Audited by

CredShields

Smart Contract Audit

April 1st, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Fomodotbiz between March 19th, 2025, and March 25th, 2025. A retest was performed on March 31st, 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)

Prepared for
Fomodotbiz

Table of Contents

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

Fomodotbiz engaged CredShields to perform a smart contract audit from March 19th, 2025, to March 25th, 2025. During this timeframe, 19 vulnerabilities were identified. **A retest was performed on March 31st, 2025, and all the bugs have been addressed.**

During the audit, 3 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Fomodotbiz" 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
Fomo Contracts	3	0	3	7	0	6	**19**
	3	**0**	**3**	**7**	**0**	**6**	**19**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Fomo Contract's scope during the testing window while abiding by the policies set forth by Fomodotbiz'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 Fomodotbiz'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 Fomodotbiz can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Fomodotbiz can future-proof its security posture and protect its assets.

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

Fomodotbiz engaged CredShields to perform a Fomo 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 March 19th, 2025, to March 25th, 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/tree/c425141a39e933290560bf93e6fc49965f1e1631

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

Fomodotbiz is actively partnering with CredShields to validate the remediations implemented toward 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, 19 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Incorrect fee calculation can result in excessive fee deduction from users	Critical	Logical Error
A malicious can front-run and cause the bonding curve graduation to fail	Critical	Front-running
AddLiquidityETH DoS by directly depositing WETH	Critical	Denial of Service (DoS)
Missing zero minTokensOut validation can lead to undesired token purchases	Medium	Input Validation Issue
Factory contract refund misroute causes devAddress to lose funds	Medium	Improper Funds Transfer
Reorg attack	Medium	Blockchain Reorg
Unauthorized link injection in user params	Low	Insufficient Input Validation
Use Ownable2Step	Low	Missing Best Practices

Missing zero address validations	Low	Missing Input Validation
Missing events in important functions	Low	Missing Best Practices
Floating and outdated pragma	Low	Floating Pragma
Dead Code	Low	Code With No Effects
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing best practices
Cheaper conditional operators	Gas	Gas Optimization
Unused Imports	Gas	Gas Optimization
Public constants can be private	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas Optimization
Custom error to save gas	Gas	Gas Optimization
Cheaper Inequalities in if()	Gas	Gas Optimization

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	Vulnerable	Bug ID #11
SWC-103	Floating Pragma	Vulnerable	Bug ID #11
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 ------------------

Fomodotbiz is actively partnering with CredShields from this engagement to validate the remediation of the discovered vulnerabilities. **A retest was performed on March 31st, 2025, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Incorrect fee calculation can result in excessive fee deduction from users	Critical	**Fixed** [March 31, 2025]
A malicious can front-run and cause the bonding curve graduation to fail	Critical	**Fixed** [March 31, 2025]
AddLiquidityETH DoS by directly depositing WETH	Critical	**Fixed** [March 31, 2025]
Missing zero minTokensOut validation can lead to undesired token purchases	Medium	**Fixed** [March 31, 2025]
Factory contract refund misroute causes devAddress to lose funds	Medium	**Fixed** [March 31, 2025]
Reorg attack	Medium	**Fixed** [March 31, 2025]
Unauthorized link injection in user params	Low	**Won't Fix** [March 31, 2025]
Use Ownable2Step	Low	**Won't Fix** [March 31, 2025]
Missing zero address validations	Low	**Fixed** [March 31, 2025]
Missing events in important functions	Low	**Won't Fix** [March 31, 2025]
Floating and outdated pragma	Low	**Fixed** [March 31, 2025]

Dead Code	Low	**Fixed** [March 31, 2025]
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed** [March 31, 2025]
Cheaper conditional operators	Gas	**Won't Fix** [March 31, 2025]
Unused Imports	Gas	**Won't Fix** [March 31, 2025]
Public constants can be private	Gas	**Won't Fix** [March 31, 2025]
Gas Optimization in Increments	Gas	**Won't Fix** [March 31, 2025]
Custom error to save gas	Gas	**Fixed** [March 31, 2025]
Cheaper Inequalities in if()	Gas	**Partially Fixed** [March 31, 2025]

