

# Security Audit Report: Migrate.cairo Smart Contract

**Contract:** Migrate.cairo

**Location:** /Users/johannes/Documents/dev/vesu/vesu-v2-periphery/src/migrate.cairo

**Audit Date:** 2025-11-13

**Auditor:** Smart Contract Security Auditor

**Blockchain:** StarkNet (Cairo)

## Executive Summary

The Migrate.cairo smart contract facilitates position migrations from V1 and V2 protocols to a new V2 protocol using flash loans. The audit identified **1 Critical vulnerability, 3 High-risk issues, 5 Medium-risk issues, 3 Low-risk issues, and 2 Informational findings**. The most severe issues involve missing access control for critical operations, division-by-zero vulnerabilities in LTV calculations, and potential integer underflow in unsigned arithmetic. The contract requires significant security improvements before deployment.

## Summary Statistics

Severity	Count
Critical	1
High	3
Medium	5
Low	3
Informational	2
**Total**	**14**

## Current Risk Assessment

- Risk Level:** HIGH
- Deployment Status:** NOT RECOMMENDED in current state
- Recommended Actions:** Address all Critical and High severity issues before deployment

---

# Table of Contents

1. [Critical Findings](#critical-findings)
  2. [High-Risk Issues](#high-risk-issues)
  3. [Medium-Risk Issues](#medium-risk-issues)
  4. [Low-Risk Issues](#low-risk-issues)
  5. [Informational Findings](#informational-findings)
  6. [Gas Optimizations](#gas-optimizations)
  7. [Positive Observations](#positive-observations)
  8. [Recommendations Summary](#recommendations-summary)
  9. [Overall Security Assessment](#overall-security-assessment)
- 

## Critical Findings

### 1. Missing Access Control on Migration Functions

**Severity:** CRITICAL

**Location:** Lines 401-420, 422-441

**Functions:** `migrate_position_from_v1`, `migrate_position_from_v2`

#### Description

The `migrate_position_from_v1` and `migrate_position_from_v2` functions have no access control checks. Anyone can call these functions to migrate another user's position, provided the user has delegated to the Migrate contract. This allows malicious actors to:

- Force migrations at unfavorable times (e.g., when prices are moving against the position)
- Front-run legitimate migrations
- Grief users by triggering migrations when LTV conditions are barely met

## Vulnerable Code

```
// Line 401-420
fn migrate_position_from_v1(ref self: ContractState, params: MigratePositionFromV1Params) {
    // No check that caller is from_user or authorized
    let MigratePositionFromV1Params {
        from_pool_id, to_pool, collateral_asset, debt_asset, from_user, debt_to_migrate, ...
    } = params.clone();
    // ... rest of function
}
```

## Impact

- Users lose control over when their positions are migrated
- Potential for MEV (Miner Extractable Value) attacks and front-running
- Forced migrations during unfavorable market conditions could result in liquidations
- Loss of user funds due to timing manipulation

## Recommendation

Add a caller authorization check:

```
fn migrate_position_from_v1(ref self: ContractState, params: MigratePositionFromV1Params) {
    let caller = get_caller_address();
    assert!(caller == params.from_user, "unauthorized-caller");
    // ... rest of function
}
```

Alternatively, implement a more flexible authorization system:

```
// Option 2: Allow authorized operators
#[storage]
struct Storage {
    // ... existing storage
    authorized_operators: LegacyMap<(ContractAddress, ContractAddress), bool>,
}

fn migrate_position_from_v1(ref self: ContractState, params: MigratePositionFromV1Params) {
    let caller = get_caller_address();
    let is_authorized = caller == params.from_user ||
        self.authorized_operators.read((params.from_user, caller));
    assert!(is_authorized, "unauthorized-caller");
    // ... rest of function
}
```

---

## High-Risk Issues

## 1. Division by Zero in LTV Calculation

**Severity:** HIGH

**Location:** Lines 162, 244, 361

### Description

The LTV (Loan-to-Value) calculation `debt_value * SCALE / collateral_value` will panic if `collateral_value` is zero. This can occur if:

- The position has no collateral
- Oracle returns zero price for collateral asset
- There's a rounding issue resulting in zero collateral value

### Vulnerable Code

```
// Line 162
let from_ltv = debt_value * SCALE / collateral_value;

// Line 244
let from_ltv = debt_value * SCALE / collateral_value;

// Line 361
let to_ltv = debt_value * SCALE / collateral_value;
```

