DreamStarter Protocol Security Review

- Prepared by: LamsyA
- Reviewed Version: polygonscan.com
- Contract Address: 0xDD06EAAf2eDA8D4bFF2898cAc084e8330404F092

Figure 1: DreamStarter.xyz

Audited by LamsyA

- Twitter: @lamsy
- Email: olamide.adetula.1@gmail.com
- Telegram: Lamsy
- GitHub: LamsyA
- Discord: LamsyA

Protocol Summary

The DreamStarter protocol employs a bonding curve mechanism for dynamic pricing, where the price of tokens increases with supply through an exponential formula. This mechanism also facilitates automated market making. Upon reaching the funding goal, an automatic Uniswap V3 pool is created, integrating with WETH for trading pairs and allowing for custom tick range and fee tier settings.

Key features of DreamStarter include a token creator management system, pausable functionality for emergency stops, a protocol fee collection mechanism, and customizable token metadata. The core components of the protocol are:

- **IdeaFactory**: The main contract for token creation and management.
- **Idea**: The ERC20 token implementation.
- **IdeaFactoryMath**: Provides mathematical utilities for pricing.
- **IdeaFactoryLiquidity**: Handles Uniswap V3 integration.

The protocol aims to provide a robust platform for tokenizing ideas while ensuring liquidity through automated market making and bonding curve mechanics. The target network for DreamStarter is the Polygon Mainnet.

Disclaimer

This security review is not to be considered as a security guarantee. The review is conducted to the best of my knowledge according to industry best practices and understanding of the codebase at the time of review.

    * Likelihood Explanation
    * Recommendation
  - Gas Savings in Idea.sol - Summary - Finding Description - Impact Explanation - Recommendation

Audit Details

Impact	High	Medium	Low	Informational	Gas Optimization
High	1	0	0	0	0
Medium	0	2	0	0	0
Low	0	0	2	0	0
Informational	0	0	0	4	0
Gas Optimization	0	0	0	0	2

Scope

The audit covers the core smart contracts of the DreamStarter protocol deployed on Polygon Mainnet:

Contract	SLOC	Purpose
IdeaFactory.sol	173	Core contract for idea token creation and management
IdeaFactoryMath.sol	36	Handles bonding curve calculations
IdeaFactoryLiquidity.sol	107	Manages Uniswap V3 liquidity integration
Idea.sol	15	ERC20 token implementation for ideas
IdeaFactoryErrors.sol	14	Custom error definitions
IdeaFactoryStorage.sol	20	Storage layout and variables
IdeaFactoryTypes.sol	71	Type definitions and structs
Total SLOC:	**436**	

Files in Scope

```
Files in Scope:
* src/IdeaFactory.sol
* src/IdeaFactoryLiquidity.sol
* src/Idea.sol
* src/IdeaFactoryMath.sol
```

Issues found

The audit identified several issues categorized by their severity:

High Severity

- **Unrestricted Withdrawal Access Compromises Protocol Funds**: The withdrawal mechanism lacks fund segregation, posing a high risk of breaking protocol functionality.

Medium Severity

- **Zero Slippage Protection in Initial Liquidity Provision**: The `_prepareMintParams` function sets minimum amounts to zero, potentially allowing unfavorable execution prices.
- **Unhandled Excess Payments in Token Operations**: The contract allows users to send excess ETH without refund mechanisms.

Low Severity

- **State Modification in Read-Only Function**: The `getIdeaToken` function modifies contract state while appearing to be a read-only getter function, leading to unexpected gas costs and potential integration issues. This function should be updated to follow proper view function behavior.
- **Missing Event Emissions in Critical State Changes**: Several critical functions lack event emissions, impacting transparency and off-chain monitoring capabilities. Functions such as `createIdeaToken`, `buyIdeaToken`, `updateIdeaToken`, and `setIdeaStatus` should emit events to ensure proper tracking and monitoring of state changes.