Table: Summary of findings and status of remediation

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

Bug ID #1 [Fixed]

Incorrect fee calculation can result in excessive fee deduction from users

Vulnerability Type
Logical Error

Severity
Critical

Description
The bonding curve contract implements a fee structure where a percentage of the ETH input during a token purchase or sale is deducted as a fee. However, there is a critical issue in the fee calculation logic that leads to users being charged an incorrect amount. Specifically, the function buyTokens (and similarly factoryDevBuy) uses the total msg.value (the full ETH sent by the user) to calculate the fee, even when the actual ETH required to reach the graduation target is less than msg.value. In such cases, the contract unnecessarily charges a fee based on the entire msg.value, rather than only applying the fee to the remaining ETH needed for the graduation process.
This error occurs because the contract calculates the fee on the full amount of ETH received from the user (msg.value), not considering that only the remaining ETH to meet the TARGET_RAISE should be subject to the fee. The correct behavior should be to calculate the fee only on the portion of the ETH that will actually be used in the bonding curve calculation, which is capped by the remainingEthNeeded value.

Affected Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L189

Impacts
The direct consequence of this vulnerability is that users could end up paying higher fees than they should when making purchases on the bonding curve, especially when they send more ETH than required to meet the target raise amount.

Remediation

To mitigate this issue, the fee calculation logic should be adjusted to apply the fee only to the remaining ETH required to meet the TARGET_RAISE, rather than the full msg.value. This ensures that the contract accurately deducts fees based only on the ETH that will be used for the bonding curve mechanism and not on any excess ETH sent by the user.

Retest

This issue has been fixed.

Bug ID #2 [Fixed]

A malicious can front-run and cause the bonding curve graduation to fail

Vulnerability Type
Front-running

Severity
Critical

Description
The contract has a critical vulnerability that allows an attacker to exploit the bonding curve graduation process by front-running the transaction that triggers the creation of the Uniswap pair. Specifically, the vulnerability arises in the _graduate function, where the contract creates a Uniswap pair using the uniswapFactory.createPair function and subsequently adds liquidity to it. However, if an attacker manages to front-run this transaction, they can create the pair before the contract's intended transaction, causing the creation of the pair to fail when the contract attempts to call createPair. This is because Uniswap's factory contract will not allow the creation of duplicate pairs between the same assets (in this case, the token and WETH). As a result, the contract will revert, and the graduation process cannot be completed.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L332

Impacts
The direct consequence of this vulnerability is that the contract cannot graduate, which is the intended final state of the bonding curve process. Without graduation, the contract cannot provide liquidity to the Uniswap pool, and the bonding curve functionality may become permanently stuck.

Remediation
A recommended solution would be to implement a safeguard that verifies whether the pair already exists before attempting to create it. If the pair already exists, the contract should skip the pair creation step and proceed to the liquidity addition process. This can be achieved by adding a check to ensure that the createPair function is only called if the pair does not already exist in the Uniswap factory.

The following modification can be made to the _graduate function:

```
address pair = uniswapFactory.getPair(address(token), uniswapRouter.WETH());
```

```
if (pair == address(0)) {
   pair = uniswapFactory.createPair(address(token), uniswapRouter.WETH());
}
```

Retest

This issue has been fixed.

