Cyberscope

Audit Report

Rocket Protocol

Sep 2024

Network	ETH
Address	0x264d4aa232fE795f0B7bbF41963137a3D49F85a8
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	FairLaunchLimitBlockTokenV3
Compiler Version	v0.8.24+commit.e11b9ed9
Optimization	20000 runs
Explorer	https://etherscan.io/address/0x264d4aa232fe795f0b7bbf41963137a3d49f85a8
Address	0x264d4aa232fe795f0b7bbf41963137a3d49f85a8
Network	ETH
Symbol	ETEST
Decimals	18
Total Supply	20,000,000,000

Audit Updates

Initial Audit	12 Jun 2024 https://github.com/cyberscope-io/audits/blob/main/phei/v1/audit.pdf
Corrected Phase 2	27 Aug 2024 https://github.com/cyberscope-io/audits/blob/main/phei/v2/audit.pdf
Corrected Phase 3	06 Sep 2024

Source Files

Filename	SHA256
NoDelegateCall.sol	df71f011abaff271622bb1695fcf0d93cb34d91bfff304b558cd6494ae187190
Meme.sol	540e7b643f661b30afa5552b0cbf8196f07358c49468aa34542155e7e6cad74f
IMeme.sol	fc794154cb742af3cb7397005f188f6e978d66d7e321e725681514480d9574ee
IFairLaunch.sol	8a4f83221d50264b74526f30249a396c3190e92bb9e8599d8cddc3b9e94c4fc9
FairLaunchLimitBlockV3.sol	0ebb435e260d519e160d9381dfa6251e1f4af4984077f817aa6ecad7193ed918

Overview

The `FairLaunchLimitBlockV3` contract is designed to facilitate a fair token launch process, incorporating various functionalities for refunding, funding, minting, and starting token trading. It integrates mechanisms to handle user funds, ensure fairness through block-based conditions, and leverage Uniswap for liquidity provision.

The primary purpose of the `FairLaunchLimitBlockV3` contract is to manage the launch of a the `Meme` token with specific constraints and commands. It allows users to fund the contract, request refunds, mint tokens, and initiate trading, all governed by predefined block numbers and funding caps to ensure a fair and orderly process.

Before Start Functionality

Funding and Refunding

If the `canStart` condition is `false`, indicating that the designated block number has not yet been reached, users can interact with the contract in several ways. They can send native tokens (ETH) to the contract in specific amounts defined as commands. If users send the `REFUND_COMMAND` value (0.0002), the contract will process a `refund` of their previously contributed funds. Alternatively, if users send any other value, the contract will treat it as a funding contribution, adding the sent amount to their balance and the contract's total fund balance.

Starting Trading and Refunding

If the `canStart` condition is `true`, meaning the designated block number has been reached but trading has not yet started, users can send the `START_COMMAND` value (0.0005) to initiate the token trading process. This action will trigger the contract to add liquidity to Uniswap and set the started state to true. Users can also still request refunds, before the `START_COMMAND` happens, by sending the `REFUND_COMMAND` value, which will process their refund as described earlier.

After Start Functionality

Minting Tokens

Once the contract has started (i.e., trading is enabled), users can send the `MINT_COMMAND` value (0.0001) to mint tokens. The contract verifies that the sender has not already minted tokens and calculates the amount of tokens they are eligible to receive based on their contribution relative to the total funds. The tokens are then transferred to the user, and the origianl `MINT_COMMAND` amount of ETH is returned to the user.

Claiming Extra ETH

After the contract has started, if the total ETH collected exceeds a predefined `softTopCap` , users can claim their share of the excess ETH by sending the `CLAIM_COMMAND` value (0.0002). The contract calculates each user's claimable amount based on their contribution proportion and transfers the corresponding ETH to them. This ensures that any excess funds are fairly distributed among the participants.

Command Implementation

The contract uses specific ETH values as commands to trigger different functionalities:

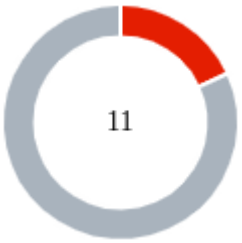
- REFUND_COMMAND (0.0002): Triggers the refund process for users who wish to withdraw their contributions before the start.
- CLAIM_COMMAND (0.0002): Allows users to claim extra funds after the start if the total collected exceeds the soft top cap.
- START_COMMAND (0.0005): Initiates the start of trading and adds liquidity to Uniswap.
- MINT_COMMAND (0.0001): Enables users to mint tokens once trading has started.