Informational

- **Meaningless Return Values in State-Changing Functions**: Multiple functions return boolean values that don't reflect the actual state changes.
- **Lack of Public Price Calculation Function**: The `calculateCost` function is not publicly accessible, making it difficult for users to determine the exact ETH amount needed for token purchases.
- **Redundant Event Declaration**: The `ParamsPositionManager` event is redundant as the same information is captured in the `LiquidityProvided` event.

Gas Optimization

- **Constants Declared as Local Variables Instead of Immutable**: Constants are declared as local variables instead of immutable contract-level variables, leading to higher gas costs.
- **Gas Savings in Idea.sol**: The `mint` function returns a boolean value that is always true, consuming unnecessary gas. The `owner` state variable can be declared as immutable to save gas.

Findings

High

Unrestricted Withdrawal Access Compromises Protocol Funds

Summary

The current withdrawal mechanism lacks fund segregation, creating significant complexity and posing a high risk of breaking protocol functionality—even with trusted ownership.

Finding Description

The `withdraw` function in the`IdeaFactory` contract allows the owner to withdraw the entire contract balance, mixing protocol fees and user funds intended for liquidity provision:

```
function withdraw() external onlyOwner nonReentrant {
    uint256 amount = address(this).balance;
    (bool success, ) = owner().call{value: amount}("");
    require(success, "Withdrawal failed");
}
```

This creates a critical vulnerability because:

- Contract balance includes both protocol fees and user funds for liquidity.
- No separation between different fund types.
- withdraw function can be called before liquidity provision is completed.
- No tracking of withdrawable protocol fees.

Impact Explanation

Impact is classified as **HIGH** because:

- Can completely break protocol functionality.
- Users lose funds meant for liquidity provision.
- Prevents Uniswap pool creation.
- Breaks bonding curve mechanics.
- No recovery mechanism.

Likelihood Explanation

Likelihood is **HIGH** because:

- Complex fund management increases the risk of errors.
- Operations involving multiple tokens magnify confusion.
- No safeguards exist to prevent accidental full withdrawals.
- Timing-sensitive operations increase the chances of mistakes.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/IdeaFactory.sol";
import "../src/Idea.sol";
import "../src/IdeaFactoryStorage.sol";
import "../src/IdeaFactoryLiquidity.sol";


//  to run test "forge test --match-test testBuyIdeaToken  -vvvv --fork-url mainnet --fork-block-number
contract IdeaFactoryTest is Test {
    IdeaFactory public factory;
    address public owner;
    address public user1;
    address public user2;
    uint256 public IDEATOKEN_CREATION_FEE = 0.01 ether;
    uint256 public constant DECIMALS = 1e18;
    uint256 public constant MAX_SUPPLY = 1000000000 * DECIMALS;
    uint256 public constant INIT_SUPPLY = MAX_SUPPLY / 5;
    IdeaFactory.CreateIdeaTokenParams public params;
    IdeaFactory.CreateIdeaTokenParams public  params2;

    // Events to test
    event LiquidityProvided(
        address indexed token,
        uint256 tokenId,
        uint128 liquidity,
        uint256 amount0,
        uint256 amount1
    );

    error InsufficientCreationFee(uint256 provided, uint256 required);
    error CantAcceptDonation();
```

```
    function setUp() public {
        owner = makeAddr("owner");
        user1 = makeAddr("user1");
        user2 = makeAddr("user2");
        vm.prank(owner);
        factory = new IdeaFactory();
        vm.label(user1, "user1");
        vm.label(user2, "user2");
        vm.label(owner, "owner");
        vm.label(address(factory), "factory");