Bug ID #3 [Fixed]

AddLiquidityETH DoS by directly depositing WETH

Vulnerability Type
Denial of Service (DoS)

Severity
Critical

Description
The vulnerability arises in the contract due to the interaction with Uniswap's liquidity addition mechanism in the _graduate() function. Specifically, the function uses Uniswap's addLiquidityETH() method to add liquidity to the Uniswap pair. However, this method is vulnerable to manipulation by malicious users who can directly deposit 1 wei of WETH into the pair contract. By calling the sync() function on the pair, they can disrupt the internal balance of the reserves.
When the malicious user deposits WETH (or any ERC-20 token) directly into the pair, it will update the reserves of the Uniswap pair contract. Since the liquidity is added using both ETH and tokens, the liquidity pool is based on the balances within the pair contract. The pair contract is designed to sync the reserves when liquidity changes, and a malicious deposit followed by a sync() call can result in an invalid or insufficient liquidity state, triggering a revert with the error UniswapV2Library: INSUFFICIENT_LIQUIDITY. This would cause the addLiquidityETH function to fail, potentially disrupting the contract's graduation process and preventing the addition of liquidity.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L361

Impacts
If an attacker successfully exploits this vulnerability, they can prevent the contract from adding liquidity to Uniswap and completing its graduation process. The graduation process is essential as it enables the contract to move from the bonding curve phase to the Uniswap listing phase, where liquidity is crucial for the functioning of the contract.

Remediation
Before adding liquidity, the _graduate() function should call skim() and check if the reserve is manipulated. If it is, mint an equivalent amount of token directly to the pool and call sync()

and then perform addLiquidityETH().
And make sure the minOut for Meme Token is not too strict as it may result in failure, so consider setting it lower.

Retest
This issue has been fixed.

Bug ID #4 [Fixed]

Missing zero minTokensOut validation can lead to undesired token purchases

Vulnerability Type
Input Validation Issue

Severity
Medium

Description
In the buyTokens and sellTokens functions, the user provides a minTokensOut value to protect against significant slippage. However, there is no explicit validation to check if minTokensOut is set to zero. If this parameter is set to zero, the contract allows token purchases or sales without validating that a meaningful amount of tokens or ETH is actually transferred. The absence of this validation can allow for invalid or unintended trades to occur, where the user effectively pays ETH or sells tokens without receiving or getting the expected amount in return.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L209
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L279

Impacts
The impact of this vulnerability is significant, as it may result in users inadvertently executing token purchases or sales where no tokens are received or transferred. This can lead to a loss of funds, particularly if users unknowingly specify a zero value for minTokensOut, allowing the contract to accept trades with no meaningful token transfer.

Remediation
To resolve this issue, it is essential to add a validation check in both the buyTokens and sellTokens functions to ensure that the minTokensOut parameter is greater than zero.

Retest
This issue has been fixed.

Bug ID #5 [Fixed]

Factory contract refund misroute causes devAddress to lose funds

Vulnerability Type
Improper Funds Transfer

Severity
Medium

Description`
The factoryDevBuy() function is designed to handle token purchases by the factory owner while deducting a fee. However, the excess ETH refund mechanism mistakenly sends the refund back to msg.sender, which in this case is the factory contract itself, instead of the intended developer address (devAddress). As a result, the developer's rightful refund gets trapped within the factory contract, effectively causing a direct financial loss.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L468

Impacts
The vulnerability directly impacts the developer's financial interests by preventing the proper return of excess ETH during token purchases.

Remediation
It is recommended to modify the refund logic to explicitly use the predefined devAddress instead of msg.sender when processing refunds.

Retest
This issue has been fixed.

Bug ID #6 [Fixed]

Reorg attack

Vulnerability Type
Blockchain Reorg

Severity
Medium

Description
The _deployContracts() function in the contract is responsible for deploying new Bonding Curve using the new opcode, which internally uses the create opcode for contract creation. This mechanism is vulnerable to a **reorg attack**, a type of attack that exploits blockchain reorganization events. During a reorg, the blockchain network can temporarily reorganize its blocks, replacing old blocks with new ones that are consistent with network consensus.

In the event of a reorg, an attacker can exploit this mechanism to create a contract with the same address to which another user has already transferred funds. This is particularly relevant for Ethereum-based blockchains, which experience occasional reorgs. Optimistic rollups such as Optimism and Arbitrum are also prone to reorgs, especially when fraud proofs are discovered, leading to reverted blocks.