### Impact

- Contract becomes unusable for positions with zero collateral value
- DOS (Denial of Service) attack vector if attacker can manipulate oracle prices
- Migration fails unexpectedly for edge cases
- Users unable to migrate positions even with proper setup

### Recommendation

Add zero-check before division:

```
assert!(collateral_value > 0, "zero-collateral-value");
let from_ltv = debt_value * SCALE / collateral_value;
```

For more robust handling:

```

// Handle zero collateral gracefully
let from_ltv = if collateral_value > 0 {
    debt_value * SCALE / collateral_value
} else {
    if debt_value > 0 {
        panic!("invalid-position-zero-collateral-with-debt");
    } else {
        0 // Both zero, valid empty position
    }
};

```

## 2. Integer Underflow in LTV Range Check

**Severity:** HIGH

**Location:** Line 362

### Description

The LTV range check `from_ltv - max_ltv_delta <= to_ltv` will underflow if `max_ltv_delta > from_ltv`, since Cairo's `u256` subtraction on unsigned integers can underflow. This could allow migrations that significantly increase LTV beyond safe limits.

### Vulnerable Code

```

// Line 362
assert!(from_ltv - max_ltv_delta <= to_ltv && to_ltv <= from_ltv + max_ltv_delta, "ltv-out-of-range");

```

### Impact

- If `max_ltv_delta > from_ltv`, underflow occurs and check may pass incorrectly
- Positions could be migrated with dangerously high LTV
- Users could lose funds due to immediate liquidation after migration
- Protocol could accumulate bad debt

### Recommendation

Use checked subtraction or reorder the comparison:

```

// Option 1: Check bounds separately
assert!(to_ltv <= from_ltv + max_ltv_delta, "ltv-too-high");
if from_ltv >= max_ltv_delta {
    assert!(to_ltv >= from_ltv - max_ltv_delta, "ltv-too-low");
} else {
    // If max_ltv_delta > from_ltv, any to_ltv >= 0 is acceptable for lower bound
    assert!(to_ltv >= 0, "negative-ltv");
}

```

```
// Option 2: Use signed integers for the comparison
let ltv_diff = I257Trait::new(to_ltv, false) - I257Trait::new(from_ltv, false);
assert!(
    ltv_diff.abs() <= max_ltv_delta.into(),
    "ltv-out-of-range"
);
```

### 3. Unchecked i257 Absolute Value Conversion

**Severity:** HIGH

**Location:** Lines 210, 218, 219, 285, 293, 294

#### Description

The code uses `.abs()` on `i257` values and directly uses them as `u256` without checking if the conversion is safe. For negative deltas, `.abs()` returns the absolute value, but the context matters. The code assumes deltas are always in the expected direction (negative for withdrawals).

#### Vulnerable Code

```
// Line 210
assert!(debt_delta.abs() == amount, "debt-amount-mismatch");

// Lines 218-219
collateral_delta.abs(),
debt_delta.abs(),
```

#### Impact

- If deltas have unexpected signs, the `.abs()` could mask errors
- Amounts could be misinterpreted, leading to incorrect position creation
- Potential for loss of funds if collateral/debt amounts are incorrect
- Silent failures that could be exploited

#### Recommendation

Add sign checks before using `.abs()`:

```
assert!(debt_delta.sign, "debt-delta-should-be-negative");
assert!(debt_delta.abs() == amount, "debt-amount-mismatch");

assert!(collateral_delta.sign, "collateral-delta-should-be-negative");
let collateral_amount = collateral_delta.abs();
let debt_amount = debt_delta.abs();
```

---

# Medium-Risk Issues

## 1. Insufficient Flash Loan Callback Validation

**Severity:** MEDIUM

**Location:** Lines 385-386

### Description

The flash loan callback only validates that the caller is the stored pool address and the sender is the contract itself. However, it doesn't verify that the pool address was legitimately set during an active migration. A malicious pool contract could potentially be registered temporarily if there's any way to manipulate the storage.

