Audited by CRED SHIELDS

CredShields

Smart Contract Audit

June 19th, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Manadotwin between June 13th, 2025, and June 17th, 2025. A retest was performed on June 19th, 2025.

Author

Shashank (Co-founder, CredShields) shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead), Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor), Yash Shah (Auditor), Prasad Kuri (Auditor)

Prepared for

Manadotwin (mana.win)

Table of Contents

1. Executive Summary ----------

Manadotwin engaged CredShields to perform a smart contract audit from June 13th, 2025, to June 17th, 2025. During this timeframe, Four (4) vulnerabilities were identified. **A retest was performed on June 19th, 2025, and all the bugs have been addressed.**

During the audit, Three (3) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Manadotwin" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Smart Contract	2	1	2	0	0	0	**5**
	2	**1**	**2**	**0**	**0**	**0**	**5**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in the Smart Contract's scope during the testing window while abiding by the policies set forth by Manadotwin's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Manadotwin's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Manadotwin can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Manadotwin can future-proof its security posture and protect its assets.

2. The Methodology ----------

Manadotwin engaged CredShields to perform a Smart Contract Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation Phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from June 13th, 2025, to June 17th, 2025, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

IN SCOPE ASSETS
https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting Phase

Manadotwin is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	None	Low	Medium
		LOW	MEDIUM	HIGH
Likelihood				

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. **Low**

   Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. **Medium**

   Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. **High**

   High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. **Critical**

   Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- Shashank, Co-founder CredShields shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

3. Findings Summary ----------

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, Four (4) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
WETH Reserve is not balanced	Critical	Logical Error
Bonding Curve Fee Calculation Error	Critical	Logical Error
Excess Tokens Are Without a Withdrawal Mechanism	High	Missing Functionality
Remaining Tokens Are Not Burned After Graduation	Medium	Missing Functionality
Owner Can Drain-All Funds After Graduation	Medium	Centralization Risk

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0
SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use

SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found
SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.

SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status ----------

Manadotwin is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on June 19th, 2025, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
WETH Reserve is not balanced	Critical	**Fixed** [June 19th, 2025]
Bonding Curve Fee Calculation Error	Critical	**Fixed** [June 18th, 2025]
Excess Tokens Are Without a Withdrawal Mechanism	High	**Fixed** [June 19th, 2025]
Remaining Tokens Are Not Burned After Graduation	Medium	**Fixed** [June 18th, 2025]
Owner Can Drain-All Funds After Graduation	Medium	**Fixed** [June 18th, 2025]

Table: Summary of findings and status of remediation

5. Bug Reports ----------

Bug ID #1 [Fixed]

WETH Reserve is not balanced

Vulnerability Type
Logical error

Severity
Critical

Description
The _handleReserveManipulation function attempts to mitigate manipulation of Uniswap reserves by comparing current pair reserves and adjusting token amounts accordingly before calling addLiquidityETH(). However, the logic only considers token imbalance and ignores any (WETH) imbalance in the pair.
If ETH reserves have been manipulated (e.g., via a 1 wei deposit and sync()), the function blindly sends tokens to match the skewed ETH reserve. This can result in incorrect token:ETH ratios, wasting token supply or causing liquidity addition failure.
Similarly, if malicious tokens deposited in a pair contract this logic ignores any WETH imbalance in the pair.

Affected Code

- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L464

Impacts
The flawed logic in _handleReserveManipulation can lead to an incorrect token-ETH ratio in the liquidity pool, distorting price discovery and affecting fair trading. By overestimating the required token amount, the function may unnecessarily drain the contract's token reserves, reducing tokens available for other intended uses. If the contract doesn't hold enough tokens to cover the transfer, the _graduate() function may revert, disrupting the graduation process.

Additionally, the lack of ETH-side reserve checks allows attackers to manipulate ETH balances in the pair contract (e.g., by depositing Token and calling sync()), leading to an imbalanced pool and

potential denial-of-service scenarios. This could block the contract from adding liquidity and transitioning to the post-bonding phase.

Remediation

The function should first call skim() on the pair to remove any tokens or ETH accidentally or maliciously deposited into the contract. This will clean up the pool and restore intended balances. After skimming, it should then check both ETH and token reserves for discrepancies. Only if imbalance persists should extra tokens and ETH be sent—and only after ensuring the contract holds enough tokens and ETH

Additionally, the logic for determining how much to transfer should be corrected to ensure it maintains the intended lpReserve:actualReserve ratio. Using Uniswap's syncing functions is recommended for more precise reserve handling.

Retest

This issue has been fixed by first calling skim() to remove manipulated token, then calculating the imbalance, topping up the correct amount, and calling sync() to restore accurate reserves.

Bug ID #2 [Fixed]

Bonding Curve Fee Calculation Error

Vulnerability Type
Critical

Severity
Logical Error

Description
The vulnerability exists in three critical functions within the BondingCurve contract: buyTokens(), factoryDevBuy(), and previewBuyTokens(). The current implementation incorrectly calculates the required ETH amount and fee structure when users attempt to purchase tokens that would exceed the remaining capacity needed to reach the graduation target of 5 ETH.

In the current flawed implementation, when a user sends more ETH than needed to reach the target, the contract calculates the fee based only on the remaining ETH needed (remainingEthNeeded), rather than accounting for the total ETH required including fees. This creates a logical inconsistency where the fee calculation does not properly account for the circular dependency between the net ETH needed and the fee itself.

