hashlock.

Security Audit

Blockstreet 2nd (DeFI)

September 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

Executive Summary

The Blockstreet team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

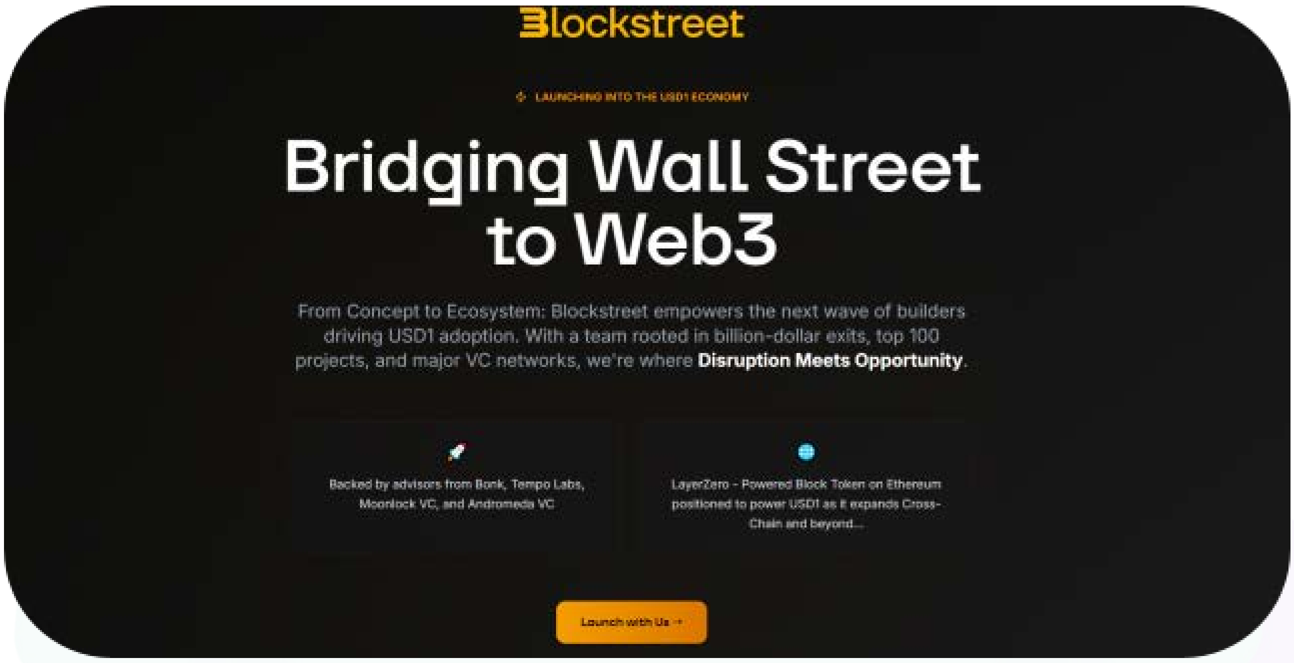
Blockstreet is a multichain platform launched with the goal of driving the adoption of USD1, a regulated stablecoin created by World Liberty Financial, serving as a hub for developers and projects in sectors such as DeFi, payments, gaming, and real-world assets (RWA). Built on LayerZero technology, the platform provides infrastructure for project launches (launchpad), USD1 liquidity, cross-chain support, community governance, and modular compliance tools (KYC/AML) to foster institutional adoption and innovation within the traditional-finance Web3 ecosystem. The native token BLOCK, with a maximum supply capped at 1 billion, is designed for utility, governance, and revenue sharing with holders, while also representing direct exposure to all projects created on the platform.

Project Name: Blockstreet
Project Type: Defi
Compiler Version: ^0.8.0
Website:https://www.blockstreet.xyz/
Logo:

Visualised Context:

Project Name	BLOCKSTREET
Launch Date	TBA
Compiler Version	v^0.8.0
Language	SOLIDITY
Network	ETHEREUM
Token Ticker	$BLOCK

Project Visuals:


