Iron Badger Brotherhood Audit

Prepared by: Nemi

Table of Contents

Introduction

The purpose of this report is to document the findings from the security audit of `Iron Badger Brotherhood`. This audit was conducted to identify potential vulnerabilities and provide actionable recommendations to improve the contract's security and adherence to best practices. The findings are categorized with detailed descriptions, impacts, proof of concepts, recommended mitigations, and tools used. Individual bugs have been documented separately in the findings section.

Scope of Audit

The audit focused on the following aspects:

- Security vulnerabilities
- Code correctness and logic
- Adherence to best practices
- Gas efficiency

Methodology

The audit process involved:

- Manual code review
- Automated analysis using Slither and Aderyn
- Scenario-based testing using Foundry

Disclaimer

This report is based on the information provided at the time of the audit and does not guarantee the absence of future vulnerabilities. Subsequent security review and on-chain monitoring are strongly recommended.

Architecture Overview

The Iron Badger Brotherhood ecosystem is a modular DeFi protocol leveraging smart contracts to enable peer-to-peer lending through ERC-1155s (Pacts). The architecture is designed for transparency, flexibility, and scalability, comprising of these key components:

- **Iron Pact**: Core protocol for creating and managing collateralized Pacts, with automated liquidations triggered by default, supported by `Pact.sol` and `PactStorage.sol`.
- **Iron Forge**: Launch interface for borrowers to propose Pacts, managed by `PactLaunch.sol` and `PactLaunchStorage.sol`.
- **Iron Rise/Fall Auctions**: Decentralized secondary markets for trading Pacts via English Auctions (`UpwardAuction.sol`) and Dutch Auctions (`DownwardAuctions.sol`), enhancing liquidity.
- **Utility Contracts**: `TimeManagement.sol` and `helperFunction.sol` handle time-based operations and reusable logic.

Data Flow

Borrowers create Pacts via Iron Forge, minting semi-NFTs stored in `PactStorage.sol`. Creditors purchase Pacts, which can be traded on auction markets. Liquidations are automated, with collateral managed on-chain.

Auditor's View

The architecture is **secure** and **novel**, offering a robust framework for DeFi lending with innovative use of ERC-1155s and default-based liquidations.

Manual Findings Summary

The security review was carried out from July 18th, 2025 to August 5th, 2025

Review Commit Hash: `24ec7b029a42d30608e15a2d68b351c06b53f1f7`

Contract Scope: The following smart contracts were in scope of the audit:

- `Pact.sol`
- `PactStorage.sol`
- `PactLaunch.sol`
- `PactLaunchStorage.sol`
- `UpwardAuction.sol`
- `UpwardAuctionStorage.sol`
- `DownwardAuctions.sol`
- `DownwardAuctionStorage.sol`
- `TimeManagement.sol`
- `helperFunction.sol`

The following number of issues were found manually, categorized by their severity:

- **High**: 1 issue(s)
- **Medium**: 2 issue(s)
- **Low**: 3 issue(s)

Findings Summary:

ID	Title	Severity	Status
H-1	Broken Pot Validation in `_instalmentPot()`	High	Fixed
M-2	`createNewPact` doesn't account for fee-on-transfer tokens causing discrepancies in internal accounting	Medium	Fixed
L-1	Redundant `nonReentrant` Modifier in Functions	Low	Fixed
L-2	Unnecessary uint16 Typecasting in Penalty Arrays and `_setScoreForUser` Function	Low	Fixed
L-3	Inefficient Gas Usage Due to Repeated Array Length Access in Loops	Low	Fixed

[H-1] Broken Pot Validation in `_instalmentPot()`

Description:

The `_instalmentPot()` function contains a critical logical error in its pot validation. The function compares `auctions[_index].pot` with itself (`auctions[_index].pot > auctions[_index].pot`) which will always evaluate to false, effectively disabling the intended pot size validation.

