borrow_status(borrow_user.id(), c.get_prices().await)
    .await
    .unwrap();

assert_eq!(
    borrow_status,
```

```
        BorrowStatus::Healthy,
        "Borrow should be healthy when no assets are borrowed",
    );

    // Step 3: Withdraw some of the borrow asset
    println!("=======Borrower borrowing=======");
    let balance_before = c.borrow_asset.balance_of(borrow_user.id()).await;

    // Borrowing 1000 borrow tokens with 2000 collateral tokens should be fine given equal
price and MCR of 120%.
    let mut res = c.borrow(&borrow_user, 1000).await;
    for log in res.logs() {
        eprintln!("[contract] {log}");
    }
    let balance_after = c.borrow_asset.balance_of(borrow_user.id()).await;

    assert_eq!(
        balance_before + 1000,
        balance_after,
        "Borrow user should receive assets"
    );

    let borrow_position = c.get_borrow_position(borrow_user.id()).await.unwrap();

    assert_eq!(u128::from(borrow_position.collateral_asset_deposit), 2000);
    assert_eq!(
        u128::from(borrow_position.get_total_borrow_asset_liability()),
        1000 + 100, // origination fee
    );

    println!("=======User1 supplying again=======");
    res = c.supply(&supply_user, 11000).await;
    for log in res.logs() {
        eprintln!("[contract] {log}");
    }

    // Step 4: Repay borrow
    println!("=======Borrower repaying=======");
    res = c.repay(&borrow_user, 1123).await;
    for log in res.logs() {
        eprintln!("[contract] {log}");
```

```
    }
    snap = c.get_finalized_snapshots_len().await;
    println!("Snapshots len after repayment: {}", snap);
    // Ensure borrow is paid off.
    let borrow_position = c.get_borrow_position(borrow_user.id()).await.unwrap();

    assert_eq!(u128::from(borrow_position.collateral_asset_deposit), 2000);
    assert_eq!(
        u128::from(borrow_position.get_total_borrow_asset_liability()),
        0,
    );

    join!(
        // Supply withdrawals.
        async {
            // Withdraw yield.
            {
                println!("=======User1 harvesting yield=======");
                let mut res1 = c
                    .harvest_yield(&supply_user, None, Some(HarvestYieldMode::Default))
                    .await;

                let mut supply_position =
c.get_supply_position(supply_user.id()).await.unwrap();
                println!(
                    "Supply position: {:?}",
                    supply_position.borrow_asset_yield.get_total()
                );

                // Move the yield to the principal so that it can be withdrawn
                let amount_moved_to_principal = c
                    .harvest_yield(&supply_user, None, Some(HarvestYieldMode::Compounding))
                    .await;

                while !c
                    .get_supply_position(supply_user.id())
                    .await
                    .unwrap()
                    .get_deposit()
                    .incoming
                    .is_empty()
```

```
                {
                    c.harvest_yield(&supply_user, None, None).await;
                }
                supply_position = c.get_supply_position(supply_user.id()).await.unwrap();
                let total_deposited = supply_position.get_deposit().total();
                println!("Total deposited: {}", total_deposited);

                c.supply(&supply_user_2, 1000).await;

                while !c
                    .get_supply_position(supply_user_2.id())
                    .await
                    .unwrap()
                    .get_deposit()
                    .incoming
                    .is_empty()
                {
                    c.harvest_yield(&supply_user_2, None, None).await;
                }

                let balance_before = c.borrow_asset.balance_of(supply_user.id()).await;
                println!("Balance before {}", balance_before);
                // Withdraw all
                res = c
                    .create_supply_withdrawal_request(&supply_user, total_deposited)
                    .await;
                for log in res.logs() {
                    eprintln!("[contract] {log}");
                }
                res = c.execute_next_supply_withdrawal_request(&supply_user).await;
                for log in res.logs() {
                    eprintln!("[contract] {log}");
                }
                let balance_after = c.borrow_asset.balance_of(supply_user.id()).await;
                println!("Balance after {}", balance_after);
            }
        }
    );
}
```

Recommendations: Ensure that the snapshot pushed to `finalized_snapshots` contains the **updated active deposit values**.

Customer's response: Fixed in 5f1abbb.

Fix Review: Confirmed.

C-02. Incorrect Liquidation Amount Formula Causes Over-liquidation and Wrong Post-Liquidation CR

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: borrow.rs	Status: Fixed	

Description:

The current formula for liquidatable_collateral ignores the fact that the protocol only recovers net USD = c · P · (1 – spread), but solves the equation as if recovery were c · P. This results in over-liquidation of the positions.

Example:

- Collateral: 2 BTC
- Liability: $85,000 USDC
- BTC Price: $50,000 USDC
- MCR: 1.2
- Spread: 5% (0.05)

As per current logic,

```JavaScript
scaled_liability = 85,000 / 50,000 = 1.7 BTC
scaled_collateral = 2 * 0.95 / 1.2 = 1.5833 BTC
unscaled = (1.7 – 1.5833) / 0.2 = 0.5835
amount = 1.2 * 0.5835 = 0.7002 BTC  ← returned by contract
```

But liquidating 0.7002 BTC gives:

- Recovered USD = 0.7002 · 50,000 · 0.95 = 33,259.50

- New collateral = 1.2998 BTC = 64,990 USD
- New liability = 51,740.50 USD
- CR = 1.256 (> MCR) – over-liquidation.

If we compare it with the correct formula `(c - l) / (b - l * (1 - spread)) = m`

- Liquidatable_collateral = 0.285714 BTC

Which yield,

- New collateral = 1.714286 BTC * 50000 = 85,714.3
- New liability = 85,000 – 13,571.43 = 71,428.57
- CR = exactly 1.2

Therefore, as per current logic, the difference of 0.4145 BTC is incorrectly liquidated from the user's position.

Recommendations:

Modify the formula to return the correct liquidation amount.

Customer's response: Fixed in 704c7dd.

Fix Review: Confirmed.

High Severity Issues

H-01. Unfair distribution of yield		
Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Supply.rs	Status: Fixed	

Description: Currently, yield is only recorded when discrete events occur (e.g., borrower repayment or liquidation). This means distribution is not tied to the actual **time a supplier's funds were active**, but rather to who happens to be active when an event occurs.

Example:

1. **t=9**: Alice supplies **1,000 tokens**.
2. **t=11**: Borrower borrows **900 tokens**.
3. **t=111**: After 100 seconds (while Alice's funds earned interest), Bob supplies **10,000 tokens**.
    - By this point, **100 tokens of interest** have accrued.
    - Ideally, this interest should belong entirely to Alice, since she was the sole liquidity provider during the accrual.
4. **t=113**: Borrower repays **900 principal + 100 interest**.

Current Distribution:

- At repayment, active supply = 11,000.
- Interest is split proportionally across both suppliers:
    - Alice = $1000 \times 100 / 11000 = 9.09$
    - Bob = $10000 \times 100 / 11000 = 90.90$

Therefore, the yield distribution is unfair.

Recommendations: In the current design, there does not seem to be a straightforward fix to this problem, since yield is only realized during discrete events. However, the ideal approach

would be to have yield accounted for incrementally in each snapshot, based on the total borrowed amount and interest rate.

Customer's response: Fixed in d3ef507.

Fix Review: Confirmed.

H-02. Borrowers cannot add collateral to restore solvency if flagged for liquidation

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: impl_helper.rs	Status: Fixed	

Description:

The `execute_collateralize` function currently prevents users from adding collateral if their borrow position is flagged as eligible for liquidation:

```
Rust
    require!(
        !borrow_position.is_eligible_for_liquidation(price_pair, env::block_timestamp_ms()),
        "Cannot add collateral when eligible for liquidation",
     );
```

This check occurs before the new collateral deposit is recorded:

```
Rust
borrow_position.record_collateral_asset_deposit(proof, amount);
```

This restriction can cause unnecessary liquidations. A position may not truly be liquidable at the real market price, but the use of conservative price bounds (price - confidence, price + confidence for debt) can make it appear liquidable.

For example, consider a market with liquidation CR at 130% and a user position with a borrowed asset amount of 100 USDC and $150 worth of collateral (150% CR). The price moves down, and now the user has $131 worth of collateral. At the real market price, their collateralization ratio may still be safe. However, when applying a $3 confidence buffer, the collateral is valued at $128, dropping the effective CR to 128%, which falls below the required minimum. The user is then blocked from adding collateral and saving their position, even though their position would not actually be liquidable under the true market price.

Allowing users to top up their collateral at any time, even when their positions are close to or below liquidation thresholds, protects them during periods of high volatility and avoids premature liquidations.

Recommendations:

Move the liquidation eligibility check after recording the collateral deposit. This ensures new collateral is always applied, allowing users to strengthen their position and avoid unnecessary liquidations.

JavaScript

```
    pub fn execute_collateralize(
        &mut self,
        account_id: AccountId,
        amount: CollateralAssetAmount,
        price_pair: &PricePair,
    ) {
        if self.borrow_position_ref(account_id.clone()).is_none() {
            self.charge_for_storage(
                &account_id,
                self.storage_usage_borrow_position + self.storage_usage_snapshot * 2,
            );
        }

        let mut borrow_position = self.get_or_create_borrow_position_guard(account_id);
        if borrow_position.inner().is_liquidation_locked {
            env::panic_str("Cannot add collateral while liquidation locked");
        }
        let proof = borrow_position.accumulate_interest();

-       require!(
-         !borrow_position.is_eligible_for_liquidation(price_pair,
env::block_timestamp_ms()),
-           "Cannot add collateral when eligible for liquidation",
-       );
        borrow_position.record_collateral_asset_deposit(proof, amount);

+       require!(
```

```
+         !borrow_position.is_eligible_for_liquidation(price_pair,
env::block_timestamp_ms()),
+           "The position is eligible for liquidation after adding collateral",
+       );
    }
```

Customer's response: Fixed in 4e2f495.

Fix Review: Confirmed.

H-03. Large oracle confidence can incorrectly trigger liquidations

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files:	Status: Acknowledged	

Description:

The protocol currently determines liquidation eligibility using conservative calculations:

- Collateral: valued pessimistically (price - confidence).
- Debt: valued optimistically (price + confidence)

This aligns with Pyth's safety guidance but can cause problems during high volatility, when oracle confidence (σ) becomes unusually large. A healthy position at true prices may appear liquidable once confidence adjustments are applied.

From Pyth's documentation:

> "It can decide that there is too much uncertainty when σ/μ exceeds some threshold and choose to pause any new activity that depends on the price of this asset."

In such extreme cases, relying on `price ± confidence` adjustments creates artificial liquidations that do not reflect the real market price. Example

Example

- **ETH price:** $1,000
- **Confidence interval:** ±$100 (10%)
- **Borrower collateral:** 13 ETH = $13,000
- **Borrower debt:** 10,000 USDC
- **Liquidation CR:** 130%

True CR (no confidence):

$13,000 / $10,000 = **130%** → exactly at liquidation threshold, but **not liquidable**.

Position CR (with high confidence):

- Collateral pessimistic = $900 × 13 = **$11,700**
- Debt optimistic = **$10,000** (USDC confidence negligible)
- CR = $11,700 / $10,000 = **117%** → far below liquidation CR.

Here, a 10% confidence interval, which can occur during severe volatility or when data publishers diverge, causes a solvent position to be flagged liquidable.

Recommendations:

Introduce a confidence cap or uncertainty threshold, aligned with Pyth's guidance. Options include:

1. Pause actions (liquidations, new borrows) when `confidence/price` exceeds a safe ratio.
2. Cap confidence at a maximum percentage when applied in collateral/borrow valuation, so extreme confidence values do not unfairly penalize users.

This ensures borrowers are not liquidated solely due to a temporary oracle disagreement or market disruption.

Customer's response: Acknowledged. This is how lending protocols should work. There is no such thing as a single "true price" of an asset—all markets have a bid/ask spread. Plus, we are intending to support cross-chain liquidations, meaning that we want a price that is representative of more than one market. So, we have to assume that the "worst" price for collateral may be the best price available. The required over-collateralization ratio should sufficiently protect the user; the exact definition of "sufficient" protection is up to the user when deciding their own risk tolerance.

H-04. Concurrent borrow operations can bypass the maximum borrow range limits

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: impl_market_external.rs	Status: Fixed	

Description:

The `borrow()` function validates borrow amounts against `borrow_range.maximum` using only the current `borrow_asset_principal`, ignoring pending in-flight borrows. This creates a race condition where multiple concurrent borrow calls can each pass the range validation independently, allowing users to exceed the intended maximum borrow limit.