        // Fund test accounts
        vm.deal(user1, 100 ether);
        vm.deal(user2, 100 ether);

        params = IdeaFactoryTypes.CreateIdeaTokenParams({
            name: "Test Token",
            symbol: "TEST",
            description: "Test Description",
            imageUrl: "http://test.com/image",
            productUrl: "http://test.com",
            categories: "Test Category",
            productScreenshotUrl: "http://test.com/screenshot",
            twitterUrl: "http://twitter.com/test",
                telegramUrl: "http://t.me/test"
            });

            params2 = IdeaFactoryTypes.CreateIdeaTokenParams({
                name: "Token 2",
                symbol: "T2",
                description: "Test 2",
                imageUrl: "http://test2.com/image",
                productUrl: "http://test2.com",
                categories: "Category 2",
                productScreenshotUrl: "http://test2.com/screenshot",
                twitterUrl: "http://twitter.com/test2",
                telegramUrl: "http://t.me/test2"
            });
    }
function testWithdrawVulnerability() public {
    // Setup
     vm.startPrank(user1);



       address tokenAddress = factory.createIdeaToken{value: IDEATOKEN_CREATION_FEE}(params);
       address tokenAddress2 = factory.createIdeaToken{value: IDEATOKEN_CREATION_FEE}(params2);
        // Idea token = Idea(tokenAddress);
        factory.buyIdeaToken{value: 42 ether}(tokenAddress, 1e6);
        factory.buyIdeaToken{value: 10 ether}(tokenAddress2, 10e6);
        factory.buyIdeaToken{value: 13 ether}(tokenAddress2, 10e6);
        factory.buyIdeaToken{value: 23 ether}(tokenAddress2, 10e6);
       uint256 balance = address(tokenAddress).balance;
       vm.expectRevert(CantAcceptDonation.selector);
```

```
        payable(address(factory)).transfer(1 ether);
         vm.stopPrank();
         vm.deal(owner, 100 ether);
         vm.startPrank(owner);
         uint256 balOfOwner = address(owner).balance;
         console.log("Owner Balance: ", balOfOwner);
        uint256 factoryBalanceBefore = address(factory).balance;
        uint256 token2BalanceBefore = address(tokenAddress2).balance;
        uint256 tokenBalanceBefore = address(tokenAddress).balance;
        console.log("token  Balance before: ", tokenBalanceBefore);
        console.log("token 2 Balance before: ", token2BalanceBefore);
        console.log("Factory Balance before: ", factoryBalanceBefore);
        factory.withdraw();
        uint256 balOfOwnerAfter = address(owner).balance;
        console.log("Owner Balance after: ", balOfOwnerAfter);
        vm.expectRevert();
        factory.buyIdeaToken{value: 1 ether}(tokenAddress, 1e6);


    //    assertGt(balOfOwnerAfter, balOfOwner);
        uint256 balance2 = address(factory).balance;
          console.log("Balance: ", balance);
          console.log("Balance: ", balance2);

         IdeaFactory.IdeaTokenFull memory tokenFullData = factory.getIdeaToken(tokenAddress);
         console.log(" Token Full Data: ", tokenFullData.creationTimestamp);
         console.log(" Token fundingRaised : ", tokenFullData.fundingRaised);


         vm.stopPrank();
}
```

output This will revert with Out of funds error.

Recommendation

Implement proper fund segregation and tracking:

```
contract IdeaFactory {
    uint256 public protocolFees;

    function createIdeaToken(...) {
        // ... other code ...
        protocolFees += msg.value;
    }

    function withdraw() external onlyOwner nonReentrant {
        uint256 amount = protocolFees;
        require(amount > 0, "No fees to withdraw");

        protocolFees = 0;  // Reset before transfer

        (bool success, ) = owner().call{value: amount}("");
        require(success, "Withdrawal failed");
```

```
        emit ProtocolFeesWithdrawn(owner(), amount);
    }
}
}
```

This ensures:

- Only protocol fees can be withdrawn.
- Liquidity funds remain safe.
- Proper accounting.
- Event emission for transparency.

Medium

Zero Slippage Protection in Initial Liquidity Provision

Severity: Medium

Summary

The `_prepareMintParams` function in `IdeaFactoryLiquidity.sol` sets minimum amounts to zero during liquidity provision, potentially allowing unfavorable execution prices.

Vulnerability Details

