Coinsult

Request your audit at **coinsult.net**

Advanced Manual

Smart Contract Audit

December 6, 2025

x.com/coinsultaudits

t.me/coinsult_tg

Audit requested by

CheesePad

-

Global Overview

Manual Code Review

In this audit report we will highlight the following issues:

Vulnerability Level	Total	Pending	Acknowledged	Resolved
Informational	0	0	0	0
Low-Risk	5	0	2	3
Medium-Risk	2	0	1	1
Critical-Risk	1	0	1	0

Risk Classification

Coinsult uses certain vulnerability levels, these indicate how bad a certain issue is. The higher the risk, the more strictly it is recommended to correct the error before using the contract.

Vulnerability Level	Description
Informational	Does not compromise the functionality of the contract in any way
Low-Risk	Won't cause any problems, but can be adjusted for improvement
Medium-Risk	Will likely cause problems and it is recommended to adjust
Critical-Risk	Will definitely cause problems, this needs to be adjusted

Audit Summary

Project	
Website	https://www.cheesepad.ai
Blockchain	-
Source Code	-
Contract Address	-

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

Audit Scope

Coinsult was commissioned to perform an audit based on the provided code.

Note that we only audited the code available to us on this URL at the time of the audit. If the URL is not from any block explorer (main net), it may be subject to change. Always check the contract address on this audit report and compare it to the token you are doing research for.

Audit Method

Coinsult's manual smart contract audit is an extensive methodical examination and analysis of the smart contract's code that is used to interact with the blockchain. This process is conducted to discover errors, issues and security vulnerabilities in the code in order to suggest improvements and ways to fix them.

Automated Vulnerability Check

Coinsult uses software that checks for common vulnerability issues within smart contracts. We use automated tools that scan the contract for security vulnerabilities such as integer-overflow, integer-underflow, out-of-gas-situations, unchecked transfers, etc.

Manual Code Review

Coinsult's manual code review involves a human looking at source code, line by line, to find vulnerabilities. Manual code review helps to clarify the context of coding decisions. Automated tools are faster but they cannot take the developer's intentions and general business logic into consideration.

Used tools

- Slither: Solidity static analysis framework
- Remix: IDE Developer Tool
- CWE: Common Weakness Enumeration
- SWC: Smart Contract Weakness Classification and Test Cases
- DEX: Testnet Blockchains

Table of Contents

Launchpad Audit Considerations

This security audit report has been prepared by Coinsult's team to identify potential vulnerabilities and security issues within the provided smart contract code. While we have conducted a thorough examination using both manual review and automated tools, this audit does not guarantee the complete absence of vulnerabilities or security flaws.

Important Considerations:

Human Limitations: Security audits are performed by humans and, as such, are subject to human error. Despite our best efforts and expertise, it is possible that some vulnerabilities, edge cases, or attack vectors were not identified during this audit.

No Absolute Security: No audit can provide absolute assurance of security. New vulnerabilities may be discovered over time, and attack vectors may evolve as the blockchain ecosystem develops.

Not Financial Advice: This report is purely technical in nature and should not be construed as financial, investment, or legal advice. Any decisions to deploy, invest in, or interact with the audited smart contracts should be made after consulting with appropriate financial and legal advisors.

Code Changes: This audit is valid only for the specific version of the code reviewed at the time of the audit. Any modifications, additions, or updates to the codebase after the audit date may introduce new vulnerabilities that are not covered by this report.

Third-Party Dependencies: This audit covers the provided smart contract code. Any third-party contracts, libraries, or external dependencies that were not explicitly included in the audit scope may contain their own vulnerabilities.

Scope Limitation - Standard Components: This audit focuses on custom business logic and security-critical implementations specific to the Cheesepad platform. Standard components including interface definitions, the EIP-2535 Diamond pattern implementation, external battle-tested libraries (such as Uniswap's FullMath), and mock contracts used for testing were not subjected to line-by-line auditing. These components represent industry-standard, extensively audited, and production-proven code that has been deployed across hundreds of projects and secured billions of dollars in value.

User Responsibility: Users and developers should conduct their own due diligence and consider obtaining multiple independent audits before deploying smart contracts to mainnet or committing significant funds.

By using this audit report, you acknowledge and agree that Coinsult and its auditors shall not be held liable for any losses, damages, or issues that may arise from the deployment or use of the audited smart contracts.

