Coinsult

Request your audit at **coinsult.net**

Advanced Manual

Smart Contract Audit

December 5, 2025

x.com/coinsultaudits

t.me/coinsult_tg

Audit requested for

SalamPresale

0x83F2cfc254F0C281DAC60A1f6D191788aE2225C8

Global Overview

Manual Code Review

In this audit report we will highlight the following issues:

Vulnerability Level	Total	Pending	Acknowledged	Resolved
🔵 Informational	2	0	0	2
🟢 Low-Risk	2	0	0	2
🟠 Medium-Risk	4	0	1	3
🔴 Critical-Risk	2	0	1	1

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://salam.capital/
Blockchain	BASE
Source Code	https://basescan.org/address/0x83F2cfc254F0C281DAC60A1f6D191788aE2225C8#code
Contract Address	0x83F2cfc254F0C281DAC60A1f6D191788aE2225C8

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

● Hardcap check is off by one

Informational *- Does not compromise the functionality of the contract in any way*

Error Code	Description
HARDCAP	When AmountRaised + _amount == totalHardCap, the tx reverts. The maximum effective raise is < totalHardCap (one unit less).

Code Snippet

```
if (salamPool.AmountRaised + _amount >= salamPool.totalHardCap) {
  revert Presale__PresaleAmountRaisedMet(
    salamPool.totalHardCap,
    salamPool.AmountRaised + _amount
  );
}
```

Recommendation

Usually you want > to revert, not >=, so that you can exactly hit the cap.

✅ **Fixed**

● Min / max purchase amounts are never enforced

Informational *- Does not compromise the functionality of the contract in any way*

Error Code	Description
MIN-MAX	The contract stores _min and _max and even defines an error. But they are never checked anywhere.

Code Snippet

```
uint256 _min;
uint256 _max;
...
error Presale__PresaleMinMaxBuyCrossed(uint256 _amount);
```

Recommendation

Add checks or remove the constants.

✅ **Fixed**

● No Direct @openzeppelin Imports

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

Error Code	Description
DIR-OPEN	No direct @openzeppelin imports can result in modified openzeppelin libraries. **For this audit scope, it is assumed the openzeppelin imports are safe and 1:1 copies from openzeppelin.**

Code Snippet

```
import {Ownable} from
"../../lib/openzeppelin-contracts/contracts/access/Ownable.sol";
import {ReentrancyGuard} from
"../../lib/openzeppelin-contracts/contracts/utils/ReentrancyGuard.sol";
import {IERC20} from
"../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {SafeERC20} from
"../../lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol";
```

Recommendation

It is highly recommended to use @openzeppelin imports instead of local libraries.

✅ **Fixed**

● setStatus lets owner bypass state machine

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

Error Code	Description
STATUS-BYPASS	This allows: switching from Finalised back to Active, etc. Ignoring the invariants enforced by deposit(), finalisePresale(), cancelPresale(), etc. That may be intentional as a privileged "escape hatch", but it means: The state machine is only soft-enforced. Anyone trusting the presale to be in a clean lifecycle must also trust the owner not to abuse this.

Code Snippet

```
function setStatus(PresaleState _state) public onlyOwner {
    salamPool.state = _state;
}
```

Recommendation

Remove this function otherwise the states make no sense.

✅ Fixed

● Phase Start Time is Always 0

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

Error Code	Description
PHASE-ZERO	When adding a phase, its start timestamp is always set to 0. It does rely on: - owner calling setPresaleTimes() before deposit(), - and pushPhase() being called via buy() to advance phases. If _start is never configured, startTime stays 0 and the sale never works

Code Snippet

```
phases.push(Phase({
    tokenAmount: _tokenAmount,
    tokensSold: 0,
    pricePerToken: _pricePerToken,
    fundsRaised: 0,
    duration: _duration,
    startTime: 0,
    completed: false,
    withdrawn: false
}));
```

Recommendation

If intentional, it needs to be documented, else update it to have a timestamp.