Impact:

- **Critical Severity**: This vulnerability completely bypasses the intended pot increment limitation (MAX_POT_MULTIPLIER = 150%)
- Undermines the auction's fairness and economic model
- Could lead to auction manipulation and potential fund loss

Proof of Concept:

1. The vulnerable code segment:

```
//Always false, will not execute
if (auctions[_index].pot > auctions[_index].pot) {
    require(
        _amount <= ((auctions[_index].pot * MAX_POT_MULTIPLIER) / 100),
        "Pot exceeds maximum allowed increment"
    );
}
```

2. Attack Scenario:

- First bidder places bid of 100 (valid)
- Second bidder can bid 10000 (should be limited to 150 but check is bypassed)
- This disrupts the intended auction mechanics

Recommended Mitigation:

Replace the incorrect comparison with:

```
- if (auctions[_index].pot > auctions[_index].pot) {
+ if (auctions[_index].pot > 0) {
    require(
        _amount <= ((auctions[_index].pot * MAX_POT_MULTIPLIER) / 100),
        "Pot exceeds maximum allowed increment"
    );
}
```

Tools Used:

- Manual code review

[M-1] `amountInSell` Accounting Vulnerability Through Direct Token Transfers

---

Description:

The `_launchNewPact` function in PactLaunch.sol uses `IERC1155(pactContract).balanceOf(address(this), _id)` to determine how many pacts are available for sale. This creates a vulnerability where the contract's accounting can become incorrect if anyone (including the debtor themselves) accidentally or intentionally sends ERC1155 tokens directly to the contract outside of the intended launch mechanism.

Vulnerable Code:

```
amountInSell[_id] = IERC1155(pactContract).balanceOf(
    address(this),
    _id
); // Update the amount of pacts available for sale.
```

Impact:

HIGH SEVERITY - This vulnerability can lead to:

1. **Incorrect Accounting**: The contract's internal accounting becomes unreliable when tokens are sent directly
2. **Unintended Sale Quantities**: More pacts will be marked as "for sale" bypassing contract logic

3. **Business Logic Failure**: The contract may allow sales of pacts that weren't properly launched through the intended mechanism
4. **Potential Economic Issues**: Discrepancies between expected and actual available quantities

Proof of Concept:

Below is a coded proof of concept demonstrating the vulnerability:

1. **Setup**: Debtor owns 10 pacts of ID #1 and wants to launch them for sale
2. **Unintended Transfer**: The debtorsends additional pacts of the same ID directly to the contract using `safeTransferFrom`
3. **Vulnerability Triggered**: When debtor calls `launchNewPact(1, 5)`, the contract incorrectly sets `amountInSell[1] = 10` (5 intended + 5 accidentally sent)
4. **Impact**: The system now thinks there are 10 pacts for sale instead of the intended 5

```
        function testVulnerabilityAmountInSellManipulation() public {
            console2.log("=== Testing amountInSell Accounting Vulnerability ===");

            // Step 1: Setup - Debtor owns 10 units of ID #1 (already set in setUp
with pactAmount = 10)
            console2.log("Initial state:");
            console2.log(
                "- Debtor (user1) pacts:",
                pactToken.balanceOf(user1, pactId)
            );
            console2.log(
                "- Contract pacts:",
                pactToken.balanceOf(address(pactLaunch), pactId)
            );
            console2.log(
                "- amountInSell:",
                pactLaunch.showAmountInSellForPact(pactId)
            );

            // Step 2: Unintended Transfer - Debtor accidentally sends 5 units to
the contract
            console2.log("\n=== Debtor accidentally sends 5 to contract ===");
            uint256 unintendedPactAmount = 5;
            vm.prank(user1);
            pactToken.safeTransferFrom(
                user1,
                address(pactLaunch),
                pactId,
                unintendedPactAmount,
                ""
            );

            console2.log("After unintended transfer:");
            console2.log(
                "- Debtor (user1) pacts:",
                pactToken.balanceOf(user1, pactId)
```

```
            );
            console2.log(
                "- Contract pacts:",
                pactToken.balanceOf(address(pactLaunch), pactId)
            );

            // Step 3: Debtor launches remaining 5
            console2.log("\n=== Debtor launches 5 ===");
            uint256 launchAmount = 5; // Launch the remaining 5 pacts
            vm.prank(user1);
            pactLaunch.launchNewPact(pactId, launchAmount);

            // Show the vulnerability: amountInSell is set to 10
            uint256 manipulatedAmount =
pactLaunch.showAmountInSellForPact(pactId);
            uint256 contractBalance = pactToken.balanceOf(
                address(pactLaunch),
                pactId
            );

            console2.log("After legitimate launch:");
            console2.log(
                "- Debtor (user1) pacts:",
                pactToken.balanceOf(user1, pactId)
            );
            console2.log("- Contract pacts:", contractBalance);
            console2.log("- amountInSell (INCORRECT):", manipulatedAmount);
            console2.log("- Expected amountInSell:", launchAmount);
            console2.log("- Inflation amount:", manipulatedAmount - launchAmount);

            // Step 4: Verify the vulnerability (amountInSell = 10)
            assertEq(
                contractBalance,
                launchAmount + unintendedPactAmount,
                "Contract should have 10 pacts (5 legitimate + 5 unintended)"
            );
            assertEq(
                manipulatedAmount,
                launchAmount + unintendedPactAmount,
                "amountInSell should be 10 (5 + 5)"
            );
            assertEq(manipulatedAmount, 10, "amountInSell should be exactly 10");
            assertTrue(
                manipulatedAmount > launchAmount,
                "amountInSell should be greater than intended 5"
            );

            console2.log("\n=== VULNERABILITY CONFIRMED ===");
            console2.log(
                "Unintended transfer of 5 pacts caused amountInSell to be 10
instead of 5"
            );
        }
```