Hardcoded emergency withdrawal fee

Low-Risk *- Won't cause any problems, but can be adjusted for improvement*

File	Lines
EmergencyWithdrawFacet.sol	59

Error Code	Description
HARD-WITH	Hardcoded 10% for emergency withdrawal fee. Makes code less maintainable and harder to understand.

Code Snippet

```
uint256 fee = (contribution * 10) / 100;
```

Recommendation

Might be intentional to avoid centralization risks... But if not, make it a parameter that can be changed within limits.

Status	Note by Cheesepad
Acknowledged	This is by design. The fee is always 10% of the withdrawal amount, making a separate parameter unnecessary.

Missing Input Validation

Low-Risk *- Won't cause any problems, but can be adjusted for improvement*

File	Lines
CreatePresaleFacet.sol	119-129

Error Code	Description
INP-VAL	Doesn't validate that tgePercent + cyclePercent doesn't exceed 100%, which could lead to over-distribution of tokens.

Code Snippet

```
if (
    params.tgePercent != 0 ||
    params.cyclePercent != 0 ||
    params.cycle != 0
) {
    require(
        params.tgePercent > 0 && params.tgePercent <= 100,
        "Invalid tge percent"
    );
```

Recommendation

Add check that checks if 100% together.

Status	Note by Cheesepad
Resolved	Additional validation checks have been implemented to enhance code safety.

Inconsistent Error Messages

Low-Risk *- Won't cause any problems, but can be adjusted for improvement*

File	Lines
CreatePresaleFacet.sol	124-126

Error Code	Description
INC-ERR	Error message says "Invalid tge percent" but the check is for cyclePercent. This makes debugging harder.

Code Snippet

```
require(
    params.cyclePercent > 0 && params.cyclePercent <= 100,
    "Invalid tge percent" // Wrong error message
);
```

Recommendation

"Invalid cycle percent"

Status	Note by Cheesepad
Resolved	Error messages have been standardized for consistency.

Unbounded Array Growth

Low-Risk *- Won't cause any problems, but can be adjusted for improvement*

File	Lines
AffiliateFacet.sol	71-74

Error Code	Description
UNB-ARR	The buyerToReferrers array can grow unbounded if a buyer uses many referrers. This could cause gas issues when iterating.

Code Snippet

```
if (!poolInfo.buyerToReferrersExists[buyer][referrer]) {
    poolInfo.buyerToReferrers[buyer].push(referrer);
    poolInfo.buyerToReferrersExists[buyer][referrer] = true;
}
```

Recommendation

Add maximum referrer limit per buyer.

Status	Note by Cheesepad
Acknowledged	This scenario is highly unlikely in practice. Introducing an arbitrary limit would be suboptimal.

TODO Comment in Production Code

Low-Risk *- Won't cause any problems, but can be adjusted for improvement*

File	Lines
CreateFairLaunchFacet	58

Error Code	Description
TODO	Production code contains TODO comment indicating incomplete feature. Service fees are not being handled. This could be intentional for later upgrade or forgotten functionality.

Code Snippet

```
// TODO Handle service fees
```

Recommendation

Either implement the feature or remove the comment if it's not needed.

Status	Note by Cheesepad
Resolved	The TODO comment has been removed and the feature fully implemented.

Hardcoded Fee Parameters

Medium-Risk *- Will likely cause problems and it is recommended to adjust*

File	Lines
ConfigFacet.sol	34-36

Error Code	Description
HARD-FEES	Critical fee parameters are hardcoded in setMasterConfig. This prevents granular control and makes the system inflexible. The owner cannot adjust fees without calling the entire function.

Code Snippet

```
config.nativeFeePercent = 5;
config.maxAffiliateBps = 1000;
```

Recommendation

Make these configurable parameters with separate setter functions and proper bounds checking.

Status	Note by Cheesepad
Acknowledged	These parameters are unlikely to change and are appropriately fixed.

Unsafe External Call Without Gas Limit

Medium-Risk *- Will likely cause problems and it is recommended to adjust*

File	Lines
AffiliateFacet.sol	358-368

Error Code	Description
UNSAFE-CALL	Using .transfer() forwards only 2300 gas, which can fail with smart contract recipients that need more gas. This breaks composability and may cause legitimate claims to fail.