✅ **Acknowledged, intentional.**

● Unsold tokens are stuck after presale

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

Error Code	Description
UNSOLD-STUCK	totalDeposit == tokens deposited at deposit(). tokenBalance is decremented on each buy. So you send only the sold tokens to the vesting contract. Any unsold tokens remain in the presale contract. There is **no function** to withdraw or recover those unsold tokens.

Code Snippet

```
uint256 totalPurchasedTokens = salamPool.totalDeposit - salamPool.tokenBalance;
if (totalPurchasedTokens > 0) {
   presaleToken.safeTransfer(vestingContract, totalPurchasedTokens);
}
salamPool.state = PresaleState.Finalised;
```

Recommendation

Unsold presale tokens are permanently stuck in the presale contract once finalized. Add a function to withdraw them.

✅ **Fixed**

● One transaction can oversell a phase

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

Error Code	Description
OVERSELL	Per-phase token caps (Phase.tokenAmount) are never enforced when selling, only after the sale. No check like require(currentPhase.tokensSold + tokensToSell <= tokenAmount). So one transaction can oversell a phase, buying more tokens at the old (cheaper) price just before the phase flips.

Code Snippet

```
uint256 usdAmount = _handleTokenSwap(_amount, _tokenId);
_computePresaleBuy(usdAmount, msg.sender);

uint256 salamTokenToSend = _calculateTokenAmount(usdAmount);

salamPool.AmountRaised += usdAmount;
salamPool.tokenBalance -= salamTokenToSend;
Phase storage currentPhase = phases[salamPool.currentPhaseIndex];
currentPhase.tokensSold += salamTokenToSend;
currentPhase.fundsRaised += usdAmount;
```

Recommendation

This breaks the economic guarantees of phased pricing. Its recommended to add the require statement.

✅ **Fixed**

● buy parameter _amount has inconsistent meaning

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

Error Code	Description
PARAM-INC	In buy(uint256 _amount, uint8 _tokenId): For ETH (_tokenId == 0), _amount is ignored. You pay with msg.value. For USDC (_tokenId == 1), _amount is the USDC amount transferred. For other tokens, _amount is the token amount swapped. This is confusing and makes the event E_PresaleBuy misleading:

Code Snippet

```
emit E_PresaleBuy(_amount, block.timestamp, salamPool.currentPhaseIndex,
_tokenId);
```

Recommendation

For ETH buys, _amount in the event is just whatever user passed (not the actual msg.value or resulting USD value).

✅ **Fixed**

● Loss of funds due to 0% slippage

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

Error Code	Description
LOSS-FUNDS	In both _swapETHForUSDC and _swapTokenForUSDC, the swap is performed with 0% slippage. Meaning the transaction can be frontrunner for 99%, resulting in a user paying $100 receiving 1$ worth of tokens.

Code Snippet

```
uint256[] memory amounts = uniswapV2Router02.swapExactETHForTokens{
    value: msg.value
}(0, path, address(this), block.timestamp + 15 minutes);

—

uint256[] memory amounts = uniswapV2Router02.swapExactTokensForTokens(
    _amount,
    0,
    path,
    address(this),
    block.timestamp + 15 minutes
);
```

Recommendation

Do NOT perform swaps inside the contract for user-based purchases, the user must NOT be the victim of sandwich attacks (frontruns). Store the currency the user pays, and manually convert it to your desired currency.

✅ **Fixed**

🔴 Cancelled presale has no refund path

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

Error Code	Description
NO-REFUND	It only returns unsold SALAM tokens (tokenBalance) to the owner. Users have paid funds (USDC/ETH/other tokens) but still have no SALAM tokens and no refund mechanism because refund() is commented out.

Code Snippet

```
salamPool.state = PresaleState.Canceled;

uint256 _amount = salamPool.tokenBalance;
if (_amount > 0) {
   salamPool.tokenBalance = 0;
   presaleToken.safeTransfer(msg.sender, _amount);
}
emit E_PresaleCanceled(_amount);
```

Recommendation

Add code for refunds of users.

✅ Acknowledged, by design

Notes

All imports and libraries are considered blackboxes (contracts that will NEVER revert or misbehave). Contact the project owners to also audit those files if needed.

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.