### Vulnerable Code

```
// Lines 385-386
assert!(get_caller_address() == self.pool.read(), "caller-not-pool");
assert!(sender == get_contract_address(), "unknown-sender");
```

### Impact

- If an attacker can find a way to set the pool storage variable, they could execute arbitrary migration logic
- Limited by the reentrant call guard, but still a concern

### Recommendation

Add additional validation:

```
let expected_pool = self.pool.read();
assert!(expected_pool != 0.try_into().unwrap(), "no-active-flash-loan");
assert!(get_caller_address() == expected_pool, "caller-not-pool");
assert!(sender == get_contract_address(), "unknown-sender");
```

## 2. Unsafe Token Approval Pattern

**Severity:** MEDIUM

**Location:** Lines 166-167, 247-248, 319-321, 331-333, 367-369, 395

## Description

The contract approves tokens without first setting approval to zero. Some ERC20 tokens (like USDT on Ethereum) require approval to be set to 0 before changing to a new value. While this might not be an issue on StarkNet currently, it's a best practice violation.

Additionally, approvals are set to exact amounts multiple times for the same spender. Failed approvals are checked with `assert!`, but there's no revocation of approvals after operations complete, leaving residual approvals.

## Vulnerable Code

```
// Line 166-167
assert!(
    IERC20Dispatcher { contract_address: debt_asset }.approve(singleton_v2.contract_address, amount),
    "approve-failed",
);
```

## Impact

- Residual approvals could be exploited if there's a vulnerability in the approved contract
- Incompatibility with some token standards
- Multiple approvals for the same spender increase gas costs
- Potential security risk if approved contracts are compromised

## Recommendation

Use a helper function for safe approvals:

```
#[generate_trait]
impl SafeERC20 of SafeERC20Trait {
    fn safe_approve(token: ContractAddress, spender: ContractAddress, amount: u256) {
        let token_dispatcher = IERC20Dispatcher { contract_address: token };
        // Reset approval to 0 first
        assert!(token_dispatcher.approve(spender, 0), "approve-reset-failed");
        // Set new approval
        assert!(token_dispatcher.approve(spender, amount), "approve-failed");
    }
}
```

## 3. Missing Zero Address Checks

**Severity:** MEDIUM

**Location:** Lines 127-130, throughout params

## Description

The constructor and migration functions don't validate that critical addresses are non-zero. This could lead to locked contracts or failed migrations if addresses are accidentally set to zero.

## Vulnerable Code

```
// Lines 127-130
fn constructor(ref self: ContractState, singleton_v2: ISingletonV2Dispatcher, migrator: ITokenMigrationDispatcher) {
    self.singleton_v2.write(singleton_v2);
    self.migrator.write(migrator);
}
```

## Impact

- Contract could be deployed with invalid addresses, requiring redeployment
- Failed migrations if params contain zero addresses
- Funds could be lost if sent to zero address
- Increased gas costs from failed transactions

## Recommendation

Add comprehensive zero address checks:

```
fn constructor(ref self: ContractState, singleton_v2: ISingletonV2Dispatcher, migrator: ITokenMigrationDispatcher) {
    assert!(singleton_v2.contract_address != 0.try_into().unwrap(), "invalid-singleton-v2");
    assert!(migrator.contract_address != 0.try_into().unwrap(), "invalid-migrator");
    self.singleton_v2.write(singleton_v2);
    self.migrator.write(migrator);
}

// Add similar checks in migration functions for all address params
fn validate_addresses(params: @MigratePositionFromV1Params) {
    assert!(*params.to_pool != 0.try_into().unwrap(), "invalid-to-pool");
    assert!(*params.collateral_asset != 0.try_into().unwrap(), "invalid-collateral-asset");
    assert!(*params.debt_asset != 0.try_into().unwrap(), "invalid-debt-asset");
    assert!(*params.from_user != 0.try_into().unwrap(), "invalid-from-user");
    assert!(*params.to_user != 0.try_into().unwrap(), "invalid-to-user");
}
```

## 4. Inconsistent Collateral/Debt Amount Handling

**Severity:** MEDIUM

**Location:** Lines 180-205, 260-281

## Description

The logic for handling `collateral_to_migrate` and `debt_to_migrate` parameters is inconsistent. When these values are 0 or exceed the actual position, the code targets the full position (native shares/debt). However, the condition `collateral_to_migrate > collateral` could allow partial values that are very close to the total, potentially leading to unexpected rounding behavior.