Attack Scenario:

1. Alice calls _deployContracts() and deploys a Bonding Curve contract.
2. Before the transaction is finalized, she transfers funds to the token contract.
3. A reorg occurs, reverting Alice's contract creation.
4. Bob, seeing the intended vault address, quickly calls _deployContracts(), getting the same address in the reorged chain.
5. When Alice's fund transfer is reprocessed, it goes to Bob's Bonding Curve instead.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L174

Impacts
Users may unintentionally buy tokens from the contracts controlled by malicious users due to reorg-based manipulation which will lead to loss of funds.

Remediation

It is recommended to use create2 to ensure deterministic contract creation. The create2 opcode generates the contract address based on the deployer's address, a salt, and the bytecode. Including msg.sender as part of the salt ensures that an attacker cannot easily predict or duplicate contract addresses.

Retest

This issue has been fixed.

Bug ID #7 [Won't Fix]

Unauthorized link injection in user params

Vulnerability Type
Insufficient Input Validation

Severity
Low

Description
The parameters params.socials, params.image, and params.description accept user-controlled input and without proper validation. This allows attackers to embed malicious links, misleading images, or deceptive descriptions in user profiles, potentially leading to phishing attacks or other fraudulent activities via frontend

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L210-L248

Impacts
In frontend, users may be tricked into clicking malicious links, leading to credential theft, malware infections, or exposure to harmful content. This can compromise user security, damage platform reputation, and facilitate social engineering attacks.

Remediation
Enforce strict input validation and sanitization for user-controlled fields, allowing only properly formatted URLs and images.

Retest
Client's Comments: We decided that since we already do input validation on frontend and backend for the images and links (and only send IPFS hashes to Smart contract) its not necessary to do more input validation that the name, symbol and description.

Bug ID #8 [Won't Fix]

Use Ownable2Step

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Affected Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L11
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L10
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/MemeToken.sol#L7

Impacts
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Remediation
It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

Retest:

These issues have no significant cause to the contract and Credshield's team agrees, hence marked as won't fix.

Bug ID #9 [Fixed]

Missing zero address validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contracts were found to be setting new addresses without proper validations for zero addresses.
Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.
Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L69-L77
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/MemeToken.sol#L18

Impacts
If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation
Add a zero address validation to all the functions where addresses are being set.

Retest
This issue has been fixed.

Bug ID #10 [Won't Fix]

Missing events in important functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L85-L108
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L306-L312

Impacts
Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation
Consider emitting events for important functions to keep track of them.

Retest
Client's Comments: We already emit events quite comprehensively wherever required.

Bug ID #11 [Fixed]

Floating and outdated pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contract allowed floating or unlocked pragma to be used, i.e., ^0.8.20. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L2
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L2
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/MemeToken.sol#L2

Impacts
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.
Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is low.

Remediation
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.28 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

This issue has been fixed by setting pragma version to 0.8.28

Bug ID #12 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects - [SWC-135]

Severity
Low

Description
The manualGraduation() function includes an owner check that prevents the factory contract from calling the function, effectively rendering the manual graduation mechanism inaccessible. While the function includes appropriate checks for graduation conditions like insufficient funds and already graduated status, the owner restriction creates a functional deadlock where no entity can trigger manual graduation.

Affected Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L387

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement. This reduces the overall size of the contracts and also helps in saving gas.

Remediation
It is recommended to review the owner validation logic to ensure the factory contract can execute manual graduation when necessary.

Retest
This issue has been fixed by removing manualGraduation()

Bug ID #13 [Fixed]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom

Vulnerability Type
Missing best practices

Severity
Low

Description
The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

Affected Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L219
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L289
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L327
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L456
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L189

Impacts
Using safeTransferFrom has the following benefits -
- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.
- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Remediation
Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest

This issue has been fixed by using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom()

Gas ID #14 [Won't Fix]

Cheaper conditional operators

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Upon reviewing the code, it has been observed that the contract uses conditional statements involving comparisons with unsigned integer variables. Specifically, the contract employs the conditional operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L229
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L297
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L326
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L460
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L105
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L160
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L199

Impacts
Employing x != 0 in conditional statements can result in reduced gas consumption compared to using x > 0. This optimization contributes to cost-effectiveness in contract interactions.