Liquidity Pool Initialization

The `_start` function initializes a Uniswap V3 liquidity pool, setting the stage for trading. It first verifies that the contract hasn’t started and holds enough tokens, then checks if a pool already exists. If not, it calculates the ETH amount to be added, approves token management, and creates the pool via `_initPool` . The function then mints and locks the liquidity tokens to prevent manipulation, marks the contract as started, and emits a

`LaunchEvent` to confirm the successful creation of the liquidity pool. If a locker is specified, the liquidity tokens are securely locked in the contract.

Findings Breakdown



Critical	2
Medium	0
Minor / Informative	9

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	0	0	0	0
Minor / Informative	9	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IPI	Incorrect Pool Initialization	Unresolved
●	PFT	Premature Functionality Triggering	Unresolved
●	ILC	Ineffective Logic Checks	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	RRC	Redundant Require Checks	Unresolved
●	UPS	Unrestricted PoolFee Set	Unresolved
●	UBL	Unused BNB Lock	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

IPI - Incorrect Pool Initialization

Criticality	Critical
Location	FairLaunchLimitBlockV3.sol#L187,441
Status	Unresolved

Description

The contract is designed to create and initialize a Uniswap V3 pool during the constructor phase by setting the initial price using specific amounts (`amount0` and `amount1`). However, since the pool price is determined and fixed during this constructor phase, it cannot be changed afterward. When the `_initPool` function is called during the start functionality, it attempts to calculate a new price based on updated amounts. However, this newly calculated price is not applied because the pool has already been initialized with the initial price during the constructor. This results in a scenario where the intended price, reflecting the total funds accumulated during the presale, is not set, causing a mismatch between the project's financial objectives and the actual market price of the tokens in the pool.

```
constructor(
        address _locker,
        uint24 _poolFee,
        address _projectOwner,
        FairLaunchLimitBlockStruct memory params
    ) Meme(params.name, params.symbol, params.meta) {
        ...
        weth =
INonfungiblePositionManager(uniswapPositionManager).WETH9();

        (address token0, address token1) = address(this) < weth
            ? (address(this), weth)
            : (weth, address(this));

        (uint256 amount0, uint256 amount1) = address(this) < weth
            ? (totalDispatch / 2, softTopCap)
            : (softTopCap, totalDispatch / 2);

        uint160 sqrtPriceX96 = getSqrtPriceX96(amount0, amount1);
        INonfungiblePositionManager(uniswapPositionManager)
            .createAndInitializePoolIfNecessary(
                token0,
                token1,
                poolFee,
                sqrtPriceX96
            );
    }
```

```
    function _initPool(
        uint256 totalAdd,
        INonfungiblePositionManager _positionManager
    )
        private
        returns (
            address token0,
            address token1,
            uint256 amount0,
            uint256 amount1,
            address pool
        )
    {
        (token0, token1) = address(this) < weth
            ? (address(this), weth)
            : (weth, address(this));

        (amount0, amount1) = address(this) < weth
            ? (totalDispatch / 2, totalAdd)
            : (totalAdd, totalDispatch / 2);

        uint160 sqrtPriceX96 = getSqrtPriceX96(amount0, amount1);
        pool = _positionManager.createAndInitializePoolIfNecessary(
            token0,
            token1,
            poolFee,
            sqrtPriceX96
        );
    }
```

Recommendation

It is recommended that, when the start functionality is invoked, the contract should calculate the total amount of funds accumulated and then create and initialize the pool based on these amounts to ensure that the initial price accurately reflects the funds raised as intended by the contract's functionality. The contract should implement safeguards to prevent malicious actors from creating a pool position before the intended time. This could be achieved by deploying the token address that will be used in the pool during the start transaction, ensuring that only the contract has control over the pool's initialization and pricing set up.