## Vulnerable Code

```
// Lines 180-192
collateral: if (collateral_to_migrate == 0 || collateral_to_migrate > collateral) {
    AmountSingletonV2 {
        amount_type: AmountType::Target,
        denomination: AmountDenomination::Native,
        value: I257Trait::new(0, false),
    }
} else {
    AmountSingletonV2 {
        amount_type: AmountType::Delta,
        denomination: AmountDenomination::Assets,
        value: I257Trait::new(collateral_to_migrate, true),
    }
},
```

## Impact

- Unclear semantics for users (when does full migration occur?)
- Potential for partial migrations to leave dust amounts
- Rounding errors could accumulate
- Unexpected behavior in edge cases

## Recommendation

Add explicit validation and clearer logic:

```
let use_full_collateral = collateral_to_migrate == 0 || collateral_to_migrate >= collateral;
if !use_full_collateral {
    assert!(collateral_to_migrate <= collateral, "collateral-exceeds-position");
}

collateral: if use_full_collateral {
    AmountSingletonV2 {
        amount_type: AmountType::Target,
        denomination: AmountDenomination::Native,
        value: I257Trait::new(0, false),
    }
} else {
    AmountSingletonV2 {
        amount_type: AmountType::Delta,
        denomination: AmountDenomination::Assets,
        value: I257Trait::new(collateral_to_migrate, true),
    }
},
```

```
    },
```

## 5. Assumed 1:1 Token Swap Rate Not Validated

**Severity:** MEDIUM

**Location:** Lines 323-324, 370-371

### Description

The code assumes token migrations (legacy USDC to new USDC) maintain a 1:1 ratio, but this is not validated. The contract doesn't check the balance before and after swap to ensure it received the expected amount.

### Vulnerable Code

```
// Lines 323-324
// assume swap amounts are 1:1
migrator.swap_to_new(collateral_delta);

// Lines 370-371
// assume swap amounts are 1:1
migrator.swap_to_legacy(debt_delta);
```

### Impact

- If swap rate deviates from 1:1, position will have incorrect collateral/debt amounts
- User could receive less collateral than expected
- Debt repayment could be insufficient, leaving contract in debt to the pool
- Potential for sandwich attacks on token swaps

### Recommendation

Validate swap outputs by checking balances:

```
// For collateral swap
let balance_before = IERC20Dispatcher { contract_address: new_token }.balance_of(get_contract_address());
migrator.swap_to_new(collateral_delta);
let balance_after = IERC20Dispatcher { contract_address: new_token }.balance_of(get_contract_address());
let received = balance_after - balance_before;
assert!(received >= collateral_delta, "insufficient-swap-output");

// Similar for debt swap with appropriate tolerance
let balance_before = IERC20Dispatcher { contract_address: legacy_token }.balance_of(get_contract_address());
migrator.swap_to_legacy(debt_delta);
let balance_after = IERC20Dispatcher { contract_address: legacy_token }.balance_of(get_contract_address());
let received = balance_after - balance_before;
```

```
assert!(received >= debt_delta, "insufficient-swap-output");
```

---

## Low-Risk Issues

### 1. Storage Variables Never Read

**Severity:** LOW

**Location:** Lines 121-123

#### Description

The storage variables `usdc_e`, `usdc`, and `pool` are declared but never used directly (pool is only used for the reentrancy guard). These appear to be legacy code or placeholder variables.

#### Vulnerable Code

```
// Lines 121-123
usdc_e: ContractAddress,
usdc: ContractAddress,
pool: ContractAddress,
```

#### Impact

- Wasted storage slots (gas inefficiency)
- Code confusion and maintenance burden
- Potential for future bugs if these are used incorrectly

#### Recommendation

Remove unused storage variables or document why they exist:

```
#[storage]
struct Storage {
    singleton_v2: ISingletonV2Dispatcher,
    // Used for reentrancy guard
    pool: ContractAddress,
    migrator: ITokenMigrationDispatcher,
}
```

## 2. Missing Event Emissions

**Severity:** LOW

**Location:** Throughout contract

### Description

The contract has no event emissions for critical operations like successful migrations, flash loan execution, or token swaps. This makes it difficult to track contract activity and debug issues.