Blockstreet

LAUNCHING INTO THE USD1 ECONOMY

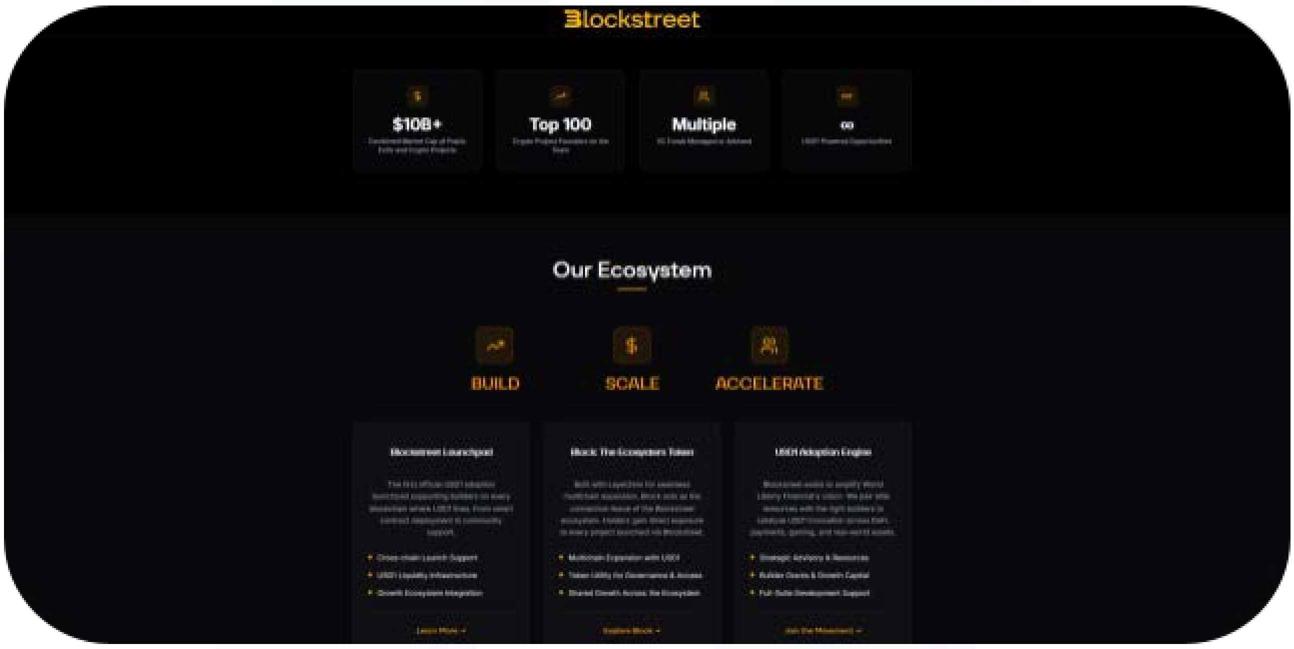
Bridging Wall Street to Web3

From Concept to Ecosystem: Blockstreet empowers the next wave of builders driving USD1 adoption. With a team rooted in billion-dollar exits, top 100 projects, and major VC networks, we're where **Disruption Meets Opportunity**.

Backed by advisors from Bonk, Tempo Labs, Moonlock VC, and Andromeda VC

LayerZero - Powered Block Token on Ethereum positioned to power USD1 as it expands Cross-Chain and beyond...

Launch with Us →


Blockstreet

$10B+
Combined Market Cap of Public Exits and Crypto Projects

Top 100
Crypto Project Founders on the Team

Multiple
VC Funds Managers Advised

∞
USD1 Powered Opportunities

Our Ecosystem

BUILD

SCALE

ACCELERATE

Blockstreet Launchpad

The first official USD1 adoption launchpad supporting builders on every blockchain where USD1 lives. From smart contract deployment to community support.

- Cross-chain Launch Support
- USD1 Liquidity Infrastructure
- Growth Ecosystem Integration

Learn More →

Block: The Ecosystem Token

Built with LayerZero for seamless multichain expansion, Block acts as the connective tissue of the Blockstreet ecosystem. Holders gain direct exposure to every project launched via Blockstreet.

- Multichain Expansion with USD1
- Token Utility for Governance & Access
- Shared Growth Across the Ecosystem

Explore Block →

USD1 Adoption Engine

Blockstreet exists to amplify World Liberty Financial's vision. We pair elite resources with the right builders to catalyze USD1 innovation across DeFi, payments, gaming, and real-world assets.

- Strategic Advisory & Resources
- Builder Grants & Growth Capital
- Full-Suite Development Support

Join the Movement →

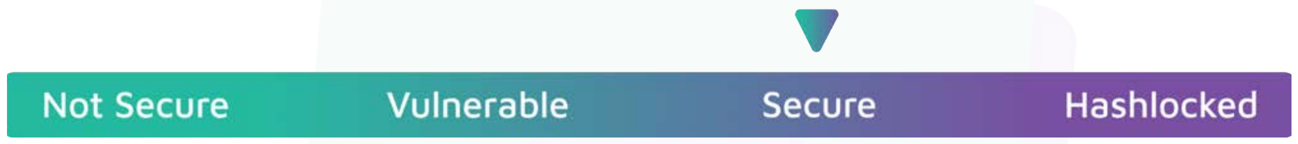
Audit Scope

We at Hashlock audited the solidity code within the Blockstreet project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Blockstreet Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**September, 2025**
Contract 1	BondingCurve.sol
Contract 2	Factory.sol
Contract 3	Foundry.sol
Contract 4	Lock.sol
Contract 5	TokenImplementation.sol
Contract 6	UniswapPoolCreator.sol
Contract 7	BondingMath.sol
Audited GitHub Commit Hash	58e493fdca13ad36bf6364ba8cfe6c520be6af61
Fix Review GitHub Commit Hash	d3a21dacce782b6c35e9aeefc5c160445badd51a

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.



The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved or acknowledged.