PFT - Premature Functionality Triggering

Criticality	Critical
Location	FairLaunchLimitBlockV3.sol#L230,264
Status	Unresolved

Description

The contract's current configuration allows for unintended triggering of critical functionalities due to the premature return of `true` by the `canStart` function. This creates two potential issues. Firstly, it prevents any additional contributions from users since the current block number exceeds `untilBlockNumber`, halting fundraising efforts prematurely and limiting the project's ability to achieve its financial objectives. Secondly, it enables any user to send the `START_COMMAND` to invoke the `_start` function, which mints a position and sets the `started` variable to `true`, thereby preventing any future addition of liquidity to the Uniswap V3 pool. These combined issues result in suboptimal liquidity conditions, limiting both fundraising capacity and the pool's flexibility and growth potential.

```
if (started) {
    // after started
    if (msg.value == MINT_COMMAND) {
        // mint token
        _mintToken();
    } else if (msg.value == CLAIM_COMMAND) {
        _claimExtraETH();
    } else {
        revert("FairMint: invalid command - mint or claim only");
    }
} else {
    // before started
    if (canStart()) {
        if (msg.value == REFUND_COMMAND) {
            // before start, you can always refund
            _refund();
        } else if (msg.value == START_COMMAND) {
            // start trading, add liquidity to uniswap
            _start();
        } else {
            revert("FairMint: invalid command - start or refund only");
        }
    } else {
        if (msg.value == REFUND_COMMAND) {
            // before start, you can always refund
            _refund();
        } else {
            // before start, any other value will be considered as fund
            _fund();
        }
    }
```

```
   function canStart() public view returns (bool) {
        // return block.number >= untilBlockNumber || totalEthers >=
softTopCap;
        // eth balance of this contract is more than zero
        return block.number >= untilBlockNumber;
    }
```

Recommendation

It is recommended to carefully evaluate and align the conditions under which the `canStart` function returns `true` with the intended fundraising timeline and objectives.

Additionally, implement stricter controls and timing mechanisms for invoking the `_start` function to prevent premature execution. This can be achieved by incorporating additional checks, such as time-based restrictions or access controls, ensuring that the functions are only executed at appropriate times and under the correct circumstances, thereby preserving the contract's intended functionality and financial goals.

ILC - Ineffective Logic Checks

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L357,333
Status	Unresolved

Description

The contract is implementing checks that do not provide any meaningful logic validation since, based on the contract's setup and variable assignments, they will always evaluate to true. These checks do not contribute to enhancing the security or functionality of the contract, leading to unnecessary complexity and potential inefficiencies.

```
require(_mintAmount > 0, "FairMint: mint amount is zero");
assert(_mintAmount <= totalDispatch / 2);
...
uint256 fee = (amount * refundFeeRate) / 10000;
assert(fee < amount);
```

Recommendation

It is recommended to consider the removal of these redundant checks to simplify the contract. This will improve the contract's readability and maintainability without compromising its security or intended functionality.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	NoDelegateCall.sol#L19
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(address(this) == original)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

RRC - Redundant Require Checks

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L284,349,350
Status	Unresolved

Description

The contract is performing additional checks that have already been validated earlier in the code. This redundancy can lead to inefficiencies and unnecessary complexity within the contract, potentially increasing gas costs and making the contract harder to maintain.

```
require(started, "FairMint: withdraw extra eth must after start");
...
require(started, "FairMint: not started");
...
require(msg.sender == tx.origin, "FairMint: can not mint to contract.");
```

Recommendation

It is recommended to consider the removal of redundant checks to streamline the code. This will enhance the efficiency of the contract by reducing unnecessary validations, lowering gas costs, and improving code clarity and maintainability.

UPS - Unrestricted PoolFee Set

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L207
Status	Unresolved

Description

The contract contains a mechanism that allows the factory owner to set any fee value as `poolFee` through the `setPoolFee` function. This function enables the factory owner to assign a new fee without any constraints or validation checks on the maximum allowable value. Consequently, the owner may take advantage of it by calling the `setPoolFee` function with a high percentage value.

```
    function setPoolFee(uint24 _poolFee) public {
        IFactory _factory = IFactory(factory);
        require(msg.sender == _factory.owner(), "FairMint: only
owner");
        poolFee = _poolFee;
    }
```

Recommendation

It is recommended to implement safeguards to limit the maximum fee that can be set through the `setPoolFee` function. The contract could include a check for the maximum acceptable fee value to prevent abuse. Additionally, the team should ensure the secure management of the private keys associated with the owner's account. A robust security mechanism should be implemented to prevent a single point of failure or unauthorized access to the contract's administrative functions.

Temporary Solutions: These measures, while not reducing the finding's severity, can provide interim protection:

- Introduce a time-lock mechanism with a reasonable delay to allow for the community or stakeholders to review changes before they take effect.

- Deploy a multi-signature wallet that requires multiple parties to approve any fee changes.
- Establish a governance model where users vote on proposed fee changes, ensuring community consensus.