Test Results:

```
forge test --match-test testVulnerabilityAmountInSellManipulation -vvv --via-ir
$ forge test --match-test testVulnerabilityAmountInSellManipulation -vvv --via-ir
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/pactLaunch.t.sol:PactLaunchTest
[PASS] testVulnerabilityAmountInSellManipulation() (gas: 262362)
Logs:
  === Testing amountInSell Accounting Vulnerability ===
  Initial state:
  - Debtor (user1) pacts: 10
  - Contract pacts: 0
  - amountInSell: 0

=== Debtor accidentally sends 5 to contract ===
  After unintended transfer:
  - Debtor (user1) pacts: 5
  - Contract pacts: 5

=== Debtor launches 5 ===
  After legitimate launch:
  - Debtor (user1) pacts: 0
  - Contract pacts: 10
  - amountInSell (INCORRECT): 10
  - Expected amountInSell: 5
  - Inflation amount: 5

=== VULNERABILITY CONFIRMED ===
  Unintended transfer of 5 pacts caused amountInSell to be 10 instead of 5

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.84ms (1.45ms CPU
time)

Ran 1 test suite in 30.53ms (3.84ms CPU time): 1 tests passed, 0 failed, 0 skipped
(1 total tests)
```

```
Initial state:
- Debtor pacts: 10
- Contract pacts: 0
- amountInSell: 0

After direct transfer of 5 additional pacts:
- Contract pacts: 5

After intended launch of 10 pacts:
```

```
- amountInSell: 10 (INCORRECT - should be 5)
- System believes 10 pacts are for sale instead of 5
```

The vulnerability occurs because the contract relies on `balanceOf()` instead of tracking only the pacts that were properly launched through the `launchNewPact()` function. Any tokens sent directly to the contract (whether accidentally or intentionally) will be included in the sale amount calculation.

Recommended Mitigation:

Primary Fix: Replace external balance queries with internal accounting:

```
function _launchNewPact(address _user, uint _id, uint _amount) internal {
    // ... existing validation ...

    _depositPact(_user, address(this), _id, _amount);

    // FIX: Use internal accounting instead of balanceOf
+    amountInSell[_id] += _amount; // Increment by actual launched amount
-    amountInSell[_id] = IERC1155(pactContract).balanceOf(address(this), _id);
```

Tools used:

- Foundry for testing and proof of concept
- Manual code review

[M-2] `createNewPact` doesn't account for fee-on-transfer tokens causing discrepancies in internal accounting

---

Description:

The protocol is designed to support all ERC20 tokens as collateral for creating pacts. However, the current implementation does not account for fee-on-transfer tokens, which deduct a fee during transfers. This leads to a discrepancy between the collateral amount the contract expects to receive `_collateral` and the actual amount received after the transfer fee is deducted. The contract calculates an issuance fee based on the specified `_collateral` amount and records the collateral as `_collateral - fee`, assuming the full `_collateral` amount was received. For fee-on-transfer tokens, this assumption fails, as the contract receives less than `_collateral`, resulting in a mismatch between the recorded collateral and the actual balance held.