Hashlock found:

3 High severity vulnerabilities

4 Medium severity vulnerabilities

11 Low severity vulnerabilities

2 Gas Optimisations

3 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
TokenImplementation.sol Standard ERC20 token with controlled one-time minting for bonding curve integration. Allows the factory to: - Mint total supply to bonding curve (one-time only) Allows users to: - Standard ERC20 functions (transfer, approve, etc.)	**Contract achieves this functionality.**
Foundry.sol Main deployment hub that creates Factory and Lock contract instances using a minimal proxy pattern. Allows users to: - Deploy new Factory and Lock contract pairs Allows owner to: - Update implementation contracts for future deployments - Set deployment fees for system creation - Withdraw accumulated deployment fees - Emergency pause/unpause functionality - Transfer contract ownership	**Contract achieves this functionality.**
Factory.sol Deploys individual bonding curve systems (token +	**Contract achieves this functionality.**

bonding curve pairs) with configurable parameters. Allows users to: - Deploy complete token launch system Allows owner to: - Update global parameters for new deployments - Set fees for bonding curve system deployment - Withdraw accumulated deployment fees - Transfer contract ownership	
BondingCurve.sol Manages three-phase token distribution system with automated Uniswap V3 integration. Allows users to: - Contribute ETH for fixed token allocation - Purchase tokens with ETH using bonding curve pricing - Sell tokens for ETH - Finalize curve and create Uniswap V3 pool and lock LP NFT - Withdraw pre-bonding token allocations Allows owner to: - Withdraw accumulated trading fees - Transfer contract ownership	**Contract achieves this functionality.**
Lock.sol Manages Uniswap V3 LP NFT locking and fee collection for 10-year periods.	**Contract achieves this functionality.**

Allows users to: - Lock Uniswap V3 LP NFT - Unlock NFT after 10-year lock period - Claim accumulated trading fees from locked position	
BondingMath.sol Provides mathematical functions for bonding curve calculations including token/ETH conversions, price calculations, and sell fee computations.	**Contract achieves this functionality.**
UniswapPoolCreator.sol Handles Uniswap V3 pool creation, initialization with proper price calculations, and LP position management with full-range liquidity provision.	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Blockstreet project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring were recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Blockstreet project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	**Description**
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] UniswapPoolCreator#createAndInitializePool,#createLPPosition - DoS in `finalizeCurve()` when Uniswap pool already exists

Description

The `createAndInitializePool` function will revert if the Uniswap V3 pool has already been created, causing a Denial of Service (DoS) of the `finalizeCurve()` function.

Vulnerability Details

The `createAndInitializePool()` function calls `factory.createPool(params.token, params.weth, params.fee)` without checking if the pool already exists. According to the Uniswap V3 Factory contract, the `createPool()` function will revert if a pool with the same token pair and fee tier already exists.

Additionally, the function calls `IUniswapV3Pool(pool).initialize(sqrtPriceX96)` which will also revert if the pool has already been initialized. This means that if the pool was already created externally, which is possible even without providing liquidity, the entire function will fail, leaving the bonding curve in an unfinalized state.

Lastly, if the pool already exists and is properly initialized, there's an additional risk with the slippage protection in `createLPPosition()`. The function uses a hardcoded `10%` slippage tolerance.

If the pool was created by someone else and was just initialized with a very different price, or if the current pool price differs significantly from the bonding curve's calculated price due to prior trading activity, the actual amounts of assets provided during the `INonfungiblePositionManager.mint()` call could exceed the 10% slippage tolerance causing the `finalizeCurve()` function to revert. Although, price could be temporarily deviated and could be arbitraged back close to the bonding curve's calculated price, there is no guarantee that this will happen.

Impact

The `finalizeCurve()` function will permanently revert if the Uniswap V3 pool already exists, causing a permanent denial of service that prevents the bonding curve from being finalized and results in the effective loss of all ETH and tokens locked in the contract.

Recommendation

Implement proper checks before creating and initializing the pool:

1. Check if the pool already exists using `factory.getPool(token0, token1, fee)`

2. If the pool exists, verify it's initialized by checking `pool.slot0().sqrtPriceX96 != 0`

Consider calling the `createAndInitializePoolIfNecessary()` function from Uniswap V3 Periphery's position manager contract instead of manually creating and initializing the pool. In that case, first make sure to set the correct order of the token addresses as the function validates them (`token0 < token1`).

See
https://github.com/Uniswap/v3-periphery/blob/main/contracts/base/PoolInitializer.sol

3. For the slippage protection issue, consider implementing one or more of the following strategies:

- Dynamic Slippage Calculation
  - Calculate slippage tolerance based on current pool state and price deviation when dealing with pre-existing pools
  - Implement price validation to ensure the pool price is within acceptable bounds
- Emergency Recovery Function
  - Add an `emergencyFinalizeCurve()` function that can be called by the owner after a time delay if `finalizeCurve()` was not called yet (e.g., 48-72 hours)
  - This function should do the same as `finalizeCurve()` but allow custom slippage parameters or alternative liquidity provision strategies

4. Finally, given the mentioned issues and risks, that could cause the `finalizeCurve()` to revert, blocking users who participated in the pre-bonding from calling `withdrawTokenAllocation()` to withdraw their tokens, consider updating the `withdrawTokenAllocation()` function to instead of validate that `if (!isFinalized) revert CannotFinalizeYet()`, validate that `currentPhase` is the same as the required for the `finalizeCurve()` function to be called. This will maintain the same functionality of only allowing the users to call `withdrawTokenAllocation()` after the `bondingTarget` is met and does not affect any of the operations of the `finalizeCurve()` function and avoid locking users tokens in case `finalizeCurve()` cannot be permanent or temporarily executed.