### Impact

- Difficult to monitor contract usage off-chain
- No audit trail for migrations
- Harder to debug failed transactions
- Poor user experience for tracking migration status

### Recommendation

Add comprehensive event emissions:

```
#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    PositionMigrated: PositionMigrated,
    FlashLoanExecuted: FlashLoanExecuted,
    TokenSwapped: TokenSwapped,
}

#[derive(Drop, starknet::Event)]
struct PositionMigrated {
    from_user: ContractAddress,
    to_user: ContractAddress,
    collateral_asset: ContractAddress,
    debt_asset: ContractAddress,
    collateral_amount: u256,
    debt_amount: u256,
    from_ltv: u256,
    to_ltv: u256,
}

#[derive(Drop, starknet::Event)]
struct FlashLoanExecuted {
    pool: ContractAddress,
    asset: ContractAddress,
    amount: u256,
}

#[derive(Drop, starknet::Event)]
struct TokenSwapped {
    from_token: ContractAddress,
    to_token: ContractAddress,
```

```
        amount: u256,  
    }
```

### 3. Inconsistent Error Messages

**Severity:** LOW

**Location:** Throughout contract

#### Description

Error messages use different naming conventions (kebab-case like "debt-amount-mismatch" appears consistently, but could be improved with more descriptive messages).

#### Impact

- Minor code quality issue
- Slightly harder to maintain and debug
- User experience could be improved

#### Recommendation

Standardize and enhance error messages:

```
// Current  
"debt-amount-mismatch"  
"approve-failed"  
"ltv-out-of-range"  
  
// Improved  
"debt-amount-mismatch-expected-{expected}-got-{actual}"  
"token-approval-failed-for-{token}-to-{spender}"  
"ltv-validation-failed-outside-range"
```

---

## Informational Findings

### 1. Reentrancy Guard Implementation

**Severity:** INFORMATIONAL

**Location:** Lines 137-141

## Description

The reentrancy guard uses a simple storage flag pattern which is effective. However, the pattern could be made more robust with a dedicated modifier or internal function.

## Observation

```
// Lines 137-141
assert!(self.pool.read() == 0.try_into().unwrap(), "reentrant-call");
self.pool.write(pool.contract_address);
pool.flash_loan(get_contract_address(), asset, amount, false, data);
self.pool.write(0.try_into().unwrap());
```

**Positive:** The guard is correctly implemented and prevents reentrancy attacks.

## Recommendation (Enhancement)

Consider creating a dedicated reentrancy guard trait for reusability:

```
#[generate_trait]
impl ReentrancyGuard of ReentrancyGuardTrait {
    fn ensure_not_reentrant(ref self: ContractState) {
        assert!(self.pool.read() == 0.try_into().unwrap(), "reentrant-call");
    }

    fn set_executing(ref self: ContractState, pool: ContractAddress) {
        self.pool.write(pool);
    }

    fn clear_executing(ref self: ContractState) {
        self.pool.write(0.try_into().unwrap());
    }
}
```

## 2. Delegation Assumptions

**Severity:** INFORMATIONAL

**Location:** Throughout migration functions

## Description

The contract assumes users have properly delegated to the Migrate contract before calling migration functions. If delegation is not set, the transaction will fail within the pool contract calls, but the error might not be immediately clear to users.

## Recommendation

Consider adding pre-flight checks or clearer documentation:

```
// Add to migration functions or create a view function
fn check_delegation_status(
    self: @ContractState,
    pool: IPoolDispatcher,
    user: ContractAddress
) -> bool {
    // Check if user has delegated to this contract
    // Implementation depends on pool interface
}
```

---

# Gas Optimizations

## 1. Redundant Storage Reads

**Location:** Lines 144, 311, 406

Multiple reads of the same storage variable (`singleton_v2` and `migrator`) could be optimized by reading once and reusing the value.

**Current:**

```
let singleton_v2 = self.singleton_v2.read(); // Read in function A
// Later in same transaction...
let singleton_v2 = self.singleton_v2.read(); // Read again in function B
```

**Optimized:**

```
// Store in local variable and pass as parameter
let singleton_v2 = self.singleton_v2.read();
// Use throughout function
```

## 2. Clone Operation Overhead

**Location:** Lines 404, 425

The `params.clone()` operation creates a full copy of the parameters. Consider destructuring directly if possible.