```
return INonfungiblePositionManager.MintParams({
    // ...
  @>  amount0Min: 0,
  @>  amount1Min: 0,
    deadline: block.timestamp
});
```

The function:

- Is called during initial liquidity provision
- Sets both minimum amounts to 0

Impact

Rated as **MEDIUM** severity because:

Protected Factors:

- Pool creation and price initialization are atomic operations
- Initial price is deterministically set:

```
uint256 priceAtFundingGoal = (INIT_SUPPLY * 1e18 ) / IDEACOIN_FUNDING_GOAL;
uint160 sqrtPriceX96 = calculateSqrtPriceX96(priceAtFundingGoal);
```

Risk Factors:

- Protocol funds (`IDEACOIN_FUNDING_GOAL amount`) are at stake
- No minimum execution price guarantee
- Potential for unfavorable execution in volatile market conditions
- Each new idea token deployment is affected

Recommendations

Implement slippage protection:

```
amount0Min: ( amount0Desired * 850 )/ 1000,  // 1.5% slippage protection
amount1Min: ( amount1Desired * 850 )/ 1000,  // 1.5% slippage protection
```

References

- Uniswap V3 Documentation on Slippage Protection
- Similar issues: Lack of slippage protection in liquidity provision

The implementation of these recommendations would provide necessary protection while maintaining protocol flexibility.

Unhandled Excess Payments in Token Operations

Severity: Medium

Summary

The contract allows users to send excess ETH in both `createIdeaToken` and `buyIdeaToken` functions without refund mechanisms.

Finding Description

Two functions accept payments without exact amount validation:

- `createIdeaToken` only checks for minimum fee:

```
if (msg.value < IDEATOKEN_CREATION_FEE) {
    revert InsufficientCreationFee({
        provided: msg.value,
        required: IDEATOKEN_CREATION_FEE
    });
}
```

- `buyIdeaToken` allows overpayment:

```
if (msg.value < requiredEth) {
    revert InsufficientEthSent({
        sent: msg.value,
        required: requiredEth
    });
}
```

Impact Explanation

Impact is **MEDIUM** because:

- Users can lose funds through overpayment.
- ETH becomes trapped in the contract.
- Poor user experience.
- No recovery mechanism for excess funds.

Likelihood Explanation

High likelihood as:

- Users often send round numbers of ETH.

- Frontend calculations may be imprecise.
- Gas price fluctuations can lead to overestimation.

Recommendation

For `createIdeaToken`:

```
if (msg.value != IDEATOKEN_CREATION_FEE) {
    revert InvalidPaymentAmount(msg.value, IDEATOKEN_CREATION_FEE);
}
```

For `buyIdeaToken`:

```
 if (msg.value != requiredEth) {
            revert InsufficientEthSent({
                sent: msg.value,
                required: requiredEth
            });
        }
```

Low

State Modification in Read-Only Function

Severity: Low

Summary

The `getIdeaToken` function in `IdeaFactory.sol` modifies contract state while appearing to be a read-only getter function, leading to unexpected gas costs and potential integration issues.

Finding Description

```
function getIdeaToken(address ideaTokenAddress) external returns (IdeaTokenFull memory) {
    IdeaTokenBase storage base = tokenBase[ideaTokenAddress];
    if (base.tokenAddress == address(0)) revert TokenNotFound(ideaTokenAddress);