Status

Resolved

[H-02] Lock#_removeNFTFromOwner - Unbounded array iteration allows permanent lock DoS

Description

The `_removeNFTFromOwner()` function uses linear search to find and remove NFTs from the owner's array, making it vulnerable to a Denial of Service (DoS) attack where an attacker can prevent legitimate NFT owners from unlocking their positions.

Vulnerability Details

The `_removeNFTFromOwner()` function iterates through the `ownerNFTs[owner]` array to find the specific `tokenId` to remove. This creates an O(n) complexity where `n` is the number of NFTs locked for a specific owner. The `lockNFT()` function allows anyone to lock an NFT for any address.

Given that the `BondingCurve` locks the NFT for the owner of the contract, which is known at the moment of calling `finalizeCurve()`, an attacker can exploit this by:

1. Creating thousands of small Uniswap V3 positions (with minimal liquidity)

2. Calling `lockNFT()` for each position, specifying the target owner's address

3. When `finalizeCurve()` is called, the NFT created will be locked, placing the `tokenId` at the end of the array

4. After the lock period ends, when the legitimate owner tries to unlock their NFT, `_removeNFTFromOwner()` must iterate through all attacker-created positions first, reverting if the transaction goes out of gas because of the high amount of items in the array.

While the attack will require a lot of gas to create the positions and lock them, it will be feasible on cheap networks.

Impact

Allow attackers to permanently prevent legitimate NFT owners from unlocking their positions, effectively causing permanent loss of access to locked assets.

Recommendation

Replace the linear array search with OpenZeppelin's `EnumerableSet` library, which provides O(1) removal operations. Implement the following changes:

1. Import `@openzeppelin/contracts/utils/structs/EnumerableSet.sol`

2. Change `mapping(address => uint256[]) public ownerNFTs` to `mapping(address => EnumerableSet.UintSet) private ownerNFTs`

3. Replace array operations with `EnumerableSet` methods `ownerNFTs[owner].add(tokenId)`, `ownerNFTs[owner].remove(tokenId)` and `ownerNFTs[owner].values()`

Status

Resolved

[H-03] BondingCurve#contributePreBonding - PreBonding phase deadlock

Description

The `preBonding` phase can be permanently stuck when the remaining amount needed to reach the preBondingTarget is less than the minimum contribution amount, resulting in permanent lockup of all contributed ETH funds and allocated tokens.

Vulnerability Details

In the `contributePreBonding()` function, there are two validation checks that can create a deadlock scenario:

```
if (msg.value < settings.minContribution) revert ContributionTooLow();

uint256 newTotal = totalPreBondingContributions + msg.value;

if (newTotal > settings.preBondingTarget) revert PreBondingTargetReached();
```

The issue occurs when the remaining amount needed, `settings.preBondingTarget - totalPreBondingContributions`, is less than `settings.minContribution`.

In this scenario any contribution less than `minContribution` will revert with `ContributionTooLow()`, and any contribution equal to or greater than `minContribution` will revert with `PreBondingTargetReached()` if it would exceed the target.

A legitimate user or malicious actor can intentionally trigger this condition by contributing a small amount when the total contributions are getting close to the target, leaving a gap smaller than `minContribution`.

Impact

The `preBonding` phase cannot be completed, preventing transition to the `bonding` phase. This vulnerability results in permanent loss of all contributed ETH funds and allocated tokens.

Recommendation

Consider implementing a mechanism to automatically complete the `preBonding` phase when the remaining amount is less than the minimum contribution, or update the validation to `newTotal > settings.preBondingTarget + settings.minContribution` to allow extra contributions up to min.

Status

Resolved

Medium

[M-01] BondingCurve#sellTokens - `totalETHCollected` not decremented on token sales

Description

The `totalETHCollected` variable tracks the total ETH collected by the bonding curve but is not decremented when users sell tokens back, allowing the bonding curve to reach the target threshold without actually collecting the expected amount of ETH.

Vulnerability Details

The `totalETHCollected` variable is incremented in the `buyTokens()` function when users purchase tokens, but it is never decremented in the `sellTokens()` function when users sell tokens back to the contract. This creates a discrepancy between the actual ETH held by the contract and the tracked amount in `totalETHCollected`.

The `totalETHCollected` is used to determine if the bonding target has been reached. This means that users can buy tokens, immediately sell them back, and the bonding curve will still consider the original purchase amount as "collected" ETH, even though the ETH has been returned to the user.