Impact:

- **Under-collateralization**: If the transfer fee exceeds the issuance fee, the contract holds less collateral than recorded, risking insufficient funds for pact settlement or collateral return.
- **Over-collateralization**: If the transfer fee is less than the issuance fee, the contract holds more collateral than recorded, causing accounting inconsistencies.

- **Operational Failures**: Functions that depend on the recorded collateral amount may fail or behave unpredictably due to the discrepancy, undermining the contract's reliability.

Recommended Mitigation:

To resolve this issue, the contract should measure the actual amount of collateral received after the transfer and adjust the recorded collateral and fee calculations accordingly.

```
function createPact(
    address _tokenCollateral,
    uint256 _collateral,
    address _tokenLoan,
    uint256 _sizeLoan,
    uint256 _interest,
    uint256 _rewardMaturity,
    uint256 _expiredPact,
    uint256 _amount,
    string memory _describes
) external {

-    uint fee = _emisionPactFee(msg.sender, _tokenCollateral, _collateral);

+    // Measure balance before and after transfer to determine actual received
amount
+    uint256 balanceBefore = IERC20(_tokenCollateral).balanceOf(address(this));
     _depositCollateralToken(msg.sender, _tokenCollateral, _collateral);
+    uint256 balanceAfter = IERC20(_tokenCollateral).balanceOf(address(this));
+    uint256 received = balanceAfter - balanceBefore;

+    uint fee = _emisionPactFee(msg.sender, _tokenCollateral, received);

     // Store and initialize the new pact with the adjusted collateral amount
     _createNewPact(
         currentId,
         msg.sender,
         _tokenLoan,
         _sizeLoan,
         _interest,
         _rewardMaturity,
         _expiredPact,
         _tokenCollateral,
-        _collateral - fee,
+        received - fee,
         0, // balancLoanRepay starts at 0
         _amount,
         _describes
     );
}
```

Tools Used:

- Manual Review

[L-1] Redundant `nonReentrant` Modifier in Functions

Description:

The nonReentrant modifier is unnecessarily applied to several administrative functions in the UpwardAuction contract. These functions do not make external calls or transfer funds, making them immune to reentrancy attacks. The redundant use of the nonReentrant modifier increases gas costs without providing additional security.

The affected functions are:

- `setNewPactAddress(address _pactContrac)`: Updates the pactContract address with no external calls.
- `setFeeSystem(uint _fixedFee, uint _priceThreshold, uint _dinamicFee)`: Updates the fee system struct with no external calls.
- `setNewMoneyToken(address _money)`: Updates the money token address with no external calls.
- `setFeeSeller(uint[] memory _echelons, uint[] memory _fees)`: Updates the seller fee structure with no external calls.

Impact:

The redundant nonReentrant modifiers increase gas costs for these administrative functions due to the storage read/write operations performed by the ReentrancyGuard.

Recommended Mitigation:

Remove the nonReentrant modifier from the following functions to reduce gas costs:

- setNewPactAddress
- setFeeSystem
- setNewMoneyToken
- setFeeSeller

1. `setNewPactAddress`

```
function setNewPactAddress(
address _pactContrac

- ) external nonReentrant onlyRole(OWNER_ROLE) {

+ ) external onlyRole(OWNER_ROLE) {
  require(_pactContrac != address(0), "Invalid contract address");
  require(_pactContrac != pactContract, "Address already set to this value");
  address previousAddress = pactContract;
  pactContract = _pactContrac;
  emit PactAddressUpdated(previousAddress, _pactContrac);
  }
```