```JavaScript
    let mut borrow_principal = borrow_position.inner().get_borrow_asset_principal();
    asset_op!(borrow_principal += amount);
```

Attack scenario:

- User calls `borrow(max_amount)` twice in the same block
- Both calls read `borrow_asset_principal = 0` and pass range validation
- Both proceed to async execution, each adding `max_amount` to `temporary_lock`
- Result: User borrows 2 × max_amount, bypassing the configured limit

While collateral requirements still prevent protocol insolvency, this bypasses intended risk management controls designed to limit position sizes.

Recommendations:

Implement a proper locking mechanism to prevent concurrent borrow operations on the same position during async execution.

Customer's response: Fixed in 1843476.

Fix Review: Confirmed.

H-05. Finalized snapshots record wrong end timestamps, leading to interest durations being miscomputed

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: impl.rs	Status: Fixed	

Description:

When the contract finalizes a snapshot, it swaps the *new* `Snapshot` with `current_snapshot` and then pushes the swapped object into `finalized_snapshots`. However, the `end_timestamp` of the pushed snapshot is left as the *last interaction timestamp that previously touched that snapshot*, not the true snapshot end/finalization time. As a result, finalized snapshots store a stale `end_timestamp` (last-interaction) instead of the snapshot end time.

Because interest is later computed by iterating finalized snapshots and summing

```rust
snapshot.rate * (end_timestamp_current - end_timestamp_previous)
```

using those stored `end_timestamp` values, the durations used are wrong whenever the snapshot's last-interaction time differs from the true snapshot end. This misallocates interest whenever snapshot progression (which activates incoming deposits and can change rates) happens between interactions.

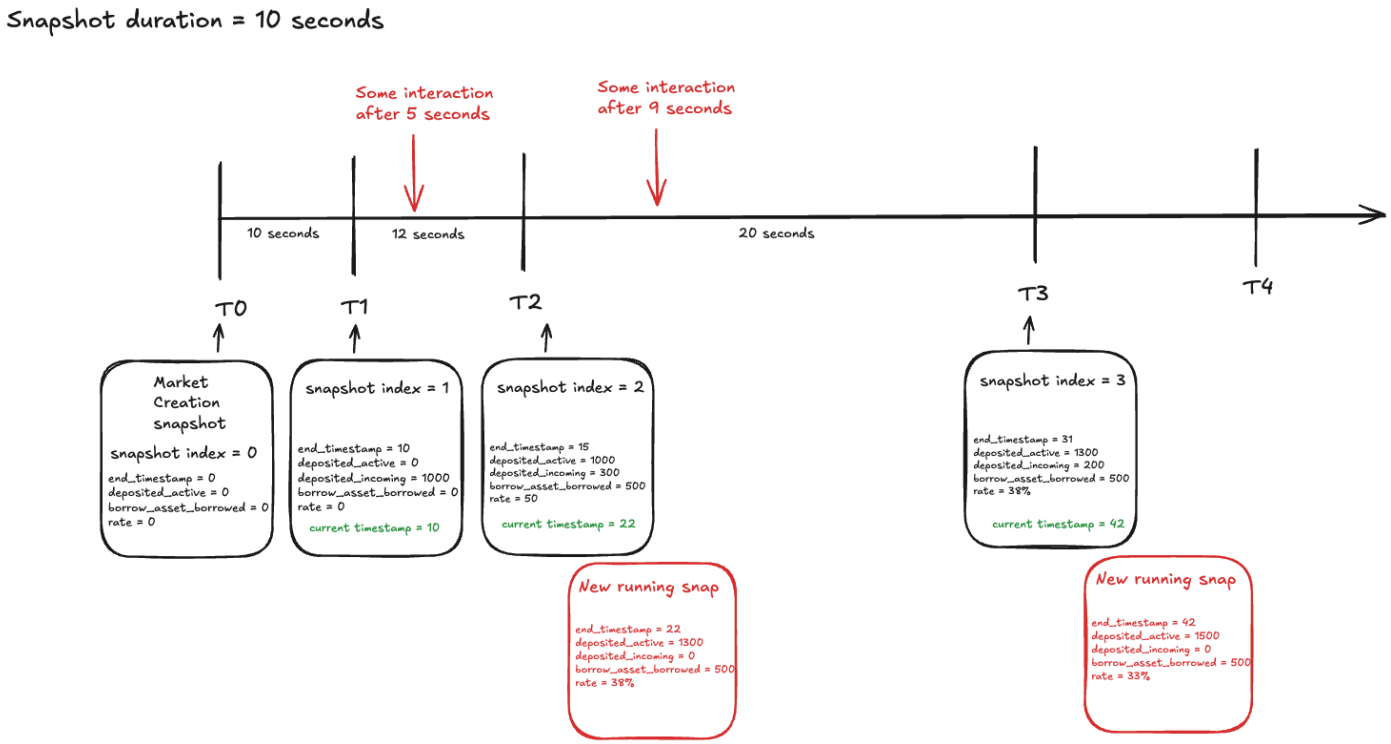
In the example given, the ideal way of calculating interest for the 2nd and 3rd is

- ***(22 - 10) 50 +*** <---(because rate changes after 22 and not 15)
- ***(42 - 22) * 38*** <---(because rate changes after 42 and not 31)
- = 1360

However, the current logic calculates,

- ***(15 - 10) * 50 +***
- ***(31 - 15) * 38***
- = 858

This is incorrect because the rate actually changes at **22** and **42**, not at **15** and **31**.



Recommendations:

Set the snapshot `end_timestamp` to the true finalization time before pushing it into `finalized_snapshots`.

Customer's response: Fixed in 5f1abbb.

Fix Review: Confirmed.

H-06. Untracked Yield Breaks Important Invariant

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: impl_market_external.rs	Status: Fixed	

Description:

The code comments state the invariant:

```rust
/// borrow_asset_deposited_active - borrow_asset_borrowed >= 0 should always be true
```

However, this invariant can be violated during certain withdrawal flows. The problem arises because withdrawals subtract directly from `borrow_asset_deposited_active`, even when the withdrawn funds actually come from **untracked yield**.

Example:

- Initial state:
  - borrow_asset_deposited_active = 1000
  - borrow_asset_borrowed = 500
  - Ideal balance = 500
- A user requests a withdrawal of 700. Normally, this should fail, but if the contract's on-chain balance is inflated (e.g., due to 200 tokens of untracked yield from repayments/liquidations), the transfer succeeds.
- When the withdrawal request is executed,
  - market.borrow_asset_deposited_active will be reduced by 700, so it will be = 300
  - market.borrow_asset_in_borrowed = 500 will stay the same

    - expect_success = borrow_asset_deposited_active - borrow_asset_in_borrowed = 300 - 500 = false
    - Even though expect_success == false, the transfer would be attempted, and it would succeed because the contract balance was 700
- Internal accounting after withdrawal:
    - borrow_asset_deposited_active = 300
    - borrow_asset_borrowed = 500
    - Invariant violated (`300 < 500`).
- Although the withdrawal succeeds using the inflated token balance, internal accounting now shows **borrowed > active deposits**. *This causes the utilization ratio to hit 100% and spikes interest rates to the maximum.*
- On further repayments or supply, the invariant may flip back, but repeated withdrawals against yield continue to create short-lived but extreme interest spikes.

This is not a one-off exploit: every time untracked yield exists in the protocol, withdrawals can again push `borrow_asset_deposited_active < borrow_asset_borrowed`.

Additionally, this can also be triggered artificially by donating tokens directly to contracts. The donor will lose those funds, but could potentially benefit in his other legitimate supply position due to the increased interest rate.

PoC: Paste the following test in untracked_funds.rs

```rust
#[tokio::test]
async fn invariant_break() {
    setup_test!(extract(c) accounts(borrow_user, borrow_user_2, supply_user));

    let mut market_contract_balance = c.borrow_asset.balance_of(c.contract().id()).await;
    assert_eq!(market_contract_balance, 0);

    tokio::join!(
        async {
            c.supply_and_harvest_until_activation(&supply_user, 10_000)
                .await;
        },
        c.collateralize(&borrow_user, 20_000),
```

```
        c.collateralize(&borrow_user_2, 20_000),
    );

    c.borrow(&borrow_user, 4_000).await;
    c.borrow(&borrow_user_2, 4_000).await;

    c.repay(&borrow_user, 4050).await;

    let mut active_deposits = c.market.get_borrow_asset_metrics().await.deposited_active;
    println!("Active deposits: {}", active_deposits);

    let mut borrowed = c
        .market
        .get_borrow_asset_metrics()
        .await
        .borrowed;
    println!("Borrowed: {}", borrowed);

    assert!(active_deposits > borrowed);

    let balance_before = c.borrow_asset.balance_of(supply_user.id()).await;

    c.create_supply_withdrawal_request(&supply_user, 6050).await;
    c.execute_next_supply_withdrawal_request(&supply_user).await;

    let balance_after = c.borrow_asset.balance_of(supply_user.id()).await;
    assert_eq!(balance_before + 6050, balance_after);

    active_deposits = c.market.get_borrow_asset_metrics().await.deposited_active;
    println!("Active deposits after: {}", active_deposits);

    borrowed = c
        .market
        .get_borrow_asset_metrics()
        .await
        .borrowed;
    println!("Borrowed after: {}", borrowed);
    assert!(active_deposits < borrowed);
}
```