Impact

Users can manipulate the bonding curve to reach the target threshold without the contract actually collecting the expected amount of ETH, potentially leading to premature finalization and affecting the liquidity for the Uniswap pool creation.

Recommendation

Decrement `totalETHCollected` in the `sellTokens()` function by the amount of ETH returned to the user.

Status

Resolved

[M-02] BondingCurve#finalizeCurve - Unhandled leftover ETH and tokens after liquidity provision

Description

After providing liquidity to Uniswap V3, leftover ETH and tokens can remain stuck in the `PositionManager` and `BondingCurve` contracts due to slippage during the minting process, with no mechanism to handle or recover these funds.

Vulnerability Details

The `finalizeCurve()` function calls `UniswapPoolCreator.createLPPosition()` which mints a Uniswap V3 position using `INonfungiblePositionManager.mint{value: params.ethAmount}()`.

In `UniswapPoolCreator`, the minting process uses a 10% slippage tolerance, this means up to 10% of the provided ETH and tokens may not be used for liquidity provision and will remain in the contracts:

1. Excess ETH remains in the Uniswap PositionManager contract. The Uniswap V3 position manager provides a `refundETH()` function to recover this ETH, but it's never called.

2. Any tokens that were not sold during the bonding phase (the difference between the remaining amount and the used for liquidity `tokenReserve - tokenLiquidity`) and not provided as liquidity due to slippage (the difference between the provided amount `tokenAmount` and actual amount returned by `mint()`) remain locked in the `BondingCurve` contract with no recovery mechanism.

Impact

Leftover funds become permanently locked in the contracts

Recommendation

For the assets that were intended to be used for liquidity provision, implement a comprehensive leftover handling mechanism:

1. Calculate the leftover ETH and tokens using the returned values from `mint()` `amount0` and `amount1` currently not used.

2. If there is an ETH difference, after position creation, call `INonfungiblePositionManager(settings.positionManager).refundETH()` to recover any leftover ETH from the PositionManager. If there is a difference in tokens, the tokens will be available in the BondingCurve contract.

3. The team should decide what to do with the leftover tokens, options could be: transfer ETH to owner, let tokens be stuck in the contract or transfer them to a burn address, or other options.

For the tokens that were not intended to be used for liquidity provision, `tokenReserve - tokenLiquidity`, they can be left stuck in the contract as the current implementation is, or explicitly transferred to some burn address.

Status

Resolved

[M-03] UniswapPoolCreator#createLPPosition - Hardcoded tick spacing causes `finalizeCurve` revert for certain fee tiers

Description

The `UniswapPoolCreator` library uses a hardcoded tick spacing of `60`, which is incompatible with certain Uniswap V3 fee tiers, causing the `INonfungiblePositionManager.mint()` function to revert and preventing the `finalizeCurve()` transaction from completing.

Vulnerability Details

The hardcoded `int24 private constant TICK_SPACING = 60` tick spacing is used to calculate the minimum and maximum ticks.

However, Uniswap V3 associates specific tick spacings with different fee tiers. For example, a fee tier of 1% (10000 basis points) requires a tick spacing of 200, while a fee tier of 0.3% (3000 basis points) requires a tick spacing of 60.

When the hardcoded tick spacing of 60 is used with a fee tier that requires a different tick spacing (like 200 for 1% fee), the resulting tick values are not evenly divisible by the required tick spacing, causing the `INonfungiblePositionManager.mint()` function to revert.

Impact

The `finalizeCurve()` function will revert, preventing the curve finalization.

Recommendation

Modify the `createLPPosition()` function to dynamically retrieve the correct tick spacing based on the fee tier by calling `IUniswapV3Factory(params.factory).feeAmountTickSpacing(params.fee)` instead of using the hardcoded value.

Status

Resolved

[M-04] UniswapPoolCreator#createAndInitializePool - `sqrtPriceX96` calculation does not account for token ordering

Description

In `createAndInitializePool()`, the `sqrtPriceX96` ignores Uniswap V3's token ordering rule (token0 < token1), causing wrong price initialization and LP creation failure.

Vulnerability Details

In the `createAndInitializePool()` function, the `sqrtPriceX96` is calculated as:

```
uint160 sqrtPriceX96 = uint160(

    Math.sqrt((params.ethReserve * Q96) / params.tokenReserve) *

       Math.sqrt(Q96) );
```

This calculation assumes that the `token` is always `token0` and `WETH` is always `token1`. However, Uniswap V3 enforces that `token0 < token1` based on address comparison. When `params.token > params.weth`, the token ordering is inverted, but the price calculation doesn't account for this inversion which can cause the subsequent liquidity addition to fail due to slippage protection. The `createLPPosition()` function correctly handles token ordering by checking `bool tokenIs0 = params.token < params.weth` and adjusting the amounts accordingly.

Impact

When the `token` address is greater than the `WETH` address, the pool is initialized with an incorrect price, causing the `createLPPosition()` function to revert due to slippage protection when attempting to add liquidity.

Recommendation

Modify the `sqrtPriceX96` calculation to account for token ordering by checking if `params.token < params.weth` and inverting the price calculation when necessary.