**Current:**

```
let params_clone = params.clone();
let MigratePositionFromV1Params { from_pool_id, ... } = params_clone;
```

## Optimized:

```
// Destructure directly without clone if params isn't needed later
let MigratePositionFromV1Params { from_pool_id, ... } = params;
```

---

## Positive Observations

The audit identified several well-implemented security features:

1. **Reentrancy Protection:** The contract implements effective reentrancy protection for flash loan operations (lines 137-141), preventing recursive calls that could drain funds.
  2. **Comprehensive Test Coverage:** Test files demonstrate thorough testing including:
    - Partial and full migrations
    - Legacy token to new token conversions
    - LTV validation scenarios
    - Reentrancy attack prevention
  3. **LTV Validation:** The contract validates that the destination LTV remains within acceptable bounds (line 362), protecting users from dangerous migrations that could lead to immediate liquidation.
  4. **Flexible Migration Parameters:** Support for both partial and full position migrations provides good user flexibility and use cases.
  5. **Flash Loan Pattern:** Proper use of flash loans to enable atomic migrations without requiring users to hold capital.
  6. **Token Migration Support:** Built-in support for migrating between legacy and new token versions.
- 

## Recommendations Summary

### Immediate Priority (MUST FIX before deployment)

1. Add access control to `migrate_position_from_v1` and `migrate_position_from_v2` functions

2. **Fix division-by-zero** vulnerabilities in LTV calculations (lines 162, 244, 361)
3. **Fix integer underflow** in LTV range check (line 362)
4. **Validate swap outputs** for token migrations (lines 323-324, 370-371)
5. **Add zero address checks** in constructor and migration functions

## High Priority (SHOULD FIX before deployment)

6. Add sign validation for `i257` values before using `.abs()`
7. Implement safe token approval pattern with zero-reset
8. Improve flash loan callback validation
9. Clarify and standardize collateral/debt amount handling logic

## Medium Priority (RECOMMENDED for production quality)

10. Add comprehensive event emissions for all critical operations
11. Remove unused storage variables (`usdc_e`, `usdc`)
12. Standardize and improve error messages
13. Add delegation status checks or pre-flight validation
14. Implement gas optimizations (reduce storage reads, avoid unnecessary clones)

## Nice to Have

15. Add pause functionality for emergency situations
  16. Implement upgradability pattern for future improvements
  17. Add detailed NatSpec documentation
  18. Create migration limits per user/per transaction
  19. Add whitelisting for supported pools and tokens
- 

## Overall Security Assessment

Current Risk Level: HIGH

The contract has several critical and high-severity vulnerabilities that must be addressed before deployment. The most serious issues are:

- **Lack of access control** allowing unauthorized migrations
- **Division-by-zero and integer underflow** vulnerabilities in critical calculations
- **Unvalidated token swap assumptions** that could lead to fund loss

## Deployment Recommendation: NOT RECOMMENDED

The contract **should not be deployed** to mainnet in its current state.

## Estimated Remediation Timeline

- **Critical Issues:** 2-3 days of development
- **High Priority Issues:** 3-5 days of development
- **Testing & Verification:** 1-2 weeks
- **External Audit:** 2-4 weeks
- **Total Estimated Time:** 4-7 weeks

## Post-Fix Recommendations

After addressing all identified issues:

1. **External Security Audit:** Engage a specialized blockchain security firm for independent verification
2. **Formal Verification:** Consider formal verification of LTV calculation logic and critical invariants
3. **Extensive Testing:** Conduct mainnet-fork testing with various edge cases, price scenarios, and attack vectors
4. **Bug Bounty Program:** Launch a bug bounty program after deployment to incentivize community security research
5. **Monitoring & Alerts:** Implement real-time monitoring for suspicious activity
6. **Time-locked Upgrades:** Consider implementing time-locked upgrades or emergency pause functionality
7. **Gradual Rollout:** Consider limiting migration amounts initially to reduce risk exposure

## Conclusion

The Migrate.cairo contract demonstrates good understanding of flash loan patterns and includes effective reentrancy protection. However, the critical access control vulnerability and several high-severity arithmetic issues present significant security risks. With proper remediation of the identified issues, comprehensive testing, and external audit, this contract can achieve production-ready security standards.

---

## Appendix: Vulnerability Classification

### Severity Definitions

**CRITICAL:** Issues that can lead to loss of funds, unauthorized access, or complete contract compromise. Must be fixed immediately.

**HIGH:** Issues that could lead to significant loss of funds or compromise of core functionality under specific conditions. Should be fixed before deployment.

**MEDIUM:** Issues that could lead to unexpected behavior, minor loss of funds, or degraded user experience. Should be addressed for production quality.

**LOW:** Issues that represent best practice violations or minor improvements. Good to fix but not blocking.

**INFORMATIONAL:** Observations and suggestions for improvement without direct security impact.

---

**Report Generated:** 2025-11-13

**Auditor:** Smart Contract Security Auditor

**Review Status:** Complete

**Recommendation:** Major revisions required before deployment