```
Active deposits: 10000
Borrowed: 4351
supply_user12011 calls market2509212011->create_supply_withdrawal_request(...)
supply_user12011 calls market2509212011->execute_next_supply_withdrawal_request(...)
Active deposits after: 3950
Borrowed after: 4351
```

Impact

- Protocol economics are destabilized because the utilization ratio can be forced to 100% arbitrarily often, spiking interest rates and skewing incentives.
- **Suppliers and borrowers are mispriced**:
  - Borrowers face artificially high interest rates.
  - Suppliers may receive abnormal yields, not reflecting true liquidity conditions.
- **Protocol accounting would also be inconsistent,** and it could have other downstream effects.

Recommendations:

We recommend introducing explicit accounting for accrued yield and restricting the withdrawals to `borrow_asset_deposited_active - borrow_asset_borrowed,` excluding tracked yield.

Customer's response: Fixed in 0d326aa.

Fix Review: Confirmed.

H-07. Incorrect Accounting of liquidation_lock Causes Inconsistent State

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: borrow.rs	Status: Fixed	

Description:

During liquidation, the protocol temporarily stores the outgoing collateral in `liquidation_lock`, but this value is still included in get_total_collateral_amount(). This means that for a short period between `liquidate_transfer_call_01_consume_price` and `liquidate_transfer_call_02_finalize`, the system incorrectly believes the account still holds collateral that has already been removed from the position.

Functionally, `liquidation_lock` behaves like collateral that is already gone, similar to `collateral_asset_in_flight`, and should not be counted toward the position's collateral. Because it is still included, several incorrect behaviors arise:

1. Borrow calls can succeed when they should not (false CR due to inflated collateral)
    - Between liquidation-initiation and finalization, if a borrow happens and the collateral price increases, the position may appear healthier because liquidation_lock is still counted.
    - The borrower could then borrow more against what's no longer collateralized.

2. Multiple concurrent liquidations behave incorrectly
    - 2a. If two liquidation calls are made before the first `liquidate_transfer_call_02_finalize` executes, and the asset price changes in between the two `liquidate_transfer_call_01_consume_price` calls, incorrect results can occur
        - If the collateral price increases, the protocol will assume the position is stronger than it really is - since part of the collateral has already been attempted for transfer but not yet subtracted.

- Example: - If the first liquidation attempts to transfer 2 BTC out of 5 BTC, and before finalization another liquidation starts while BTC price has increased by 1%, the second call would incorrectly treat the position as 1% stronger than reality and hence liquidatable_collateral will give incorrect result.

- 2b. Even without price movement, CR becomes inconsistent due to mismatch in liability reduction vs collateral reduction. For example

```JavaScript
Collateral: 2 BTC
Price: $80,000
Liability: $140,000
CR: 1.1428
MCR: 1.2
Liquidation CR: 1.15

1st liquidation initiation:
LiquidatableAmount = 1 BTC
Requested = 0.5 BTC
liquidation_lock = 0.5
Liquidator pays = 40,000 - 5% = 38,000

2nd liquidation initiation:
LiquidatableAmount = 0.5 BTC
Requested = 0.5 BTC
liquidation_lock = 1 BTC
Liquidator pays = 40,000 - 5% = 38,000

1st liquidation finalization:
Collateral = 2 - 0.5 = 1.5 BTC ($120,000)
Liability = 140,000 - 38,000 = 102,000

2nd liquidation finalization:
Collateral = 1.5 - 0.5 = 1 BTC ($80,000)
Liability = 102,000 - 38,000 = 64,000
CR = 1.25
Total collateral liquidated = 1 BTC
Total borrow recovered = 76,000
```

- But ideally, after the first initiation:

```
JavaScript
Collateral = 1.5 BTC ($120k)
Liability = 140k − 38k = 102k
CR = 1.1764 (healthy)
```

- Since CR is already above the liquidation threshold, the second liquidation should **revert**, and only 0.5 BTC should be liquidated.
- Instead, because the locked collateral is improperly counted, the system believes CR is lower than it actually is and allows a second liquidation that should never occur.

Recommendations:

Consider excluding **`liquidation_lock`** **from** **`get_total_collateral_amount()`** - treat it exactly like `collateral_asset_in_flight`, since it is already committed to be removed.

Similarly, corresponding debt should also be reduced from the position and market.

Customer's response: Fixed in be1388a.

Fix Review: Confirmed.

H-08. Underwater Liquidation Leaves Residual Debt

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: borrow.rs	Status: Partially fixed	

Description:

When a fully underwater position is liquidated, the collateral can be exhausted while the borrower still owes a residual amount. The protocol currently leaves that residual as it is in market and position's state: `borrow_asset_borrowed`. Because supplier yield / snapshot math uses market-level `borrow_asset_borrowed`, suppliers continue to accrue yield on debt that is effectively uncollectable - permanently distorting market accounting and supplier yield balances.

Example:

- Collateral = `2 BTC` → `2 * $50,000 = $100,000`
- Liability = `$110,000` (underwater by $10,000)
- After liquidation:
  - Collateral = `0`
  - Liability = `$15,000`

Market still counts `+15,000` in `market.borrow_asset_borrowed`. Interest calculations use this +15,000, so suppliers keep earning yield on debt that won't be repaid.

Even a single such position can skew yield, utilization, available-to-borrow, and supplier withdrawal math for the entire market.

Recommendations:

As a temporary mitigation, consider adding a function with access control that repays the residual debt in exchange for nothing for such positions only, allowing the protocol team or insurance account to clear the bad debt and restore correct accounting.

Ideally for longer-term, if the protocol ever registers bad debt, it must proportionally adjust supplier balances (or use reserves) to keep accounting consistent.

Customer's response: Fixed in 5d17245.

Fix Review: Fixed by allowing repayments to accounts by any other account. However, this is a partial mitigation. There is still a time window between underwater liquidation and the manual repayment where suppliers accrue phantom yield. If snapshots progress before manual clearing, yield is still recorded.

Customer's response: Acknowledged. H-08 maintains correct accounting in the case of "completely underwater" liquidations. While safe MCR parameterization should ensure that liquidators have ample opportunity to liquidate undercollateralized positions, in the unlikely event of late liquidations (e.g. oracle downtime), the fix allows for an insurance fund to make up the deficit.

H-09. Continuous Yield Distribution Without Interest Payment Causes Balance Mismatch and Invalid Liquidity Assumptions

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: impl_market_external.rs	Status: Fixed	

Description:

Under the new continuous-yield distribution model, suppliers accrue yield even when no borrower repayments occur. However, this yield is not backed by actual funds in the contract; it is only reflected in accounting (`borrow_asset_deposited`) if it's compounded. Because of this, the protocol assumes an increased available balance that does not exist on-chain yet.

Example:

- Alice supplies 100 USDC → `borrow_asset_deposited = 100`
- Bob borrows 60 USDC → `borrow_asset_borrowed = 60`
    - Actual contract balance = `40 USDC`
- Few snapshots progress
- Alice harvests `10 USDC` in compound mode
    - New `borrow_asset_deposited = 110`
    - Actual contract balance remains 40 USDC
    - Accounting says available balance = `110 - 60 = 50`, but in reality, the contract only holds 40 USDC

This creates a divergence between accounting liquidity and real liquidity.

Impact:

1. Failed Borrows Despite Passing Checks

- A user attempting to borrow 50 USDC passes all validations, but the transfer fails due to insufficient real funds.
    i. The user pays unnecessary origination fees
    ii. If a snapshot progresses mid-flow, the user pays interest on an in-flight borrow that never completed

2. Failed Withdrawals for Suppliers
    - A supplier withdrawing 50 USDC will have their transaction revert, and their deposit becomes re-activated next snapshot, losing a full snapshot's worth of interest.

3. Broader Systemic Risk
    - This accounting divergence can compound across depositors and snapshots, leading to increasingly inaccurate liquidity assumptions and potentially catastrophic market behavior.

Recommendations:

Consider introducing global yield accounting (e.g., a virtual_supply or equivalent mechanism) so that supplier yield does not immediately inflate borrow_asset_deposited but is represented separately until backed by actual contract balance.

Customer's response: Fixed in b135926.

Fix Review: Confirmed.

H-10. Incorrect Accounting of Borrowable Liquidity Causes Core Invariant Violation

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: impl_market_external.rs	Status: Fixed	

Description:

In the new version, the protocol introduces a new accounting variable `borrow_asset_balance` to track available liquidity. This balance is increased by borrower repayments, liquidation including interest and protocol fees, and decreased on withdrawals and borrows.

As a result, `borrow_asset_balance` now includes:

- Depositor principal
- Borrower-paid interest
- Protocol fees (origination fees)

This creates multiple inconsistencies in the system's accounting model.

Example Scenario

1. User A deposits 100 USDC
    - deposited_active = 100
    - borrow_asset_balance = 100
2. User B borrows 100
    - borrowed = 100
    - borrow_asset_balance = 0
3. User B repays after some time
    - Interest accrued = 10
    - Origination fee = 5
    - Total repaid = 115

        - borrowed = 0
        - borrow_asset_balance = 115
4. User A withdraws 50
    - Withdrawal fee = 5
    - Net withdrawn = 45
        - deposited_active = 50
        - borrow_asset_balance = 115 - 45 = 70
5. User C borrows 70
    - borrowed = 70
    - deposited_active = 50

Impacts:

1. Protocol Fees Become Borrowable Liquidity
    - Protocol-owned funds (origination fees, withdrawal fees, etc.) are added to `borrow_asset_balance` and treated as borrowable liquidity.
    - This allows:
        - Depositors to earn yield on protocol fees they did not supply
        - Borrowers to use protocol revenue as loan liquidity
        - Ideally, protocol fees should be segregated and not exposed to borrower risk.
2. Core Invariant borrowed ≤ deposited_active Is Broken
    - In the example:
        - borrowed = 70
        - deposited_active = 50
    - This leads to:
        - Artificially high utilization
        - Incorrect interest rate calculations
    - The root cause is that *interest paid by borrowers increases borrowable liquidity without increasing depositor principal*.
3. Additionally, depositors continue to accrue yield from interest and fees even if they never call `harvest_yield` or any compounding mechanism.

Recommendations:

The issue was introduced as part of the mitigation for H-09. We recommend removing the `borrow_asset_balance` logic and instead introducing separate yield accounting to properly mitigate H-09.

Customer's response: Fixed in 9b07a05.

Fix Review: The mitigation for H-09 introduced additional logic that led to H-10. However, the yield compounding logic responsible for H-09 has since been removed. Consequently, the logic that enabled H-10 is no longer present, and the issue is no longer applicable.

Medium Severity Issues

M-01. Unsafe CR configurations

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: configuration.rs	Status: Acknowledged	

Description:

The `MarketConfiguration::validate()` function currently checks:

```JavaScript
if self.borrow_mcr_maintenance < 1u32
 || self.borrow_mcr_maintenance < self.borrow_mcr_liquidation
{
    return Err(error::out_of_bounds("borrow_mcr_maintenance"));
}
if self.borrow_mcr_liquidation < 1u32 {
    return Err(error::out_of_bounds("borrow_mcr_liquidation"));
}
```

This allows the following unsafe configurations:

1. `borrow_mcr_liquidation = 1`(100%)
- Liquidation triggers exactly when collateral equals debt.
- Even a tiny price drop, rounding error, or interest accrual can make positions immediately undercollateralized, potentially creating bad debt.

2. `borrow_mcr_liquidation == borrow_mcr_maintenance`
- Eliminates the safety buffer between a healthy and liquidatable position.

- Borrowers can be liquidated immediately after breaching maintenance, leaving no room for corrections.

Recommendations:

Enforce minimum values greater than 1:

- `borrow_mcr_liquidation > 1`
- `borrow_mcr_maintenance > borrow_mcr_liquidation`

Customer's response: Acknowledged. This is a validity check, not a risk check. It is valid to have a market with these configurations. It is up to the user to decide their risk tolerance, not the market deployer.

M-02. `market.collateral_asset_deposited` is not reduced during liquidation

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: borrow.rs	Status: Fixed	

Description:

During liquidation, the borrower's `position.collateral_asset_deposit` is cleared correctly, but the global tracker `market.collateral_asset_deposited` is not reduced.

Currently, this variable is only used for accounting purposes and not in critical calculations. However, leaving it inflated introduces several risks:

- Inflated accounting data: The protocol will show more collateral in the system than what is actually present.
- Storage growth/overflow risk: Over time and across many liquidations, the `market.collateral_asset_deposited` may grow excessively. This could eventually hit the maximum value for the data type and cause new deposits to revert.
- Future dependency risk: If, in future upgrades, this variable is used in calculations (e.g., utilization, health metrics, etc.), incorrect values could affect the protocol.

Recommendations:

Ensure that the `market.collateral_asset_deposited` is reduced proportionally when collateral is seized during liquidation. This keeps global accounting consistent with actual collateral in the system.