Status

Resolved

Low

[L-01] BondingCurve#constructor - Missing `_disableInitializers()` call

Description

The `BondingCurve` contract is missing a constructor that calls `_disableInitializers()`, which is a security best practice for upgradeable contracts to prevent reinitialization attacks.

Vulnerability Details

The `BondingCurve` contract inherits from `Initializable` and is designed to be used behind a proxy. However, unlike other contracts in the codebase the `BondingCurve` contract does not have a constructor that calls `_disableInitializers()`.

The `_disableInitializers()` function is a security mechanism provided by OpenZeppelin's `Initializable` contract that prevents the implementation contract from being initialized or reinitialized.

Impact

The missing `_disableInitializers()` call creates a potential security vulnerability where the implementation contract could be initialized directly

Recommendation

Add a constructor to the `BondingCurve` contract that calls `_disableInitializers()`.

Status

Resolved

[L-02] BondingCurve, Factory, Foundry, Lock, TokenImplementation - Floating pragma

Description

All contracts use floating pragma statements `pragma solidity ^0.8.20` instead of fixed version pragmas.

Vulnerability Details

Floating pragmas allow contracts to be compiled with different compiler versions than tested, potentially introducing unidentified security issues.

Impact

Different pragma versions in test and production may pose unidentified security issues.

Recommendation

Consider locking the pragma version to the specific version `pragma solidity 0.8.20`.

Status

Resolved

[L-03] BondingCurve#currentPhase - No distinction between ready to finalize and finalized states

Description

The `Phase` enum lacks a proper state to distinguish between when the bonding curve is ready to be finalized and when it has actually been finalized, leading to state confusion.

Vulnerability Details

The `Phase` enum only contains three states: `PreBonding`, `Bonding`, and `Finalized`. However, the `currentPhase` is set to `Phase.Finalized` in two different scenarios:

1. In the `buyTokens()` function when `totalETHCollected >= settings.bondingTarget`

2. In the `finalizeCurve()` function after the Uniswap pool has been successfully created and the LP position is locked

This creates an inconsistency where the contract enters the "Finalized" state before the actual finalization process (creating the Uniswap pool, adding liquidity, and locking the LP position) is complete. The contract should have a separate "ReadyToFinalize" state to represent when the bonding target has been reached but finalization hasn't occurred yet.

Impact

The lack of proper state distinction can lead to confusion about the actual state of the bonding curve and may cause issues with external integrations that depend on the phase state.

Recommendation

Add a new `ReadyToFinalize` phase to the enum and update the logic accordingly:

1. Set `currentPhase = Phase.ReadyToFinalize` when the bonding target is reached in `buyTokens()`

2. Keep `currentPhase = Phase.Finalized` only after successful completion of the `finalizeCurve()` function

Status

Resolved

[L-04] Lock#constructor, Lock#lockNFT - Missing zero address validation

Description

The `Lock` contract constructor and `lockNFT` function lack zero address validation for critical parameters, which could lead to contract malfunction or loss of funds.

Vulnerability Details

The `Lock` contract has two instances where zero address validation is missing:

1. In the constructor the `_positionManager` parameter is directly assigned. If a zero address is passed, the contract would be permanently broken as all interactions with the Uniswap V3 Position Manager would fail.

2. In the `lockNFT` function the `owner` parameter is used without zero address validation. If a zero address is passed as the owner, the NFT would be locked but the owner would be set to address(0), making it impossible to unlock the NFT or claim fees, resulting in permanent loss of the locked NFT.

Impact

Missing zero address validation can result in permanent loss of funds and contract malfunction, as NFTs could be locked with invalid owner addresses or the contract could be initialized with invalid position manager addresses.

Recommendation

Add zero address validation checks in both locations.

Status

Resolved

[L-05] BondingCurve - Missing events for phase transitions

Description

The `BondingCurve` contract transitions between different phases but only emits an event for the final phase transition, missing events for PreBonding to Bonding and Bonding to Finalized transitions.

Vulnerability Details

The contract has three phases: `PreBonding`, `Bonding`, and `Finalized`. However, only the final transition emits an event `CurveFinalized`.

Impact

Missing phase transition events make it difficult for external systems to track the current state of the bonding curve and react to important state changes, reducing transparency and potentially causing integration issues.

Recommendation

Emit appropriate events for all phase transitions.

Status

Resolved

[L-06] Factory#initialize, Factory#updateBondingCurveSettings - Missing settings parameter validation

Description

The `Factory` contract's `initialize()` and `updateBondingCurveSettings()` functions lack comprehensive validation for the `BondingCurveSettings` parameters, which could lead to system malfunction and potential loss of funds.

Vulnerability Details

The `BondingCurveSettings` struct contains 10 parameters that are assigned without proper validation in both functions:

1. Address parameters `uniswapV3Factory`, `positionManager`, `weth`, `feeTo` are not validated for zero addresses

2. Numeric parameters lack minimum/maximum bounds validation:

- `virtualEth` (should be > 0)
- `preBondingTarget` (should be > 0 and < `bondingTarget`, although it is recalculated so no need to check)
- `bondingTarget` (should be > 0 and consider to have a reasonable maximum value to allow the bonding curve to be finalized)
- `minContribution` (should be > 0)
- `poolFee` (should be a valid Uniswap V3 fee tier)
- `sellFee` (should be reasonable percentage, e.g., < 1000 for 10%)

Impact

Missing parameter validation lacks best practices and may cause small complications, such as system configuration issues.