    IdeaTokenData storage data = tokenData[ideaTokenAddress];
    Idea ideaTokenCt = Idea(ideaTokenAddress);
    data.tokenCurrentSupply = ideaTokenCt.totalSupply(); // @audit State modification

    return IdeaTokenFull({
        // ... struct fields ...
    });
}
```

The function:

- Has a `get` prefix suggesting read-only behavior
- Updates storage variable `tokenCurrentSupply`
- Returns token data structure
- Missing `view` modifier despite name implying view function

Impact Explanation

Impact is **LOW** because:

- Unexpected gas costs for what appears to be a read operation
- Could break integrations expecting view behavior
- Affects contract composability
- Unnecessary state updates on every read

Likelihood Explanation

Likelihood is **HIGH** because:

- Function is primary method to get token data
- Likely to be frequently called by:
  - Frontend applications
  - Other smart contracts
  - Protocol integrations
- Every call modifies state unnecessarily

Recommendation

```
// Read-only getter
function getIdeaToken(address ideaTokenAddress) external view returns (IdeaTokenFull memory) {
    IdeaTokenBase storage base = tokenBase[ideaTokenAddress];
    if (base.tokenAddress == address(0)) revert TokenNotFound(ideaTokenAddress);

    IdeaTokenData storage data = tokenData[ideaTokenAddress];
    Idea ideaTokenCt = Idea(ideaTokenAddress);

    return IdeaTokenFull({
        // ... other fields ...
        tokenCurrentSupply: ideaTokenCt.totalSupply(), // Read directly
        // ... remaining fields ...
    });
}
```

- Follows proper view function behavior
- Reduces gas costs for read operations
- Improves contract integration capabilities.

Missing Event Emissions in Critical State Changes

Severity: Low

Summary

The `IdeaFactory` contract lacks event emissions for critical state changes in token creation and management functions, impacting transparency and off-chain monitoring capabilities.

Finding Description

Several critical functions lack event emissions:

- `createIdeaToken()` - No event when new tokens are created
- `buyIdeaToken()` - No event when tokens are purchased
- `updateIdeaToken()` - No event when token metadata is updated
- `setIdeaStatus()` - No event when token status changes
- `_provideLiquidity` - No event when liquidity is provided

This impacts:

- Off-chain monitoring systems
- Frontend synchronization
- Protocol analytics
- User transaction tracking

Impact Explanation

Impact is **LOW** because:

- Difficult to track protocol activity
- Reduced transparency for users and integrators
- Complicates protocol analytics
- No historical record of important state changes

Likelihood Explanation

Likelihood is **HIGH** because:

- Affects every token creation
- Impacts all token updates
- No workaround available
- Essential for protocol monitoring

Proof of Concept

Recommendation

Add appropriate events:

```
contract IdeaFactory {
    event IdeaTokenCreated(
        address indexed tokenAddress,
        address indexed creator,
        string name,
        string symbol,
        uint256 timestamp
    );

    event IdeaTokenUpdated(
        address indexed tokenAddress,
        string description,
        string productUrl,
        address indexed updater
    );

    event IdeaStatusChanged(
        address indexed tokenAddress,
        bool newStatus,
        address indexed updater
    );

      event IdeaTokenPurchased(
        address indexed tokenAddress,
        address indexed buyer,
        uint256 ethAmount,
        uint256 tokenAmount,
        uint256 timestamp
```

```
    );

    function createIdeaToken(CreateIdeaTokenParams calldata params) external payable returns (address) {
        // existing logic...

        emit IdeaTokenCreated(
            tokenAddress,
            msg.sender,
            params.name,
            params.symbol,
            block.timestamp
        );
        return tokenAddress;
    }

  function buyIdeaToken(address tokenAddress, uint256 tokenQty) external payable {
        // existing logic...

        // Emit event after successful purchase
        emit IdeaTokenPurchased(
            tokenAddress,
            msg.sender,
            msg.value,
            tokenQty,
            block.timestamp
        );
    }

    function updateIdeaToken(address tokenAddress, string calldata description, string calldata productU
        // existing logic...

        emit IdeaTokenUpdated(
            tokenAddress,
            description,
            productUrl,
            msg.sender
        );
    }

    function setIdeaStatus(address tokenAddress, bool newStatus) external {
        // existing logic...

        emit IdeaStatusChanged(
            tokenAddress,
            newStatus,
            msg.sender
        );
    }
}
```

The LiquidityProvided event is defined but never emitted when liquidity is provided through `_provideLiquidity`.