```
JavaScript
    pub fn record_full_liquidation(
        &mut self,
        liquidator_id: AccountId,
```

```
        mut recovered_amount: BorrowAssetAmount,
    ) {
        let principal = self.position.get_borrow_asset_principal();
        let collateral_asset_liquidated = self.position.collateral_asset_deposit;

+       asset_op!(self.market.collateral_asset_deposited -= collateral_asset_liquidated);

        MarketEvent::FullLiquidation {
            liquidator_id,
            account_id: self.account_id.clone(),
            borrow_asset_principal: principal,
            borrow_asset_recovered: recovered_amount,
            collateral_asset_liquidated,
        }
        .emit();

        let snapshot_index = self.market.snapshot();
        self.position.full_liquidation(snapshot_index);

        asset_op! {
            @msg("Invariant violation: market borrow_asset_borrowed > position principal")
            self.market.borrow_asset_borrowed -= principal;
        };

        if recovered_amount.split(principal).is_some() {
            // Distribute yield.
            // record_borrow_asset_yield_distribution will take snapshot, no need to do it.
            self.market
                .record_borrow_asset_yield_distribution(recovered_amount);
        } else {
            // Took a loss on liquidation.
            // This can be detected from the event (borrow_asset_principal >
borrow_asset_recovered?).
            // Deficit should be covered by protocol insurance.
            // No need for additional action.
        }
    }
```

Customer's response: Fixed in [fd44011]

Fix Review: Confirmed.

M-03. Missing maximum borrow cap allows whale positions that cannot be liquidated

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: configuration.rs	Status: Fixed	

Description:

The protocol supports a `borrow_range` configuration that defines the minimum and maximum borrowable amount.

However, the maximum (`self.configuration.borrow_range.maximum`) can be left as `None`.

In this case, there is no upper bound on the borrow size.

This introduces a problem due to the lack of partial liquidation support:

1. Protocol is deployed with `borrow_range.maximum = None`.
2. A whale supplies collateral and borrows **50M USDC** in a single position.
3. Later, the position falls below the liquidation threshold.
4. A liquidator would need 50M USDC upfront to liquidate, but:
    - The liquidator's own liquidity is lower than that.
    - No flash loan market has sufficient liquidity.
5. The position remains unliquidated

Recommendations:

Mitigation options:

1. Enforce a maximum borrow cap for every market (ensure `borrow_range.maximum` is always set).
2. Introduce partial liquidation so that even oversized positions can be liquidated in increments.

At least one of these protections must be in place to guarantee that all positions remain liquidatable under all conditions.

Customer's response: Obviated by partial liquidations fd44011.

Fix Review: Confirmed.

M-04. Bad debt not covered by the insurance fund during liquidations

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: borrow.rs	Status: Acknowledged	

Description:

The protocol has a functional insurance fund infrastructure, but fails to automatically utilize it when bad debt occurs during liquidations. When liquidation recovery is insufficient to cover the outstanding debt, the deficit is silently absorbed by suppliers rather than being covered by the protocol's insurance fund.

Current behavior in `record_full_liquidation()`:

```JavaScript
    if recovered_amount.split(principal).is_some() {
        // Profit: distribute excess yield
        self.market.record_borrow_asset_yield_distribution(recovered_amount);
} else {
        // Bad debt detected: just log it in comment
        // This can be detected from the event
        // Deficit should be covered by protocol insurance.
        // No need for additional action.
}
```

Attack scenario:

- Alice borrows 1000 USDC with 1500 USDC collateral
- Collateral value crashes to 600 USDC
- Liquidator pays 570 USDC (600 USDC - 5% liquidation spread)
- 430 USDC bad debt simply vanishes from the system
- Suppliers effectively lose 430 USDC from their collective deposits

Existing infrastructure:

- Insurance fund exists via `static_yield[protocol_account_id]`
- Fund accumulation works via `record_borrow_asset_protocol_yield()`
- Fund withdrawal works via `withdraw_static_yield()`
- Missing: Automatic bad debt coverage during liquidations

Recommendations:

Implement automatic bad debt coverage by utilizing the existing insurance fund infrastructure. This ensures suppliers are protected from bad debt losses as intended, utilizing the insurance fund that already accumulates protocol fees and yields for this exact purpose.

Customer's response: Acknowledged. Insurance is not managed by the contract, but by the owner of the insurance account, which can be configured to earn yield using the yield_weights configuration parameter. Insurance can then be applied as-needed by the owner of the account.

M-05. Assets can be borrowed from borrow positions that are about to expire

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: borrow.rs; configuration.rs	Status: Acknowledged	

Description:

As we know, we have the `borrow_maximum_duration_ms` configuration parameter, which, in combination with the `started_at_block_timestamp_ms` value of each individual borrow position, is used to limit their lifespan. This is enforced through the `is_eligible_for_liquidation` function (where we both check whether the position satisfies the minimum collateralization ratio and the maximum borrow duration), which is checked against on each `borrow` call.

Although because of that, users won't be able to borrow through an expired position, they are still able to do so through one that is very close to being expired. This can lead to the following situation:

1. I have a borrow position with 1000 tokens already borrowed, which is supposed to expire in block 100.
2. I decide to borrow another 1000 tokens from it in block 99 - the borrow operation succeeds
3. In block 100, though, since my position is now expired, my position will now be marked as liquidatable, and it will be liquidated, because position expiration is a non-reversible process.

Another harmful situation to the borrowers might be where they have some leftover dust within their position, so that when they borrow, their position will actually have a

`started_at_block_timestamp_ms` set for it, which in turn will lead to it expiring earlier than it otherwise would.

Recommendations:

Consider either:

- Not allowing the borrowing of assets through positions that are very close to expiring;
- Implementing a mechanism that updates the `started_at_block_timestamp_ms` of positions in some specific scenarios (for example, when the newly borrowed amount is greater than the current `borrow_asset_principal` of the borrow position)

Customer's response: Acknowledged. This is intentional design. This is a matter of risk that is up to the user.

M-06. Immutable `borrow_maximum_duration_ms` might prove to be problematic in some specific cases

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: configuration.rs	Status: Acknowledged	

Description:

In the case where the borrow token contract of a market gets paused (we have pauseable NEP-141 tokens, for example USDT) for whatever reason and it is paused for long enough duration this will lead to a lot of user positions expiring without the users being able to do anything about it - the borrow token is paused, so they can't repay and close their positions. And if they become expired, they also become liquidatable (due to the way that the `borrow_maximum_duration_ms` is enforced), and when they become liquidatable, they can no longer be repaid. And since the `borrow_maximum_duration_ms` configuration parameter can't be changed once a market is deployed, the protocol admins won't be able to do anything about this either. So essentially, if the borrow token contract gets paused for long enough, it will lead to a mass liquidation.

Recommendations: Add permissioned update functionality for the market configuration or at least for its `borrow_maximum_duration_ms` value, so that if such a scenario occurs, the protocol admins can react by increasing or removing the `borrow_maximum_duration_ms` value to avoid negative consequences.

Customer's response: Acknowledged. These are permissionless markets. The likelihood of a token contract getting paused is a risk factor that can be decided by the user.

certora

M-07. Borrowers might be charged extra interest on repayment

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: [impl_token_receiver.rs]	Status: Fixed	

Description:

As can be seen in the `estimate_current_snapshot_interest` function:

```rust
pub fn estimate_current_snapshot_interest(&self) -> BorrowAssetAmount {
    let prev_end_timestamp_ms = self.market.get_last_finalized_snapshot().end_timestamp_ms.0;
    let interest_in_current_snapshot = self.market.current_snapshot.interest_rate()
        * (env::block_timestamp_ms().saturating_sub(prev_end_timestamp_ms))
        * Decimal::from(self.position.get_borrow_asset_principal())
        / *MS_IN_A_YEAR;
    #[allow(clippy::unwrap_used, reason = "Interest rate guaranteed <= APY_LIMIT")]
    interest_in_current_snapshot.to_u128_ceil().unwrap().into()
}
```

We are calculating the interest based on the `borrow_asset_principal` of the borrow position. However, because of that, in the case where a borrower performs multiple partial repayments within a single snapshot, they will actually be charged more interest than they otherwise would be.

To illustrate the issue, let's take the following example scenario, where Borrower A has borrowed 1000 USDC and each |---| block represents a single snapshot:

1. |----------|----------|***---------| - Borrower A repays 500 USDC of his borrow position;

2. Borrower A is charged the estimated interest for 1/4 of the snapshot for the current snapshot on 1000 USDC and is now left with 500 USDC;
3. |----------|----------|******------| Borrower A repays the remaining 500 USDC of his borrow position;
4. Borrower A is charged the estimated interest for 1/2 of a snapshot for the current snapshot on 500 USDC, and his borrow position is now closed. (He should have been charged only for 1/4 of a snapshot, as he already paid the interest for the first 1/4. Basically, the estimated interest should have been calculated only for the following timeline |---***------|)

Recommendations:

To mitigate this issue, you can either:

- Track the last timestamp at which a position has accrued partial interest in a given snapshot and use it to calculate the partial interest from it onwards instead of the end of the last snapshot, in case a borrower repays their position more than once in the span of a single snapshot.
- Redesign the interest accrual mechanism so that there is no need for partial snapshot interest accrual.

Customer's response: Fixed in 0b8095ac.

Fix Review: The fractional snapshot logic has been removed, and with the new behavior, interest is now calculated only on whole snapshots. Based on this update, the issue is resolved.

M-08. Incorrect Liquidity Accounting Temporarily Blocks Legitimate Borrows

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: impl_market_external.rs	Status: Fixed	

Description:

When a supplier executes `execute_next_supply_withdrawal_request`, the contract:

- Decreases `borrow_asset_deposited_active` by the withdrawal amount, and
- Increments `borrow_asset_in_flight`.

While the transfer is pending, `get_borrow_asset_available_to_borrow()` computes capacity as:

```
JavaScript
        pub fn get_borrow_asset_available_to_borrow(&self) -> BorrowAssetAmount {
         // Code

         u128::from(self.borrow_asset_deposited_active)
             .saturating_sub(u128::from(self.borrow_asset_borrowed))
             .saturating_sub(u128::from(self.borrow_asset_in_flight))
             .saturating_sub(must_retain)
             .into()
    }
```

Because the withdrawal amount has already been deducted from `deposited_active`, subtracting it again via `in_flight` underestimates the true available liquidity.

If the withdrawal is taken *from incoming* deposits (i.e., the withdrawal request consumes `borrow_asset_deposited_incoming` rather than active), then those incoming tokens are not present in `deposited_active` at all. Counting the requested withdrawal amount in `in_flight` while the tokens were never part of `deposited_active` causes the `available` to report an incorrect value.

Example
Starting state:

- `borrow_asset_deposited_active = 1000`
- `borrow_asset_borrowed = 500` → contract actually has ~500 usable tokens

User requests withdrawal 300: (before promise resolves)

- `deposited_active` → 700
- `in_flight` → 300

Computed available = 700 - 500 - 300 = 0

Recommendations:

If the requested withdrawal amount is **already subtracted from the position and market state** (`deposited_active`), then updating `in_flight` in `execute_next_supply_withdrawal_request` is redundant and should be removed.

Customer's response: Fixed in 1843476.

Fix Review: Confirmed.

M-09. Borrow / Withdrawal Denial via NEP-141 Storage Opt-Out (In-Flight Inflation Attack)

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: impl_helper.rs	Status: Fixed	

Description:

An attacker can repeatedly initiate borrow operations while opting out of NEP-141 storage. The contract increments `market.borrow_asset_in_flight`, but only records the position principal on successful transfer.

- borrow_01_consume_price updates the `market.borrow_asset_in_flight`

```
JavaScript
borrow_position.record_borrow_asset_in_flight_start(proof, amount, fees);
```

- borrow_position.principal is only updated during the final step `record_borrow_asset_withdrawal`, only if transfer is successful. Otherwise, no changes in the position state, only `borrow_asset_in_flight` is reduced

Because `get_borrow_asset_available_to_borrow()` subtracts `borrow_asset_in_flight`, the attacker can artificially consume available liquidity, blocking borrows and withdrawals, without having a recorded principal or paying interest. The attack is profitable whenever the gas cost of executing the failing transactions is **less than** the interest the attacker saves by depressing utilization on his another legitimate borrow position.

Anyone can call `storage_deposit(account_id, ...)` for any account. A third party could re-register the attacker and make the transfer succeed, but that simply shifts the storage cost to

that third party. The attacker still benefits: they cause someone else to pay the storage deposit, or they can create fresh accounts and repeat the attack.

Recommendations:

Introduce a small, non-refundable **borrow initiation fee** that is charged every time a borrow is attempted, regardless of whether the transfer succeeds. Alternatively, **enforce NEP-141 storage registration upfront** before attempting the transfer, so that borrow transfers cannot systematically fail.

Customer's response: Fixed in 18434761.

Fix Review: The borrower is now charged interest on the in-flight amount as well, which resolves the issue.

M-10. Amortized Interest Not Cleared on Full Repayment Allows Borrowers to Underpay Interest

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Borrow.rs	Status: Fixed	

Description:

When a borrower repays, the contract estimates the running snapshot's interest and adds it to `amortized`. However, if the repayment fully clears the position, the `amortized` field is **not reset to zero**. This stale amortized value carries forward into future borrowings.

Example:

- The borrower has a debt of **100 tokens**.
- After 5 minutes at a rate = 9:
  - Repayment adds `100 × 5 × 9 = 4500` to `amortized`.
  - Amortized = 4500.
- Borrower fully repays the position (debt = 0).
- After 3 snapshots, the same borrower borrows again: **100 tokens**.
- Next snapshot (10 minutes): expected interest = `100 × 10 × 9 = 9000`.
- Instead, only **4500** is added, because the old `amortized` value (4500) is subtracted, resulting in **underpayment of 4500 interest**.

Thus, full repayment leaves behind amortized interest, which benefits the borrower unfairly upon reopening a position. Borrowers can systematically underpay interest if they close and reopen positions, making Suppliers lose yield, and protocol revenue decreases.

Recommendations:

On full repayment (when a position's debt becomes zero), explicitly reset `amortized` to zero.

Customer's response: Fixed in 0b8095a.

Fix Review: Confirmed.

M-11. Borrowers Charged Full Snapshot Interest Even for Late Loans

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: Borrow.rs	Status: Partially Fixed	

Description:

In borrow positions, `next_snapshot_index` is recorded as the index of the **current running snapshot**. During interest calculation, the loop iterates over **finalized snapshots** starting from `next_snapshot_index`. This design means that whenever a borrower opens a loan within a snapshot, whether at the start or just before it finalizes, they are charged interest for the **entire snapshot duration**.

As a result, a borrower who borrows just before a snapshot finalizes pays interest as if they had held the debt for the full length of that snapshot, leading to consistent overcharging.

Recommendations:

In the current design, the most viable fix is to track a borrower-specific timestamp, such as the last borrowed timestamp or the last interest accrued timestamp. This value can then be used to compute partial interest for the elapsed portion of the current snapshot, rather than always charging for the full snapshot.

Customer's response: Partially addressed in 0b8095a.

The market will still err on the side of overcharging by adding an origination fee to each loan equal to the maximum amount of interest that can be charged for the duration of one snapshot. For example, if the maximum interest rate that can be charged by the market is 25%/year, and the snapshot duration is 10 minutes, a $10k loan would incur a ~$0.048 origination fee. This covers the amount of interest that could be lost by abusing loan repayment timing.

Fix Review: The new behavior calculates interest only over whole snapshots. Since the updated logic doesn't charge partial-snapshot interest for the snapshot in which repayment occurs, the extra interest the borrower pays for the full snapshot at the time of borrowing helps offset the discrepancy. Combined with the origination fee, this partially addresses the issue.

Customer's response: Acknowledged. M-11 solution's mandatory origination fee will primarily affect extremely short-term positions (e.g. 30-second borrows), and ensures that even if repayment timing is gamed, the suppliers still receive at least a fair payout.

M-12. Incorrect Interest Calculation After Partial Repayment

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: Borrow.rs	Status: Fixed	

Description:

The protocol accrues interest in two steps:

1. **During repayment,** it estimates interest as:

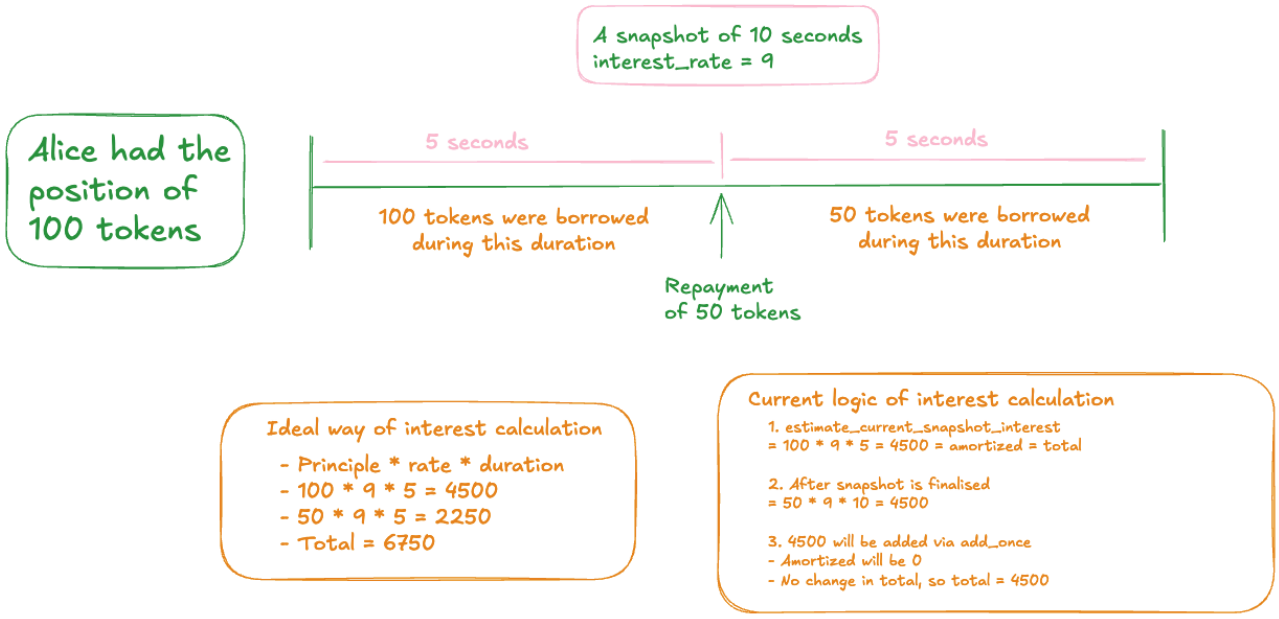
```
None
estimate = elapsed_time * amount₁ * rate
```

   and adds this to the `total` and `amortized`.

2. **At snapshot finalization,** it recomputes interest for the entire snapshot using the reduced borrowed amount (`amount₂`) and applies:

```
None
total += (value - amortized)
```

This works if `amount₂ == amount₁`. However, when a repayment reduces the borrowed amount mid-snapshot, the `amount₂ will be less than amount₁`. The issue is that **the "surplus cancellation" at finalization overwrites the earlier accrual**, effectively treating the smaller post-repayment amount as if it had been borrowed for the whole snapshot duration, leading to underpayment.



Example (from diagram):

- Alice borrows **100 tokens** for 5 seconds, repays 50, then holds **50 tokens** for another 5 seconds, with a rate = 9.
- **Ideal interest:**
  - 100 × 9 × 5 = 4500
  - 50 × 9 × 5 = 2250
  - **Total = 6750**
- **Current logic:**
  - Accrues 4500 during repayment.
  - At finalization: 50 × 9 × 10 = 4500.
  - Net result = 4500 (instead of 6750).
- Suppliers lose **2250 units of interest**, and borrowers benefit by repaying early.

Recommendations:

While there may not be a simple one-line fix, the correct approach is to adjust how accrual is handled:

- Do not recompute interest for the full snapshot using only the reduced principal.
- Instead, **divide accruals into segments** whenever a repayment occurs:
    - First, settle the interest accrued up to the repayment point based on the original borrowed amount.
    - Then, continue accruing interest from that point forward using the updated (reduced) principal until snapshot finalization.

Customer's response: Fixed in 0b8095a.

Fix Review: The new behavior calculates interest only over whole snapshots, effectively eliminating the issue.

M-13. Delayed Activation of Deposits Causes Unfair Yield Accrual

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files:	Status: Fixed	

Description:

Deposits made during a time chunk are expected to become active in the next chunk. However, due to how activation is indexed, funds are only activated one snapshot later than intended.

When a user supplies funds, the deposit is scheduled to activate at `finalized_snapshots.len() + 1`. This is recorded both in the individual position and in `market.borrow_asset_deposited_incoming`. However, during snapshot finalization, the contract fetches deposits using `self.finalized_snapshots.len()`