Remediation
Whenever possible, use the x != 0 conditional operator instead of x > 0 for unsigned integer variables in conditional statements.

Retest

Client’s Comments: As Taraxa is almost feeless we dont consider the gas optimizations to be necessary, so we can omit those.

Gas ID #15 [Won't Fix]

Unused Imports

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract FomoFactory.sol was importing contracts MemeToken.sol which was not used anywhere in the code. This increases the gas cost and overall contract's complexity.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L7

Impacts
Unused imports in smart contracts can lead to an increase in the size of the code, making it more difficult to verify and potentially slowing down its execution. Moreover, having unused code in a smart contract can also increase the attack surface by potentially introducing vulnerabilities that can be exploited by malicious actors. This can lead to security issues and compromise the integrity of the contract.

Additionally, including unused imports in smart contracts can also increase deployment and gas costs, making it more expensive to deploy and run the contract on the Ethereum network.

Remediation
It is recommended to remove the import statement if the external contracts or libraries are not used anywhere in the contract.

Retest:
Client's Comments: As Taraxa is almost feeless we dont consider the gas optimizations to be necessary, so we can omit those.

Gas ID #16 [Won't Fix]

Public constants can be private

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Affected Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L33
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L33

Impacts
Public constants are more costly due to the default getter functions created for them, increasing the overall gas cost.

Remediation
If reading the values for the constants is not necessary, consider changing the public visibility to private.

Retest
Client's Comments: As Taraxa is almost feeless we dont consider the gas optimizations to be necessary, so we can omit those.

Gas ID#17 [Won't Fix]

Gas Optimization in Increments

Vulnerability Type
Gas optimization

Severity
Gas

Description
The contract uses two for loops**,** which use post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Vulnerable Code
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L133
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L143
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L296

Impacts
Using **i++** instead of **++i** costs the contract deployment around 600 more gas units.

Remediation
It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest
Client's Comments: As Taraxa is almost feeless we dont consider the gas optimizations to be necessary, so we can omit those.

Gas ID #18 [Fixed]

Custom error to save gas

Vulnerability Type
Gas Optimization

Severity
Gas

Description
During code analysis, it was observed that the smart contract is using the revert() statements for error handling. However, since Solidity version 0.8.4, custom errors have been introduced, providing a better alternative to the traditional revert(). Custom errors allow developers to pass dynamic data along with the revert, making error handling more informative and efficient. Furthermore, using custom errors can result in lower gas costs compared to the revert() statements.

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L263

Impacts
Custom errors allow developers to provide more descriptive error messages with dynamic data. This provides better insights into the cause of the error, making it easier for users and developers to understand and address issues.

Remediation
It is recommended to replace all the instances of revert() statements with error() to save gas..

Retest
This issue has been fixed.

Gas ID #19 [Partially Fixed]

Cheaper Inequalities in if( )

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

Description
The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L129-L129
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L197-L197
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L433-L433
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L209-L209
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L445-L445
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L223-L223
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L466-L466
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L229-L229
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L297-L297
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L460-L460
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L279-L279

- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L282-L282
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L326-L326
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L347-L347
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/BondingCurve.sol#L390-L390
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L105-L105
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L117-L117
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L120-L120
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L124-L124
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L128-L128
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L160-L160
- https://github.com/fedzdev/fomo/blob/c425141a39e933290560bf93e6fc49965f1e1631/contracts/FomoFactory.sol#L199-L199

Impacts
Using strict inequalities inside “if” statements costs more gas.

Remediation
It is recommended to go through the code logic and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest:
This issue has been partially fixed. Some of the affected links are left to be updated.
Client’s Comments: As Taraxa is almost feeless we dont consider the gas optimizations to be necessary, so we can omit those.

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