2. `setFeeSystem`

```
function setFeeSystem(
uint _fixedFee,
uint _priceThreshold,
uint _dinamicFee

- ) external onlyRole(OWNER_ROLE) nonReentrant {

+ ) external onlyRole(OWNER_ROLE) {
  require(_dinamicFee <= 10000, "Dynamic fee cannot exceed 100%");
  feeSystem.fixedFee = _fixedFee;
  feeSystem.priceThreshold = _priceThreshold;
  feeSystem.dinamicFee = _dinamicFee;
  emit NewFeeSystem(_fixedFee, _priceThreshold, _dinamicFee);
  }
```
3. `setNewMoneyToken`

```diff
- function setNewMoneyToken(
-     address _money
- ) external onlyRole(ACCOUNTANT_ROLE) nonReentrant {

+ ) external onlyRole(ACCOUNTANT_ROLE) {
  require(_money != address(0), "Invalid token address");
  address oldMoney = money;
  money = _money;
  emit NewMoneyTokenAddress(oldMoney, _money);
  }
```
4. `setFeeSeller`
```diff
function setFeeSeller(
uint[] memory _echelons,
uint[] memory _fees

- ) external virtual onlyRole(OWNER_ROLE) nonReentrant {

+ ) external virtual onlyRole(OWNER_ROLE) {
  require(_echelons.length == _fees.length, "Echelons and fees length mismatch");
  for (uint i = 1; i < _echelons.length; i++) {
  require(_echelons[i] > _echelons[i - 1], "Echelons must be in ascending order");
  }
  feeSeller.echelons = _echelons;
  feeSeller.fees = _fees;
  }
```

Tools Used:

- Manual Review

[L-2] Unnecessary uint16 Typecasting in Penalty Arrays and `_setScoreForUser` Function

Description:

The `PactContract.sol` and `PactStorage.sol` contracts contain unnecessary explicit uint16 typecasting in the initialization of penalty arrays and in the `_setScoreForUser` function. Specifically:

- Penalty Arrays in `PactStorage.sol`: The arrays `mediumPenalties`, `highPenalties`, `lowPenalties`, and `veryLowPenalties` are initialized with explicit uint16 typecasts for their elements, as seen in:

```
uint16[3] internal mediumPenalties = [uint16(100), uint16(200), uint16(400)];
uint16[3] internal highPenalties = [uint16(50), uint16(100), uint16(200)];
uint16[3] internal lowPenalties = [uint16(200), uint16(400), uint16(600)];
uint16[3] internal veryLowPenalties = [uint16(280), uint16(450), uint16(720)];
```

These casts are redundant because the literals (e.g., 100, 200, 400) are interpreted as uint256 by default in Solidity and can be safely assigned to uint16 without explicit casting, provided the values are within the uint16 range (0 to 65,535). The Solidity compiler enforces this range check at compile time.

- `_setScoreForUser` Function in `PactContract.sol`: The `_setScoreForUser` function also uses unnecessary uint16 typecasting when initializing penalty arrays for different score tiers:

```
uint16[3] memory penalties = [uint16(100), uint16(200), uint16(400)];
uint16[3] memory penalties = [uint16(50), uint16(100), uint16(200)];
uint16[3] memory penalties = [uint16(200), uint16(400), uint16(600)];
uint16[3] memory penalties = [uint16(280), uint16(450), uint16(720)];
```

Recommended Mitigation:

Remove the explicit uint16 typecasting from the penalty array initializations in `PactStorage.sol` and the `_setScoreForUser` function in `PactContract.sol`. Below are the diffs for the recommended changes:

For `PactStorage.sol`:

```
-    uint16[3] internal mediumPenalties = [uint16(100), uint16(200), uint16(400)];
+    uint16[3] internal mediumPenalties = [100, 200, 400];

-    uint16[3] internal highPenalties = [uint16(50), uint16(100), uint16(200)];
+    uint16[3] internal highPenalties = [50, 100, 200];

-    uint16[3] internal lowPenalties = [uint16(200), uint16(400), uint16(600)];
+    uint16[3] internal lowPenalties = [200, 400, 600];
```

```
-    uint16[3] internal veryLowPenalties = [uint16(280), uint16(450),
uint16(720)];
+    uint16[3] internal veryLowPenalties = [280, 450, 720];
```