```
function _provideLiquidity(address ideaTokenAddress, uint256 ethAmount, address _owner)
    internal
    returns (uint128)
```

```
{
    // ... existing logic ...

    (uint256 tokenId, uint128 liquidity, uint256 amount0, uint256 amount1) =
        positionManager.mint{value: ethAmount}(params);

    emit LiquidityProvided(
        ideaTokenAddress,
        tokenId,
        liquidity,
        amount0,
        amount1
    );

    return liquidity;
}
```

Informational

Meaningless Return Values in State-Changing Functions

Severity: Informational

Summary

Multiple functions in the `IdeaFactory` contract return boolean values that don't reflect the actual state changes, leading to unreliable function results.

Finding Description

Three functions return `true` regardless of their operation success:

- `updateIdeaToken`:

  `function updateIdeaToken(address ideaTokenAddress, UpdateIdeaTokenParams calldata params) external`

- `setIdeaStatus`:

  `function setIdeaStatus(address ideaTokenAddress, bool newStatus) external returns (bool)`

- `setCreationFeeInWei`:

  `function setCreationFeeInWei(uint256 newFee) external returns (bool)`

- `_provideLiquidity`:

  `function _provideLiquidity( address ideaTokenAddress,  uint256 ethAmount,  address _owner ) interna`

Impact Explanation

Impact is **LOW** because:

- No direct security vulnerability
- No fund loss potential
- Only affects contract integrations
- Can be worked around using events or direct state reads

Likelihood Explanation

Likelihood is **HIGH** because:

- Affects every function call
- No way to determine if state actually changed
- Integrators might rely on return values
- Multiple functions affected

Recommendation

Return meaningful status values that reflect actual state changes:

remove the return value from the function

- `setIdeaStatus`:
- `setCreationFeeInWei`:
- `updateIdeaToken`:
- `_provideLiquidity`:

Lack of Public Price Calculation Function

Severity: Informational

Summary

The `calculateCost` function is not publicly accessible, making it difficult for users to determine the exact ETH amount needed for token purchases.

Finding Description

The `calculateCost` function is internal:

```
function calculateCost(uint256 currentSupply, uint256 tokenQty) internal view returns (uint256) {
    // ... calculation logic
}
```

This means:

- Users can't calculate exact costs off-chain
- Frontend integration becomes more complex
- Higher chance of transaction failures due to incorrect ETH amounts
- Poor user experience when trying to determine purchase amounts

Impact Explanation

Impact is **INFORMATIONAL** because:

- No security vulnerability
- Doesn't affect protocol safety
- Only impacts user experience
- Can be worked around with off-chain calculations

Likelihood Explanation

Affects every token purchase transaction where users need to:

- Calculate required ETH
- Determine maximum purchase amount for available ETH

- Plan purchases based on price impact

Recommendation

Make the function public and pure:

```
function calculateCost(
    uint256 currentSupply,
    uint256 tokenQty
) public pure returns (uint256) {
    // existing calculation logic
}
```

Benefits:

- Users can calculate exact costs before transactions
- Better frontend integration
- Reduced failed transactions
- Improved user experience
- Enables price simulation tools

This change would significantly improve the protocol's usability without compromising security.

Redundant Event Declaration

Severity: Informational

Summary

The `ParamsPositionManager` event is redundant as the same information is captured in the `LiquidityProvided` event.

Finding Description