The specific problematic logic occurs in the conditional branch where the input amount exceeds the remaining needed amount. In buyTokens() and factoryDevBuy(), the current code sets feeEth = (remainingEthNeeded * BPS_FEE) / BPS_MAX and then calculates ethToUse = remainingEthNeeded + feeEth, followed by netEthToUse = remainingEthNeeded. This approach fails to properly solve for the total ETH amount needed when fees are considered.

Similarly, in previewBuyTokens(), the function suffers from the same mathematical error in the else branch where it sets netEthToUse = remainingEthNeeded after calculating fees, rather than properly accounting for the fee structure. This means the preview function provides inaccurate token output estimates that don't match the actual execution logic, creating a discrepancy between expected and actual results for users.

Affected Code
- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L156
- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L247

- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L581

Impacts
Users can bypass fees while giving less fees and due to that users don't have to pay fees and fees can be bypassed. Users can create a case where ethToUse will be greater than msg.value which will result in users to bypass fees and result in incorrect calculation.

Remediation
The fix requires recalculating the remaining ETH needed to properly account for fees in the total amount calculation across all three affected functions. The corrected approach should first determine the net reserve amount still needed to reach the target, then calculate the total ETH required (including fees) to provide that net amount.

The remediation involves replacing the current calculation logic with a more mathematically sound approach. Instead of calculating `remainingEthNeeded` as simply `TARGET_RAISE - tokenData.actualReserve`, the contract should calculate the total ETH needed including fees using the formula: `remainingEthNeeded = (remainingReserveNeeded * BPS_MAX) / (BPS_MAX - BPS_FEE)`, where `remainingReserveNeeded` represents the net amount still needed for reserves.

Retest
This issue has been fixed by updating the fee calculation logic.

Bug ID #3 [Fixed]

Excess Tokens Are Without a Withdrawal Mechanism

Vulnerability Type
Missing Functionality

Severity
High

Description
The contract uses uniswapPair.skim(address(this)) in _handleReserveManipulation to pull excess tokens or WETH into its own address, reducing the risk of manipulated reserves. However, the contract lacks any mechanism to retrieve these funds afterward. Without a withdrawal function gated by graduation status, any skimmed assets remain permanently locked inside the contract.

Affected Code
- https://github.com/fedzdev/fomo/blob/2bc2918d8007421c902b61b8c78ba3a02371cf90/contracts/BondingCurve.sol#L478

Impacts
Over time, tokens or ETH collected via skim accumulate in the contract and become inaccessible, leading to capital inefficiency or unintended value loss for the project.

Remediation
Implement a withdrawSkimmedAssets function callable only by the owner after graduation to recover stranded funds.

Retest
This issue has been fixed by adding a withdrawal withdrawSkimmedAssets function.

Bug ID #4 [Fixed]

Remaining Tokens Are Not Burned After Graduation

Vulnerability Type
Missing Functionality

Severity
Medium

Description
After the bonding curve contract "graduates" (i.e., reaches its TARGET_RAISE and adds liquidity to Uniswap), it transitions from a bonding curve mechanism to a decentralized liquidity model.
However, any leftover unsold tokens in availableSupply remain in the contract and are not burned or otherwise handled.
These tokens may:

- Be misused by the contract owner (despite Ownable) or,
- Introduce supply inflation,
- Undermine trust in the supply dynamics.

Since LP tokens are sent to the burn address and liquidity becomes immutable, any remaining tokens should also be burned to preserve scarcity and ensure alignment with bonding economics.

Affected Code

- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L378

Impacts
After graduation, the contract retains control over unsold tokens, creating a token supply centralization risk. This contradicts the expectations of a decentralized and trustless token distribution. Users generally assume that any undistributed tokens will be burned, and failure to do so raises trust and transparency concerns. Economically, these retained tokens could dilute the circulating supply if used later, potentially impacting token value and undermining user confidence.

Remediation
In _graduate(), burn the remaining tokenData.availableSupply tokens after adding liquidity, like so:

```
function _graduate() internal {
    ...
```

```
    // Prepare and add liquidity
    _addLiquidity(pair);

    +     uint256 leftover = tokenData.availableSupply;
    +     if (leftover > 0) {
    +       tokenData.availableSupply = 0;
    +       token.safeTransfer(DEAD_ADDRESS, leftover);
    +     }
}
```

Retest

This bug has been fixed by introducing the function _burnRemainingTokens(). Fixed commit.

Bug ID #5 [Fixed]

Owner Can Drain-All Funds After Graduation

Vulnerability Type
Centralization Risk

Severity
Medium

Description
The emergencyWithdrawBalance function permits the contract owner() to unilaterally withdraw the entire ETH balance to the treasuryAddress once tokenData.graduated is true. There are no additional access controls, time locks, or multi-signature requirements. This design introduces a single point of trust that contradicts decentralization principles.

Affected Code
- https://github.com/fedzdev/fomo/blob/7d2993bec7e58665ca0a0bbc3c8fed2b71208a89/contracts/BondingCurve.sol#L531-L543

Impacts
Once the contract is marked as graduated, the owner can immediately extract all user-deposited funds, potentially without community oversight or warning.

Remediation
Restrict emergency withdrawals using a multi-signature check or governance approval to distribute authority and reduce centralization risk.

Retest
This issue has been fixed by removing the emergencyWithdrawBalance function.

6. The Disclosure ----------

The Reports provided by CredShields are not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.

YOUR SECURE FUTURE STARTS HERE

CRED SHIELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHIELDS

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com