For `_setScoreForUser` in `PactContract.sol`:

```
     function _setScoreForUser(address _user) internal {
         uint score = conditionOfFee[_user].score; // Legge il punteggio una sola
volta

         if (score == 0 || (score >= 700000 && score <= 1000000)) {
             // Caso: nuovo utente o punteggio nella fascia "media" [700k, 1M]
-            uint16[3] memory penalties = [uint16(100), uint16(200), uint16(400)];
+            uint16[3] memory penalties = [100, 200, 400];
             conditionOfFee[_user] = ConditionOfFee(penalties, 700100);
             emit ScoreUpdated(_user, 700100);
         } else if (score > 1000000) {
             // Caso: punteggio alto (>1M)
-            uint16[3] memory penalties = [uint16(50), uint16(100), uint16(200)];
+            uint16[3] memory penalties = [50, 100, 200];
             conditionOfFee[_user].penalityForLiquidation = penalties;
             emit ScoreUpdated(_user, 100000);
         } else if (score >= 500000 && score < 700000) {
             // Caso: punteggio basso [500k, 700k)
-            uint16[3] memory penalties = [uint16(200), uint16(400), uint16(600)];
+            uint16[3] memory penalties = [200, 400, 600];
             conditionOfFee[_user].penalityForLiquidation = penalties;
             emit ScoreUpdated(_user, 500000);
         } else {
             // Caso: punteggio molto basso (<500k)
-            uint16[3] memory penalties = [uint16(280), uint16(450), uint16(720)];
+            uint16[3] memory penalties = [280, 450, 720];
             conditionOfFee[_user].penalityForLiquidation = penalties;
             emit ScoreUpdated(_user, 499999);
         }
     }
```

[L-3] Inefficient Gas Usage Due to Repeated Array Length Access in Loops

---

Description:

In the `PactContract` contract, the functions `_upDateRewardBuy` and `_upDateRewardSell` iterate over the `pact[_id].rewardMaturity` array without caching its length and also the `safeBatchTransferFrom` function. Each loop condition access to `pact[_id].rewardMaturity.length` and `ids.length` triggers a storage read . Caching the array length in a local variable would reduce gas costs.

Impact:

Repeated SLOADs increase gas consumption in `_upDateRewardBuy` and `_upDateRewardSell`, and in `safeBatchTransferFrom` with multiple IDs. The impact is low due to `MAX_REWARDS` limiting array size, but gas costs accumulate, raising transaction fees slightly.

Recommended Mitigation:

- In `_upDateRewardBuy`:

```
 function _upDateRewardBuy(uint _id, address _user, uint qty) internal {
     uint64 time = uint64(block.timestamp);
+    uint8 length = uint8(pact[_id].rewardMaturity.length);
     // @audit cache array length
-    for (uint8 i = 0; i < pact[_id].rewardMaturity.length; i++) {
+    for (uint8 i = 0; i < length; i++) {
         if (time < pact[_id].rewardMaturity[i]) {
             rewardToClaim[_id][_user][i] += qty;
         }
     }
 }
```

- In `_upDateRewardSell`

```
 function _upDateRewardSell(uint _id, address _user, uint qty) internal {
     uint64 time = uint64(block.timestamp);
+    uint length = pact[_id].rewardMaturity.length;
     // @audit cache array length here too
-    for (uint i = 0; i < pact[_id].rewardMaturity.length; i++) {
+    for (uint i = 0; i < length; i++) {
         if (time < pact[_id].rewardMaturity[i]) {
             if (rewardToClaim[_id][_user][i] >= qty) {
                 rewardToClaim[_id][_user][i] -= qty;
             } else {
                 rewardToClaim[_id][_user][i] = 0;
             }
         }
     }
 }
```

- In `safeBatchTransferFrom`:

```
function safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) public override whenNotPaused nonReentrant correctAddress(to) {
    require(
```

```
    ids.length == amounts.length,
        "ERC1155: ids and amounts length mismatch"
    );
    // Other logic
    // Update scheduled reward ownership for each ID, and enforce 'firstTransfer'
rules
+    uint256 idLength = ids.length;
-    for (uint256 i = 0; i < ids.length; ++i) {
+    for (uint256 i = 0; i < idLength; ++i) {
```

Tools Used:

- Manual code review
- Static analysis