```
event ParamsPositionManager(
    address indexed token0,
    address indexed token1,
    uint24 fee,
    int24 tickLower,
    int24 tickUpper,
    uint256 amount0Desired,
    uint256 amount1Desired,
    uint256 amount0Min,
    uint256 amount1Min,
    address recipient
);
```

This event:

- Is never emitted in the contract
- Duplicates information from `LiquidityProvided`
- Increases contract size unnecessarily

Impact Explanation

Impact is **INFORMATIONAL** because:

- No security impact
- No functional impact

- Only affects code cleanliness
- Slightly increases deployment gas cost

Recommendation

Remove the redundant event declaration:

```
// Remove this event
event ParamsPositionManager(...);

// Keep the existing
event LiquidityProvided(
    address indexed token,
    uint256 tokenId,
    uint128 liquidity,
    uint256 amount0,
    uint256 amount1
);
```

Floating Pragma Used in Contracts

Severity: Informational

Summary

The contracts use floating pragmas, which could lead to inconsistent compiler versions being used across different deployments.

Finding Description

```
// Current implementation
pragma solidity ^0.8.0;  // @audit Floating pragma
```

Found in:

- IdeaFactory.sol
- IdeaFactoryLiquidity.sol
- Idea.sol
- IdeaFactoryMath.sol

The caret (^) prefix in the pragma allows the compiler to use any version from 0.8.0 to 0.8.x, which could lead to:

- Different bytecode being deployed
- Inconsistent behavior across deployments
- Potential incompatibility issues

Impact Explanation

Impact is **INFORMATIONAL** because:

- No direct security vulnerability
- Could affect deployment consistency
- Best practice recommendation
- Important for long-term maintenance

Likelihood Explanation

Likelihood is **HIGH** because:

- Affects all contract deployments
- Different development environments might use different versions
- Future deployments might use newer compiler versions

Recommendation

Lock the pragma to a specific version:

```
// Use a fixed pragma version
pragma solidity 0.8.27;  // Or the specific version intended
```

Benefits:

- Consistent bytecode across deployments
- Predictable behavior
- Better deployment tracking
- Clearer dependency management

This should be applied to all contracts in the codebase to ensure consistency.

Gas

Constants Declared as Local Variables Instead of Immutable

Severity: Gas Optimization

Summary

In `_prepareMintParams`, constants are declared as local variables instead of immutable contract-level variables.

Finding Description

```
function _prepareMintParams(...) private view returns (...) {
    int24 TICK_SPACING = 60;
    int24 tickRange = 10000;
    // ... rest of function
}
```

These values never change and are redeclared on every function call.

Impact Explanation

Impact is **GAS** because:

- Unnecessary memory allocation on each call
- Higher gas costs for repeated declarations
- Less gas efficient than immutable variables

Likelihood Explanation

Occurs on every liquidity provision operation.

Recommendation

Declare as immutable contract variables:

```
contract IdeaFactoryLiquidity {
    int24 private immutable TICK_SPACING = 60;
    int24 private immutable TICK_RANGE = 10000;
}
```

Gas Savings in Idea.sol

Severity: Gas Optimization

Summary

- The `mint` function in `Idea.sol`returns a boolean value that is always true, consuming unnecessary gas. change
- The owner state variable is declared as public but is only set once in the constructor and never changed, making it a candidate for the immutable keyword.

```
address public owner;
```

to

```
address public immutable owner;
```

Finding Description

```
function mint(uint mintQty, address receiver) external returns(bool) {
    if(msg.sender != owner) revert OnlyOwnerCanMint();
    _mint(receiver, mintQty);
    return true; // @audit Unnecessary return value
}
```

Impact Explanation

Impact is **GAS** because:

- Unnecessary storage of return value
- Extra gas consumed for every mint operation
- Return value is never false

Recommendation

Remove the return value:

```
function mint(uint mintQty, address receiver) external {
    if(msg.sender != owner) revert OnlyOwnerCanMint();
    _mint(receiver, mintQty);
}
```

This saves gas by:

- Eliminating unnecessary return value
- Reducing function execution cost
- Simplifying function signature
- Reduces gas cost for reading owner address
- Saves ~2100 gas per read