Permanent Solution:

- Consider renouncing ownership of the `factory` contract to permanently eliminate the threat of a single entity having unrestricted control over critical functions. Note that this is a non-reversible action and should be carefully weighed against future operational flexibility.

UBL - Unused BNB Lock

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L413
Status	Unresolved

Description

The contract is invoking the `refundETH()` function after minting liquidity, which successfully returns any excess BNB that was not used in the liquidity provision process back to the contract's balance. However, the contract lacks any mechanisms to either utilize this refunded BNB within its existing functionalities or to withdraw it. As a result, the refunded BNB remains locked within the contract indefinitely, potentially leading to inefficiencies or unintended behavior in terms of resource allocation.

```
(
    uint256 _tokenId,
    uint128 _liquidity,
    uint256 _amount0,
    uint256 _amount1
) = _positionManager.mint{value: totalAdd}(params);
_positionManager.refundETH();
```

Recommendation

It is recommended to consider the intended functionality regarding the refunded BNB. The contract should implement a mechanism to properly handle the refunded BNB, either by integrating it into the contract's operational logic (e.g., adding it to available funds for other uses) or providing a secure method to withdraw or redistribute the funds. This will ensure that the contract’s resources are managed efficiently and prevent unnecessary locking of BNB.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L40,208,271
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function WETH9() external pure returns (address);
uint24 _poolFee
address _addr
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L211
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
poolFee = _poolFee
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	FairLaunchLimitBlockV3.sol#L181,182
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
locker = _locker
projectOwner = _projectOwner