Recommendation

Add comprehensive validation for all `BondingCurveSettings` parameters.

Status

Resolved

[L-07] BondingCurve#sellTokens - `FeeTo` address can cause DoS of `sellTokens` operations

Description

The `sellTokens` function uses `sendValue` to transfer fees to the `feeTo` address, which can fail if the recipient is a contract that doesn't properly handle ETH transfers, causing a denial of service for all sell operations.

Vulnerability Details

In the `sellTokens` function, the contract transfers fees to the designated fee recipient. The `feeTo` address is set when initializing the bonding curve by the Factory contract and cannot be changed. On the factory contract, the owner can set the `feeTo` address without any validation that the address can properly receive ETH.

The `sendValue` function will revert if the recipient is a contract that doesn't properly handle ETH transfers.

When this happens, the entire `sellTokens()` transaction will revert, effectively denying service to all users trying to sell tokens.

Impact

The `sellTokens()` function can be permanently disabled if the `feeTo` address is set to an invalid address, preventing all token sales.

Recommendation

Consider implementing a pull based fee mechanism where fees are accumulated in the contract and the `feeTo` address can claim them later, or validate that the `feeTo` address can receive ETH before allowing the settings update. Alternatively, send fees to the Factory contract (which is known to handle ETH) and allow the fee recipient to claim from there.

Status

Acknowledged

[L-08] Factory, Foundry - Centralization risk

Description

Multiple critical functions are restricted to the contract owner, creating centralization risks where the owner has significant control over the protocol.

Vulnerability Details

The Factory and Foundry contracts contain several functions that are restricted to the owner only, including:

- `updateBondingCurveSettings()` - allows owner to modify bonding curve parameters used for new deployments

- `updateDeploymentFee()` - allows owner to change deployment fees

- `updateImplementation()` - allows owner to update contract implementations for new deployments

- `pause()` and `unpause()` - allows owner to pause/unpause the contract for new deployments

Impact

A compromised or malicious owner can cause denial of service, or manipulation of the protocol's core functionality, affecting future users of new deployments.

Recommendation

Consider using a multisig wallet for the address controlling the owner role. This would require multiple approvals for critical actions

Status

Acknowledged

[L-09] Factory#deployBondingCurveSystem - Missing pause mechanism

Description

The Factory contract lacks pause functionality for the `deployBondingCurveSystem()` function, unlike the `Foundry` contract which has pause controls for new deployments.

Vulnerability Details

The `Factory` contract's `deployBondingCurveSystem()` function does not include pause functionality, while the `Foundry` contract implements pause controls for its deployment functions. This creates an inconsistency in the pause mechanism across the protocol, where the `Factory` can continue deploying new bonding curve systems even during emergency situations. This is important especially considering that the `Factory` uses some parameters that were initialized from the `Foundry` contract like `tokenImplementation`, `bondingCurveImplementation`, and `lockAddress`.

Impact

The lack of pause functionality in Factory deployments could allow new bonding curve systems to be deployed during emergency situations.

Recommendation

Add pause functionality to the `Factory` contract and allow the `deployBondingCurveSystem()` function to be called when the contract is not paused.

Status

Resolved

[L-10] Factory#updateBondingCurveSettings - Incomplete event emission

Description

The `updateBondingCurveSettings()` function emits the `BondingCurveSettingsUpdated` event with only 5 out of 10 fields from the `BondingCurveSettings` struct, missing important configuration parameters.

Vulnerability Details

The `BondingCurveSettings` struct contains 10 fields however, the `BondingCurveSettingsUpdated` event only emits 5 of these fields, missing `sellFee`, `uniswapV3Factory`, `positionManager`, `weth`, and `feeTo`. This incomplete event emission makes it difficult for off-chain systems to track all configuration changes.

Also, when the settings are initialized in the `initialize()` function, the event is not emitted.

Impact

The incomplete event emission reduces transparency and makes it harder for external systems to monitor all bonding curve settings changes

Recommendation

Consider updating the `BondingCurveSettingsUpdated` event to include all fields from the `BondingCurveSettings`, or splitting the fields in multiple events to ensure complete transparency of configuration changes in `initialize()` and `updateBondingCurveSettings()`.

Status

Resolved

[L-11] Lock#checkAvailableFees - Incorrect calculation of fees

Description

The `checkAvailableFees` function incorrectly calculates accumulated fees by directly adding `feeGrowthInside0LastX128` and `feeGrowthInside1LastX128` values to the owed tokens, which does not represent the actual fee calculation logic used by Uniswap V3.