```
JavaScript
let deposited_incoming = self
    .borrow_asset_deposited_incoming
    .remove(&self.finalized_snapshots.len())
    .unwrap_or(0.into());
asset_op!(self.borrow_asset_deposited_active += deposited_incoming);
```

This mismatch introduces a one-snapshot delay.

Example:

1. `finalized_snapshots.len() == 1`
    - Snapshot 0 is finalized, snapshot 1 is active.
2. A user deposits 100 tokens during snapshot 1.

   - Recorded as `incoming(2) = 100`.
3. At finalization, the protocol fetches `incoming(1)` (not `incoming(2)`).
   - Result: the 100 tokens do not activate in snapshot 2.
   - Instead, they only activate in snapshot 3.

So while snapshot 2 is running, those funds:

- Are **not available for borrowers to borrow**, but still continue accruing the yield during calculate_yield despite never contributing to borrowable funds.

Recommendations:

Fix the activation logic to ensure that deposits made during snapshot *N* become active in snapshot *N+1*, not *N+2*.

Customer's response: Fixed in 14dd04c.

Fix Review: Confirmed.

M-14. Under-liquidation when full liquidation is required

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: borrow.rs	Status: Fixed	

Description:

The liquidatable_collateral function may return a collateral amount smaller than what is actually required. In some scenarios the only valid solution is full liquidation, but the current logic instead produces a partial liquidation amount, leaving the position still undercollateralized.

Example:

- Collateral = 2.0 BTC (value = $100,000)
- Liability = $94,000
- Price = $50,000 / BTC
- Spread = 0.05
- MCR = 1.2

```JavaScript
scaled_liability = L / P = 94,000 / 50,000 = 1.88
scaled_collateral = C * (1 - s) / m = 2 * 0.95 / 1.2 = 1.5833333...
unscaled = (1.88 - 1.5833333) / (m - 1) = 0.2966667 / 0.2 = 1.4833333
amount = m * unscaled = 1.2 * 1.4833333 = 1.78 BTC
```

- Net recovered USD = `1.78 * 50,000 * 0.95 = 1.78 * 47,500 = $84,550`

- Post-liquidation:
  - Liability = 94,000 - 84,550 = $9,450
  - Collateral = 2 - 1.78 = 0.22 BTC → value = $11,000
  - Post-CR = 11,000 / 9,450 ≈ 1.164 < 1.2 → position still unhealthy

Recommendations:

Ideally in such a scenario all collateral should be liquidated and liquidatable_collateral should handle that case.

Customer's response: Fixed in 704c7dd.

Fix Review: Confirmed.

M-15. Delayed Liability Reduction Causes Temporarily Inflated Borrowed Amount

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: impl_helper.rs	Status: Fixed	

Description:

In the current liquidation flow, the liquidator repays the borrower's debt by transferring the borrow asset to the protocol, but the borrower's liability is only reduced during liquidate_transfer_call_02_finalize. This is incorrect because the liquidator never gets their borrow asset back, regardless of whether the collateral transfer later succeeds or fails. The liability reduction should happen immediately at liquidate_transfer_call_01_consume_price, at the moment the liquidator's funds are received.

As implemented, during the window between `liquidate_transfer_call_01_consume_price` and `liquidate_transfer_call_02_finalize`, both the position and the market incorrectly reflect an inflated `borrow_asset_borrowed` value (which feeds into the borrowed() function).

```JavaScript
pub fn borrowed(&self) -> BorrowAssetAmount {
    self.borrow_asset_borrowed + self.borrow_asset_borrowed_in_flight
}
```

This leads to several issues:

1. **Incorrect interest rate after snapshot progression:**

- If the market advances its snapshot during this window, it uses the incorrect `borrow_asset_borrowed` amount, leading to an inaccurate interest rate update and incorrect yield calculation in `calculate_yield`.

2. **Incorrect available liquidity to borrow:**
   - `get_borrow_asset_available_to_borrow` will return a lower available amount, potentially rejecting borrows even when liquidity is actually available.

3. **Incorrect withdrawal availability:**
   - `record_withdrawal_initial.generally_available` will also be computed with outdated borrow values, causing the contract to deny withdrawals despite having sufficient liquidity.

Recommendations:

Consider reducing the borrower's liability **immediately** at `liquidate_transfer_call_01_consume_price` when the liquidator's funds are received.

Customer's response: Fixed in 22a715a.

Fix Review: Confirmed.

M-16. DoS on liquidations

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: src/borrow.rs	Status: Fixed	

Description:

In this PR: https://github.com/Templar-Protocol/contracts/pull/307/files the denominator changed from `(mcr - 1)` to `(mcr * discount - 1)`

The new denominator `(mcr * discount - Decimal::ONE)` can underflow when:

`mcr * (1 - liquidator_spread) < 1`

Example: `mcr = 1.10, liquidator_spread = 0.10`

It can also be exactly zero when `mcr * (1 - liquidator_spread) == 1,` causing division by zero

The impact is DoS on liquidations.

Recommendations:

Ensure that during deployment and configuration updates, the liquidation formula cannot underflow or cause division by zero:

```
JavaScript
assert!(
    self.borrow_mcr_liquidation * (Decimal::ONE - self.liquidation_maximum_spread) >
Decimal::ONE,
```

```
    "Invalid parameters: MCR * (1 - spread) must be > 1"
);
```

Customer's response: Fixed in d8ccc3c.

Fix Review: Confirmed.

Low Severity Issues

L-01. Big withdrawal requests might clog the withdrawal queue

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: impl_market_external.rs	Status: Fixed	

Description:

The current implementation of the withdrawal queue works in a FIFO manner. The withdrawal handling logic itself handles withdrawal requests in a "binary" manner - It either can or it can not fulfil the next request in the queue. If it can, it will transfer the requested amount to the supplier, make the necessary changes to the state, and remove the request from the queue. If it can not, the transfer will fail, the contract state will be reverted, and the failed withdrawal request will remain in the queue.

However, this could lead to the following issue. Let's say that we have:

- 2500 active borrow tokens
- 2000 borrowed tokens

Our available balance will be 500 (2500 - 2000). Now, in that scenario, let's imagine that supplier A sends a withdrawal request for 2000 tokens. After him, suppliers B, C, and D also sent withdrawal requests for 50, 100, and 100 tokens, respectively. And now, since the withdrawal request of supplier A was sent first, it has to be executed before all of the other requests. And since the market doesn't have enough available balance to fulfill it, we won't be able to execute it until it does, essentially meaning that it will just stay there and clog/DoS the withdrawal queue, preventing the other users from executing their otherwise fulfillable requests.

Recommendations:

Instead of reverting when the next withdrawal request in the queue can't be fulfilled, transfer the available balance to the supplier and then subtract it from the request state. That way, big request will be easier to process.

Customer's response: Fixed in 94c6014.

Fix Review: Confirmed.

L-02. Storage is not being cleaned up after liquidation

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: borrow.rs	Status: Fixed	

Description:

At the current state of the codebase, the state of borrow positions is cleaned up, and the storage amounts for them are being refunded when they are fully repaid and their collateral is withdrawn. The same is also done for supply positions when they are fully emptied by an executed withdrawal request. However, the same is not done for borrow positions when they are liquidated. Instead, they are just zeroed out by this call to BorrowPosition::full_liquidation. This pretty much has the same effect as disposing of that state, but without refunding the storage amounts.

Recommendations:

Clean up the state of borrow positions and refund the storage amounts attached to them on liquidation, just like it is done for full collateral withdrawals.

Customer's response: Fixed in fac1d49.

Fix Review: Confirmed.

L-03. Withdrawal-fee bypass by leaving a 1-token

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Supply.rs	Status: Acknowledged	

Description:

The supply-withdrawal fee is calculated from how long a *position* has been active, using `position.started_at_block_timestamp_ms`. The fee function is roughly:

- `supply_duration = now - position.started_at_block_timestamp_ms`
- `fee = supply_withdrawal_fee.of(amount, supply_duration)`

Because the contract keeps `started_at_block_timestamp_ms` on the *position* (not on per-deposit slices), a user can keep that timestamp artificially old by **never fully closing the position**. In practice, an attacker can:

1. Withdraw nearly all their funds but **leave 1 token** in the position (the position remains "alive" and the `started_at_block_timestamp_ms` is not reset).
2. Later, when they supply again, the contract treats the new supply as if it continued from the original `started_at` time (no penalty), because the timestamp never resets.
3. Future withdrawals of the newly supplied funds see a long `supply_duration` (or sometimes zero/low fee depending on fee logic), allowing the user to avoid (or minimize) withdrawal fees.

Recommendations:

Perform `supply_position.is_within_allowable_range()` during supply withdrawals.

Customer's response: Acknowledged. This is intentional design; the withdrawal fee is only intended to apply to newly-created positions.

L-04. Collateral Loss Due to Double Withdrawal and Premature Position Removal

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: impl_helper.rs	Status: Fixed	

Description:

In the `withdraw_collateral` flow, the contract decreases its `position.collateral_asset_deposit` immediately and creates transfer promises. When multiple withdrawals are requested within the same block, this can result in an inconsistent state.

```JavaScript
    #[private]
    pub fn withdraw_collateral_02_finalize(
        &mut self,
        account_id: AccountId,
        amount: CollateralAssetAmount,
    ) {


        if transfer_was_successful {
            if self.cleanup_borrow_position(&account_id) {
                self.refund_for_storage(&account_id, self.storage_usage_borrow_position);
            }
        } else {
        }
    }
```

Example:

- Alice has **$100 in collateral** and no liabilities.

- She calls `withdraw_collateral(50)` twice in the same block.
    - **First call:** `collateral_asset_deposit` is reduced to 50, and a transfer promise of 50 is created.
    - **Second call:** `collateral_asset_deposit` is reduced to 0, and another transfer promise of 50 is created.
- If the **first transfer succeeds**, `withdraw_collateral_02_finalize` calls `cleanup_borrow_position`, removing the position since collateral = 0 and liability = 0.
- If the **second transfer fails**, the `else` branch in `withdraw_collateral_02_finalize` attempts to fetch the now-removed position and reverts.
- Result: Alice permanently loses the **50 tokens** from the failed transfer attempt.

Therefore, Users can lose collateral if multiple withdrawals are submitted in the same block and one transfer fails.

Recommendations:

Introduce a withdrawal lock so that users cannot initiate another withdrawal while a previous withdrawal transaction is still pending.

Customer's response: Fixed in 2610307.

Fix Review: Confirmed.

L-05. Low minimum withdrawal limits can lead to temporary withdrawal DoS

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: impl_market_external.rs	Status: Acknowledged	

Description:

The protocol allows configuring `supply_withdrawal_range.minimum` to very low values (including 0), which enables a potential DoS attack against the withdrawal queue mechanism. An attacker could create numerous accounts and submit minimal withdrawal requests to clog the FIFO withdrawal queue, significantly delaying legitimate withdrawals.

Attack scenario:

- Market deployed with `supply_withdrawal_range.minimum = 0` or very low value
- The attacker creates multiple accounts and supplies minimal amounts to each
- Each account submits withdrawal requests for the minimum amount (1 wei for 18-decimal tokens)
- Legitimate users' withdrawal requests are queued behind thousands of micro-withdrawals

Processing the queue requires manual execution of each request via `execute_next_supply_withdrawal_request()`

Recommendations:

Enforce reasonable minimum withdrawal amounts during market configuration to prevent economic DoS attacks

Customer's response: Acknowledged. It is impossible to generally define a "reasonable minimum withdrawal amount."

Informational Issues

I-01. Suppliers Do Not Receive Yield from Current Running Snapshot on Withdrawal

Description:

When a supplier withdraws, yield is calculated only from **finalized snapshots**. Any interest accrued during the **current running snapshot** is excluded from the user's credited yield. In other words, suppliers only receive yield up to the last finalized snapshot, even though their funds have been active during the ongoing snapshot.

Recommendations: On withdrawal, also account for interest accrued in the current snapshot, proportional to the elapsed time and active deposits.

Customer's response: Acknowledged. The unit of granularity upon which the contract operates is the snapshot. The amount of yield lost is partly made up for by the mandatory origination fee introduced in Templar-Protocol/contracts#228.

I-02. Denial of Service on Liquidations via Locking Mechanism and Storage Opt-Out

Description:

The protocol uses a `liquidation_lock` flag to prevent double liquidation on a borrow position. The flag is set to `true` at the start of liquidation and reset to `false` in `execute_liquidate_final`. However, because users can opt out of storage management, a malicious borrower can repeatedly trigger liquidation attempts through secondary accounts. Each attempt reverts during the collateral transfer (due to unpaid storage), leaving the position locked temporarily. By looping this process with multiple accounts, an attacker can block liquidation of their own undercollateralized position.