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	NoDelegateCall.sol#L2 Meme.sol#L2 IMeme.sol#L2 IFairLaunch.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.19;
```

Recommendation

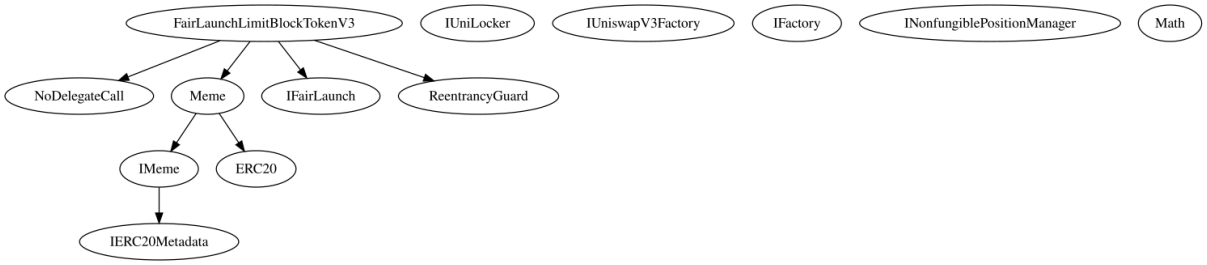
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

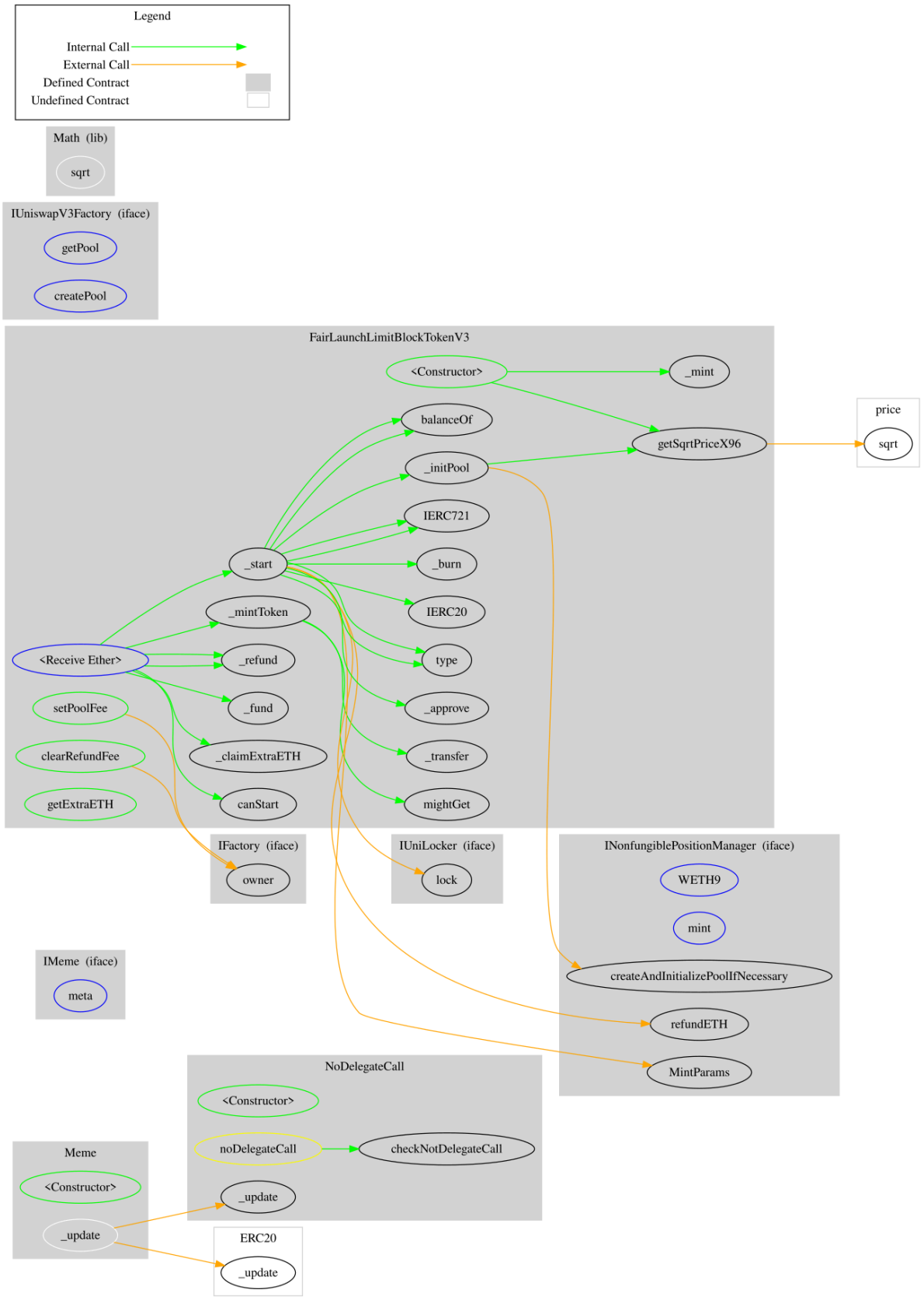
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
NoDelegateCall	Implementation			
		Public	✓	-
	checkNotDelegateCall	Private		
Meme	Implementation	IMeme, ERC20		
		Public	✓	ERC20
	_update	Internal	✓	
IMeme	Interface	IERC20Metadata		
	meta	External		-
IFairLaunch	Interface			
FairLaunchLimitBlockTokenV3	Implementation	IFairLaunch, Meme, ReentrancyGuard, NoDelegateCall		
		Public	✓	Meme

	setPoolFee	Public	✓	-
	clearRefundFee	Public	✓	-
		External	Payable	noDelegateCall
	canStart	Public		-
	getExtraETH	Public		-
	_claimExtraETH	Private	✓	nonReentrant
	mightGet	Public		-
	_fund	Private	✓	nonReentrant
	_refund	Private	✓	nonReentrant
	_mintToken	Private	✓	nonReentrant
	_start	Private	✓	nonReentrant
	_initPool	Private	✓	
	_update	Internal	✓	
	getSqrtPriceX96	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Math (lib)
sqrt
IUniswapV3Factory (iface)
getPool
createPool
FairLaunchLimitBlockTokenV3
<Constructor>
_mint
balanceOf
getSqrtPriceX96
price
sqrt
_initPool
IERC721
_start
_burn
IERC20
_mintToken
type
<Receive Ether>
_refund
_approve
setPoolFee
_fund
_transfer
clearRefundFee
_claimExtraETH
mightGet
getExtraETH
canStart
IFactory (iface)
owner
IUniLocker (iface)
lock
INonfungiblePositionManager (iface)
WETH9
mint
IMeme (iface)
meta
createAndInitializePoolIfNecessary
refundETH
MintParams
NoDelegateCall
<Constructor>
noDelegateCall
checkNotDelegateCall
Meme
<Constructor>
_update
_update
ERC20
_update

Summary

The FairLaunchLimitBlockV3 contract is a comprehensive solution for launching the MEME token contract with built-in functionalities for handling funds, initiating trading, and ensuring fairness through block-based conditions and funding caps, including a refund functionality. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io