Vulnerability Details

In the `checkAvailableFees` function, the contract attempts to calculate available fees by adding the `feeGrowthInside0LastX128` and `feeGrowthInside1LastX128` values directly to the `tokensOwed0` and `tokensOwed1` amounts.

This calculation is incorrect because the `feeGrowthInside0LastX128` and `feeGrowthInside1LastX128` values represent the last recorded fee growth values for the position, not the current accumulated fees.

The proper fee calculation requires computing the difference between current pool fee growth values and the position's last recorded values, then multiplying by the position's liquidity.

Impact

The incorrect fee calculation will return inaccurate fee amounts, potentially misleading users about the actual fees available for claiming.

Recommendation

Since the `checkAvailableFees()` function is not used in the contracts and is most likely to be consumed by external integrations, consider implementing the fee calculation off-chain using the Uniswap SDK, which handles the complex fee growth calculations correctly. Alternatively, implement the proper on-chain fee calculation logic following how it is calculated in the `collect()` function.

Status

Resolved

Gas

[G-01] TokenImplementation#_update - Unnecessary transfer validation

Description

The `_update` function in `TokenImplementation` contains an unnecessary validation that checks if tokens have been minted before allowing transfers, which wastes gas on every transfer operation.

Vulnerability Details

The `_update` function includes a validation check `if (!_minted && from != address(0))` that prevents transfers when tokens haven't been minted.

However, this validation is redundant because all tokens are minted at once to a single address during the `mintTotalSupply()` function call. If no tokens have been minted, there would be no tokens to transfer anyway.

Impact

This unnecessary validation wastes gas on every transfer operation, increasing transaction costs for users without providing any security benefit.

Recommendation

Remove the `_update()` override function since it provides no value and only increases gas costs.

Status

Resolved

[G-02] BondingCurve,Factory,Lock Use `ReentrancyGuardTransientUpgradeable` for more gas efficient `nonReentrant` modifiers

Description

The contracts use the standard `ReentrancyGuardUpgradeable` implementation instead of `ReentrancyGuardTransient`, using it would provide significant gas savings by eliminating storage operations.

Vulnerability Details

The BondingCurve, Factory, and Lock contracts use the standard `ReentrancyGuardUpgradeable` implementation. If the target chains supports EIP-1153 (transient storage), using `ReentrancyGuardTransient` would provide significant gas savings by eliminating storage operations.

Impact

Gas savings on functions using the `nonReentrant` modifier.

Recommendation

If the target deployment chains supports EIP-1153 transient storage, consider using `ReentrancyGuardTransient` instead of `ReentrancyGuard`.

Status

Acknowledged

QA

[Q-01] BondingCurve - Unused state variable

Description

The `accumulatedFees` state variable is declared but never used throughout the contract lifecycle.

Vulnerability Details

The `accumulatedFees` state variable is declared on the `BondingCurve` contract but is never read from or written to in any of the contract's functions.

Impact

This unused state variable adds unnecessary code that should be removed for cleaner and more maintainable code.

Recommendation

Remove the unused `accumulatedFees` state variable declaration

Status

Resolved

[Q-02] BondingCurve - Unused Math library import

Description

The OpenZeppelin `Math` library is imported but never used throughout the contract.

Vulnerability Details

The `@openzeppelin/contracts/utils/math/Math.sol` library is imported on the `BondingCurve` contract but is never referenced or used in any of the contract's functions.

Impact

This unused import adds unnecessary code that should be removed for cleaner and more maintainable code.

Recommendation

Remove the unused import

Status

Resolved

[Q-03] Factory, UniswapPoolCreator, BondingMath - Magic numbers instead of constants

Description

Magic numbers are used directly in calculations instead of being defined as named constants, reducing code readability and maintainability.

Vulnerability Details

The codebase uses literal magic numbers in percentage calculations instead of defining them as constants.

- In `Factory` the values `20` and `100` are used to calculate 20% of virtualEth for preBondingTarget calculations.

- In `UniswapPoolCreator` the values `90` and `100` are used to calculate 90% for slippage tolerance in minimum amount calculations.

- In `BondingMath` the value `10000` is used to calculate 100% for sell fee calculations.

Impact

This practice reduces code readability and maintainability by making it unclear what these numbers represent and harder to update if the percentages need to change.

Recommendation

This practice reduces code readability and maintainability by making it unclear what these numbers represent and harder to update if the percentages need to change.

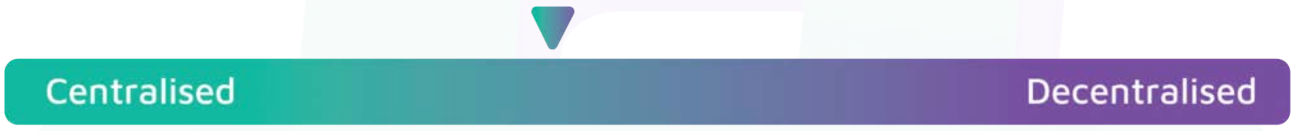
Status

Resolved

Centralisation

The Blockstreet project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.



Conclusion

After Hashlock's analysis, the Blockstreet project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved or acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:
In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:
Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:
We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:
We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au