```
JavaScript
       pub fn execute_liquidate_final(
        &mut self,
        liquidator_id: AccountId,
        account_id: AccountId,
        amount: BorrowAssetAmount,
        success: bool,
    ) -> BorrowAssetAmount {


        if success {
            borrow_position.record_full_liquidation(liquidator_id, amount);
            BorrowAssetAmount::zero()
        } else {
            // ...
            borrow_position.liquidation_unlock();
            amount
        }
    }
```

Recommendations:

Prevent accounts that have opted out of storage management from initiating liquidations.

Customer's response: Fixed in b39f96e.

Fix Review: Confirmed.

I-03. Snapshot not updated when `market.collateral_asset_deposited` is updated

Description:

In `record_collateral_asset_deposit`, the per-position collateral is increased and `market.collateral_asset_deposited` is incremented, but **the current snapshot is not updated at the same time**. Same with the `record_collateral_asset_withdrawal`

That breaks the expectation that the market-level state and the snapshot are kept in sync: the snapshot still reflects the old value while the market shows the new collateral amount.

If anyone fetches snapshot values before the snapshot is updated, it will be outdated

JavaScript
```
    pub fn record_collateral_asset_deposit(
        &mut self,
        _proof: InterestAccumulationProof,
        amount: CollateralAssetAmount,
    ) {

        asset_op!(self.market.collateral_asset_deposited += amount);

        MarketEvent::CollateralDeposited {
            account_id: self.account_id.clone(),
            collateral_asset_amount: amount,
        }
        .emit();
    }
```

JavaScript
```
    pub fn record_collateral_asset_withdrawal(
        &mut self,
        _proof: InterestAccumulationProof,
        amount: CollateralAssetAmount,
    ) {

        asset_op!(self.market.collateral_asset_deposited -= amount);

        MarketEvent::CollateralWithdrawn {
            account_id: self.account_id.clone(),
            collateral_asset_amount: amount,
        }
        .emit();
    }
```

Recommendations:

Call market.snapshot() to update the snapshot

Customer's response: Fixed in 5f1abbb.

Fix Review: Confirmed.

I-04. Unaccounted Current Snapshot Interest During Liquidation

Description:

When a position is liquidated, the protocol fails to include accrued interest from the *ongoing snapshot*. This means the borrower's debt is understated at liquidation time. By contrast, the repayment flow correctly handles this by calling `estimate_current_snapshot_interest`, ensuring that interest accumulated up to that block is accounted for. The liquidation path should apply the same logic, but currently does not, making the Borrower debt undervalued at liquidation..

Recommendations: Update the liquidation flow to explicitly include the current snapshot interest before proceeding with the health check

Customer's response: Fixed in 0b8095a.

Fix Review: Confirmed.

I-05. Incorrect value of deposited_incoming assigned to new snapshot

Description:

When finalizing a snapshot, the code removes `deposited_incoming` for the finalized index and immediately adds it to `borrow_asset_deposited_active`. Right after that it creates the new `Snapshot` and **also sets** `snapshot.borrow_asset_deposited_incoming = deposited_incoming`.

That is incorrect: `borrow_asset_deposited_incoming` on a *new/current* snapshot should represent deposits that will be activated in a *future* finalization, not the amount that was just activated. By re-using the `deposited_incoming` value here, the same deposit gets represented twice (as active and as incoming)

Example:

A deposit of `1_500_000` happens while snapshot of index `2` is running (so it should become active at snapshot index `3`):

```
None
Snapshot 3:

Snapshot { time_chunk: TimeChunk(U64(586014394)),

borrow_asset_deposited_active: FungibleAssetAmount { amount: U128(0),

borrow_asset_deposited_incoming:      FungibleAssetAmount      {      amount:
U128(1500000)}



Snapshot 4: Snapshot { time_chunk: TimeChunk(U64(586014395)),

borrow_asset_deposited_active: FungibleAssetAmount { amount: U128(1500000),

borrow_asset_deposited_incoming:      FungibleAssetAmount      {      amount:
U128(1500000)}
```

Recommendations: Assign the new snapshot.borrow_asset_deposited_incoming with the amount that is actually going to be activated on that index.

Customer's response: Fixed in 14dd04c.

Fix Review: Confirmed.

I-06. Precision Loss in Time-Based Fee Calculation (Division Before Multiplication)

Description:

In the implementation of `TimeBasedFee::of`, the Linear fee function computes the time-weighted fee as:

```rust
(Decimal::from(self.duration.0.saturating_sub(duration))
 / Decimal::from(self.duration.0)
 * u128::from(base_fee))
```

Because the division is performed before the multiplication, fractional precision is truncated early, leading to an underestimation of the fee. This results in systematic precision loss, particularly when `base_fee` is large and the ratio is not an integer fraction.

Recommendations: Reorder operations so that multiplication occurs before division, preserving fractional accuracy

Customer's response: Fixed in 8a6ca1e.

Fix Review: Confirmed.

I-07. Redundant MCR Checks in Collateral Withdrawal Validation

Description:

In `MarketConfiguration.validate`, the protocol enforces:

```rust
if self.borrow_mcr_maintenance < 1u32
 || self.borrow_mcr_maintenance < self.borrow_mcr_liquidation
{
    return Err(error::out_of_bounds("borrow_mcr_maintenance"));
```

```
}
```

This guarantees `borrow_mcr_maintenance >= borrow_mcr_liquidation`. However, in `withdraw_collateral_01_consume_price`, both conditions are checked:

```rust
require!(
    borrow_position.satisfies_mcr_liquidation(&price_pair),
    "Must satisfy liquidation MCR."
);