Code Snippet

```
function _transferRewards(
    address currency,
    address to,
    uint256 amount
) internal {
    if (currency == address(0)) {
        payable(to).transfer(amount);
    }
```

Recommendation

Use .call() with proper reentrancy protection.

Status	Note by Cheesepad
Resolved	Changed transfer to call. The caller function is already protected by the nonReentrant modifier, preventing reentrancy.

Single Owner Control

Critical-Risk *- Will definitely cause problems, this needs to be adjusted*

File	Lines
ConfigFacet.sol	19-131

Error Code	Description
SNG-OWN-CTRL	The owner has unrestricted control over critical parameters including fee destinations, fee percentages, and affiliate rates. This creates a single point of failure and trust assumption. The owner can change fee addresses to drain funds or manipulate fee structures at will.

Code Snippet

```
function setMasterConfig(
 address feeTo,
 address locker,
 address affiliateProgram,
 address verificationWallet,
 uint256 creationFee,
 uint256 fairLaunchMaxDuration
) external onlyOwner {
 config.nativeFeePercent = 5; // Hardcoded
 config.maxAffiliateBps = 1000; // Can be changed later
```

Recommendation

Implement a multi-signature wallet or timelock mechanism for critical configuration changes. Add events and transparency mechanisms. Consider implementing parameter bounds and immutable critical addresses.

Status	Note by Cheesepad
Acknowledged	Multisig wallets will be implemented in the future.

Signature Replay Attack

Critical-Risk *- Will definitely cause problems, this needs to be adjusted*

File	Lines
ConfigFacet.sol	154-183

Error Code	Description
SIGN-REPL	The signature verification doesn't include a nonce, timestamp, or chain ID. This allows signatures to be replayed multiple times or across different chains. An attacker could reuse a valid signature to make unauthorized contributions.

Code Snippet

```
function _verifyContributionSignature(
    address fundManager,
    address user,
    uint256 amount,
    bytes memory signature
) internal view {
    bytes32 messageHash = keccak256(
        abi.encodePacked(fundManager, user, amount)
    );
```

Recommendation

Add nonce tracking per user, include chain ID in signature, and add expiration timestamps.

Status	Note by Cheesepad
Acknowledged	Signatures include the user address and verification requires msg.sender to match, preventing third-party reuse. Fund manager contract addresses are chain-specific due to different deployment sequences, making cross-chain replay impossible. Signatures can only be reused for identical (fundManager, user, amount) parameters and are constrained by pool contribution limits. We don't strictly prevent replay in this case.

No Slippage Protection on Liquidity Addition

Critical-Risk *- Will definitely cause problems, this needs to be adjusted*

File	Lines
LibLaunchpad.sol	100-186

Error Code	Description
NO-SLIP	While minToken and minCurrency are calculated, they're set equal to the full amounts in most cases (lines 112-113). This provides NO slippage protection. If the pair already exists with different reserves, the calculated minToken and minCurrency might not account for actual required amounts, causing transaction to fail or accept unfavorable rates. Users' funds could be locked with poor liquidity ratios.

Code Snippet

```
if (currency == address(0)) {
    (, , liquidity) = router.addLiquidityETH{value: listingCurrency}(
        token,
        listingToken,
        minToken,
        minCurrency,
        address(this),
        block.timestamp
    );
}
```

Recommendation

Implement proper slippage tolerance (e.g., 0.5-1%)

Status	Note by Cheesepad
Acknowledged	We already warn pool owners not to distribute tokens before the pool is finalized. If an attacker sends a small amount of WETH to the pair and calls sync, we have logic in place to correct the reserves before adding liquidity. If that correction fails, we also have a separate tool available to resolve the issue. We do not plan to introduce slippage, as we want the listing rate to match the calculated value exactly. Therefore, we will leave the current implementation unchanged.

Disclaimer

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

Coinsult is not responsible if a project turns out to be a scam, rug-pull or honeypot. We only provide a detailed analysis for your own research.

Coinsult is not responsible for any financial losses. Nothing in this contract audit is financial advice, please do your own research.

The information provided in this audit is for informational purposes only and should not be considered investment advice. Coinsult does not endorse, recommend, support or suggest to invest in any project.

Coinsult can not be held responsible for when a project turns out to be a rug-pull, honeypot or scam.