require!(
    borrow_position.satisfies_mcr_maintenance(&price_pair),
    "Must satisfy maintenance MCR."
);
```

Since `borrow_mcr_maintenance >= borrow_mcr_liquidation`, passing the maintenance MCR check already implies passing the liquidation MCR check. Thus, one of these checks is redundant.

Recommendations: Retain only the stricter check (`satisfies_mcr_maintenance`) during withdrawal validation.

Customer's response: Acknowledged. We want to emit different error messages e.g. if the liquidation MCR is satisfied but the maintenance MCR is not vs. if they both are satisfied.

I-08. Lack of slippage protection for liquidations

Description:

There is no slippage protection when liquidating a given borrow position. We also do not refund excess borrowed assets to the liquidator on a successful liquidation. Because of those two things, what could happen is the following situation:

1. The 1000 USDC collateral position of borrower A becomes liquidatable.
2. Liquidator X initiates a liquidation transaction for the position of A, with 950 USDC worth of borrow assets.
3. Liquidator Y also initiates a liquidation transaction for the position of A, with 950 USDC worth of borrow assets.
4. The transaction of Y gets executed, and the transaction of X is left sitting in the mempool (as we know, we have a mempool on NEAR);
5. Borrower A borrows again, but this time a much smaller amount that only requires 100 USDC of collateral.
6. The position of A becomes liquidatable.
7. The transaction of X now gets executed, and he pays 950 USDC worth of borrow assets for 100 USDC collateral.

The incentive to take advantage of this issue is quite high, since all of the excess borrowed assets that get transferred by the liquidator will be distributed to the market as yield, resulting in a profit for all suppliers. And although, as of the time of the writing of this report, there is no MEV infrastructure developed for NEAR, which would make this issue much harder to abuse, as time goes on and the chain keeps on growing in adoption, this might change in the not so distant future.

Recommendations:

Add a slippage protection parameter (minAmountOut) that can be passed to `ft/mt_on_transfer` on liquidation initiation.

Customer's response: Fixed in 32dd4a8.

Fix Review: Confirmed.

certora

Formal Verification Report

Project Scope

Project Name	Repository	Commit Hash	Platform
Templar	https://github.com/Templar-Protocol/contracts	1d736e62	NEAR

Project Overview

This section describes the specification and verification of **Templar** using the Certora Prover. As a pioneering pilot in formal verification on the **NEAR blockchain**, it showcases innovative methods and key insights. The work was undertaken from **August 26th** to **October 22nd.**

We wrote formal specifications to partially verify the following components: **contract/market.**

As part of this, we also included all relevant parts from the common crate.

This new formal verification tool developed by Certora demonstrated that the implementations of the NEAR contracts above are correct with respect to the formal rules written by Certora.

Verification Methodology

We developed a new Certora Prover tailored to the NEAR blockchain, focusing specifically on Templar. This project marked our first practical application of the tool, aimed at demonstrating how to formally verify NEAR protocols through the real example of **Templar**. During this effort, we developed new analyses to help scale Satisfiability Modulo Theories (SMT) based verification to real-world NEAR protocols like Templar.

At a high level, the Certora Prover compiles and verifies formal specifications written in the Certora Verification Language (CVLR) and Templar's implementation source code written in Rust. More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVLR holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

Configuration

We used version `1.85.0` of the Rust compiler to generate the WASM bytecode that we verified. We rely on Rust's conditional compilation feature to selectively disable or enable code to ease verification. We provide scripts (`justfile`, `certora_build_integrity_rules.py`, `certora_build_health_rules.py`, `certora_build_other_rules.py` in `contract/specs/`) to help compile the code for formal verification.

NEAR Storage

On the NEAR blockchain, storage accesses are generally abstracted in two ways:

- Automatic serialization/deserialization of contract struct fields
- NEAR-SDK collections (e.g., some collections provide a "lazy" interface to storage)

To simplify reasoning, we treat these lazy collections as if they loaded storage *eagerly*, that is, we assume that each collection is the sole owner of any storage location it may reference.

Summarization

a. **Nondeterministic values.** In many places we replace difficult computations with a nondeterministic result (often of the form `TemplarNondet::nondet()`), which overapproximates the behavior of the original code. Note that any proof of an overapproximated representation is also a proof of the original implementation.
b. **Collection models.** In general, collections such as `Vectors` and `HashMaps` complicate automated reasoning as their sizes are unbounded and can grow and shrink dynamically as objects are added and removed. We have introduced nondeterministic but fixed size *models* of the relevant data structures in `common/src/models`, providing a sound overapproximation. Please reference the module documentation for details.

Formal Verification Properties Overview

ID	Title	Impact	Status
P-01	User health	When the user modifies a borrow position (via borrowing or withdrawing collateral), the position is guaranteed to satisfy the minimum collateralization ratio (MCR for maintenance). That is, a user cannot make a healthy borrow position unhealthy. This rule prevents users from withdrawing too much collateral (or borrowing too much) such that their position becomes dangerously under-collateralized in one step and ensures that the protocol's markets remain solvent by means of user operations and that liquidation risk is minimized.	Verified
P-02	General Integrity	Key functions in the protocol affect the state correctly. When violated, some functions behave unexpectedly and can potentially be exploited.	Verified
P-03	No double borrow	This property checks that concurrent borrow operations are not allowed to bypass the maximum borrow range limits.	Verified
P-04	Collateralize preservation non-liquidatability	Checks that successful collateralization guarantees that the borrow_position is not in a liquidatable state.	Verified
P-05	Collateralization integrity	These rules check the integrity of collateralization in terms of fees and deposited assets (no loss or overwrite).	Verified

P-06	Liquidation integrity	These properties check that the liquidation functionalities (both initial and final) correctly set and release liquidation locks and updates borrow position and market state.	Verified
P-07	Repay safety	Checks the correctness and integrity of reduce_borrow_asset_liability which is the core functionality that is used for liability reduction when recording a repayment.	Verified
P-08	Split repay correctness	We check that fee accounting is split-invariant and repaying principal in parts can never reduce it by more than when paying together.	Verified

Formal Verification Detailed Properties: Market

Module Properties

1. **Borrow position health**
   We check that when the user modifies a borrow position (via borrowing or withdrawing collateral), the position is guaranteed to satisfy MCR for maintenance, ensuring a healthy loan-to-value ratio.
   In the borrow case, we verify that the `borrow_01_consume_price` callback guarantees the referenced position satisfies MCR maintenance requirements when it returns.
   In the withdraw collateral case, we check the analogous `withdraw_collateral_01_consume_price` callback. In both cases, we refactor the callbacks to avoid reasoning about the returned promises.

   To verify these properties, we modified the `BorrowPosition` data structure to include an additional field that tracks the last-known `PricePair` for which the position is known to satisfy the MCR property (see the documentation on `BorrowPosition`).

P-01. Borrow Position Health

Status: Verified	These rules check that every successful state changing action involving collateral or debt maintains a state that satisfies the MCR maintenance criteria.

Rule Name	Status	Description	Link to rule report
borrow_preserves_health	Verified	*Verifies that if `borrow_01_consume_price` succeeds, the referenced borrow position satisfies MCR maintenance*	*Report*

2. **General Integrity checks**
   We check that key functionality of the Templar protocol modifies state correctly.

P-02. General Integrity Properties

Status: Verified	Integrity rules act as **consistency invariants**. They ensure that core accounting operation like yield accrual, interest accumulation, and snapshot updates preserves essential conservation laws: • **No unintended loss or creation of value** • **Monotonic growth of cumulative quantities** • **Consistency between related data structures** (market state, snapshots)

Rule Name	Status	Description	Link to rule report
record_borrow_asset_protocol_yield_integrity	Verified	*Checks that when the protocol's borrow-asset yield is recorded, it increases exactly by the amount passed to* `record_borrow_asset_protocol_yield`. *This checks that protocol's total yield accounting is additive and there is no loss or double counting of yield when recording protocol-level earnings.*	*Report*
record_borrow_asset_yield_distribution_integrity_1	Verified	*Asserts that distributing borrow-asset yield never decreases an account's static yield entry. Yield distribution is monotonic: after a distribution, each account's accumulated yield should stay the same or increase, never decrease due to bookkeeping or logic errors.*	*Report*

record_borrow_asset_yield_distribution_integrity_2	Verified	*Ensures that if an account's yield increases during distribution, it must have a non-zero yield weight.*	*Report*
accumulate_interest_integrity_1	Verified	*Checks that after accumulating interest on a BorrowPosition, the total accrued fees do not decrease.*	*Report*
snapshot_with_yield_distribution_integrity	Verified	*Verifies that yield distribution recorded in a market snapshot behaves consistently across time chunks: If a new time chunk begins, the snapshot's yield distribution resets to the new `amount`. Otherwise, it aggregates (`join`s) the new amount with the previous distribution.*	*Report*

3. **No double borrow**
   This property checks that concurrent borrow operations are not allowed to bypass the maximum borrow range limits. This property was violated due to issue H-04.

P-03. Double Borrow Not Possible	
Status: Violated	The `borrow()` function validates borrow amounts against `borrow_range.maximum` using only the current `borrow_asset_principal`, ignoring pending in-flight borrows tracked in `temporary_lock`.

Rule Name	Status	Description	Link to rule report
double_borrow_fails	Violated	*No borrow position can ever borrow more than maximum borrow amount*	*Report*

After the fix of the **H-04** we reran the rule corresponding to `P-03 (Double Borrow Not Possible)` on the changed version of the protocol. We took the code from commit `ad6e045e94638b0fef2511a3356a4e73c62f4d71` of the `same repository as before`.

P-03. Double Borrow Not Possible

Status: Verified	if a first borrow succeeds, then any second borrow that would make the total principal exceed the configured maximum must fail.

Rule Name	Status	Description	Link to rule report
double_borrow_fails	Verified	*No borrow position can ever borrow more than maximum borrow amount*	*Report*

Note: For this rerun, we had to make some changes to the code to make this rule work. Specifically, the complex nature of the `Market` and `MarketConfiguration` structs combined with changes to the `rustc` version led to timeouts. To mitigate them, we simplified these structs to only keep the fields needed for the property and made any additional (minimal) changes needed accordingly. We did not change the `BorrowPosition` struct itself.
The simpler version of the code is here:
https://github.com/Certora/templar-contracts/tree/branch-ad6e045/contract/specs/src.

4. **Collateralize preserves liquidation state**
   We check that the collateralization function guarantees that the borrow position is not in a liquidatable state. We avoid reasoning about parts of the code that charges for storage. To verify this property, we modified the `BorrowPosition` data structure (similar to our changes for the health property) to include an additional field that tracks the last-known `PricePair` for which the position is known to not be liquidatable (`price_pair_liquidation_eligible`).
   NOTE: This property works under the assumption that `increase_collateral_asset_deposit` cannot make a borrow position liquidatable.

P-04. **Collateralize preserves liquidation state**		
Status: Verified	Checks that collateralization guarantees that the borrow_position is not in a liquidatable state.	

Rule Name	Status	Description	Link to rule report
collateralize_preserves_liquidation_state	Verified	*Verifies that if `execute_collateralize` succeeds, then the borrow position is not eligible for liquidation*	*Report*

5. Integrity and monotonicity of collateralization

These properties check that collateralization correctly updates borrow asset fees and collateral asset deposits. Here, we again avoid reasoning about parts of the code that charges for storage.

P-05. **Integrity and monotonicity of collateralization**		
Status: Verified	These rules check the integrity of collateralization in terms of fees and deposited assets (no loss or overwrite).	

Rule Name	Status	Description	Link to rule report
collateralize_collateral_increase_monotonicity	Verified	*Verifies that calling `execute_collateralize` correctly increases the borrow position's collateral by exactly the deposited amount.*	*Report*
collateralize_interest_accumulation_consistency	Verified	*Ensures that executing `collateralize` never decreases borrow asset fees.*	*Report*

6. Integrity of liquidation

These properties check that the liquidation functionalities (both initial and final) correctly set and release liquidation locks and updates borrow position and market state.
We avoided reasoning about parts of the code that charges for storage. We also use `borrow_position_guard` instead of `get_or_create_borrow_position_guard` due to path explosion. We also used a sound `nondet` summary for minimum liquidation amount.

P-06. **Integrity of liquidation**

Status: Verified	These properties verify the following state transitions and invariants of the liquidation process: • Initialization locks the position. • Successful liquidation clears the debt. • Failed liquidation restores state and refunds. • Full liquidation updates global and per-account balances accurately.

Rule Name	Status	Description	Link to rule report

liquidate_initial_sets_lock	Verified	*Verifies that calling `execute_liquidate_initial` correctly sets the liquidation lock on a borrow position.*	*Report*
liquidation_final_success_clears	Verified	*Ensures that a successful liquidation finalization (`success = true`) returns a zero refund amount.*	*Report*
liquidation_final_fail_unlocks	Verified	*Checks that when liquidation finalization fails (`success = false`), the position's liquidation lock is released and the liquidator receives a full refund of the attempted liquidation amount.*	*Report*
full_liquidation_updates_correctly	Verified	*Checks that executing a full liquidation properly updates key parts of the position and market state.*	*Report*

7. Integrity of repay

We check the correctness and integrity of `reduce_borrow_asset_liability` which is the core functionality that is used for liability reduction when recording a repayment.

P-07. Integrity of reduce_borrow_asset_liability

Status: Verified	These properties verify correct computations of fees, correct reduction of principal, checks bounds, and ensures that `reduce_borrow_asset_liability` leaves the position settled or unchanged in all edge cases.	

Rule Name	Status	Description	Link to

			rule report
repay_liability_reduction_reverts_correctly	Verified	*Ensures that attempting to reduce liabilities while a borrow position is liquidation-locked correctly reverts and leaves the position unchanged.*	*Report*
repay_amount_to_fees_value	Verified	*Verifies that repayments are applied to outstanding fees first and that the decrease in total fees exactly matches the portion of the payment allocated to fees, ensuring consistent fee accounting.*	*Report*
repay_principal_decreases_exactly	Verified	*Checks that the repayment amount applied to principal reduces the outstanding principal by exactly that amount and never increases it, maintaining monotonic principal reduction.*	*Report*
repay_zero_principal_behavior	Verified	*Confirms that when the principal is already zero, a repayment does not allocate any portion to principal, avoiding unintended balance changes on fully repaid positions.*	*Report*
repay_components_bounded	Verified	*Ensures that each repayment component remains within valid bounds.*	*Report*
repay_zero_payment_noop	Verified	*Validates that a zero repayment amount produces no state changes to either fees or principal and issues no refund, confirming idempotent behavior for null payments.*	*Report*
repay_timestamp_clear_when_fully_paid	Verified	*Verifies that once a borrow position is fully repaid (principal becomes zero), its start timestamp is cleared.*	*Report*

8. Split vs combined repay

This rule checks that paying in two chunks (say, x then y) can never reduce principal more than paying the same total in one lump (x+y). It also checks that fee reduction is identical in both cases.

P-08. **Split vs combined repay**		
Status: Verified	We check that fee accounting is split-invariant and repaying principal in parts can never reduce it by more than when paying together.	

Rule Name	Status	Description	Link to rule report
repay_split_payments_combine	Verified	*When calling `reduce_borrow_asset_liability`, calling it twice reduces the fee correctly and does not allow extra reduction in the principal.*	*Report*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.