



# AUDIT REPORT

---

April 2025

For




**Guess.Meme**

# Table of Content

Executive Summary	04
Number of Security Issues per Severity	06
Checked Vulnerabilities	07
Techniques and Methods	08
Types of Severity	10
Types of Issues	11
 <b>High Severity Issues</b>	12
1. The create instruction is susceptible to unlimited minting attack and pump/dump scenario	12
2. Zero Token Withdrawal Risk: Complete Status Only Triggered After Last Token Sale	14
3. Possible lock of funds due to the tendency of complete status remaining active	15
4. Missing Token Account Owner Validation Check	16
 <b>Medium Severity Issues</b>	17
5. Unchecked SOL Balance in Sell Function: Risk of Failed Transfers Due to Insufficient Balance	17
6. Rent set aside for storage is never redeemed	18
7. max_allowed_amount will be bypassed by users	18



# Table of Content

 <b>Low Severity Issues</b>	20
8. No event emission for the withdraw operation	20
Functional Tests	21
Automated Tests	21
Closing Summary & Disclaimer	22



# Executive Summary

<b>Project Name</b>	Guessmeme
<b>Project URL</b>	<a href="https://guessmemelive.netlify.app/"><u>https://guessmemelive.netlify.app/</u></a>
<b>Overview</b>	<p>Guessmeme is a Solana bonding curve protocol that enables token creation and trading through an automated market maker. Users can create tokens with customizable parameters, but final metadata and minting authority must be set by an admin through the meta function.</p> <p>Users enter by calling the buy function to purchase tokens with SOL, and can exit through the sell function to receive SOL back. Both functions use a linear bonding curve formula to determine prices, with fees applied to each transaction.</p> <p>Only a hardcoded admin address (4B868oRgryogLJirzMkebmBgcWirKSUE1WUe6Z21F8o1) can withdraw SOL from the bonding curves, and this is only possible when a curve is marked as complete (when all tokens are sold). Token creators have no direct access to withdraw funds from their curves.</p>
<b>Audit Scope</b>	The scope of this Audit was to analyze the Guessmeme Smart Contracts for quality, security, and correctness.
<b>Source Code</b>	<a href="https://github.com/sardar-khan/guess-meme-contracts/tree/main/Solana"><u>https://github.com/sardar-khan/guess-meme-contracts/tree/main/Solana</u></a>
<b>Contracts in Scope</b>	lib.rs
<b>Branch</b>	Main

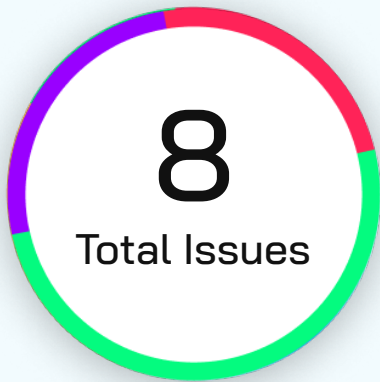


# Executive Summary

<b>Commit Hash</b>	716bdbd47cc3f1d896eb678f0fda0b288f26dfc1
<b>Language</b>	Rust
<b>Blockchain</b>	Solana
<b>Method</b>	Manual Analysis, Functional Testing, Automated Testing
<b>Review 1</b>	6th March 2025 - 13th March 2025
<b>Updated Code Received</b>	16th April 2025
<b>Review 2</b>	21st April 2025
<b>Fixed In</b>	<u><a href="https://drive.google.com/file/d/1wffKrTmRG7rwRRHtyyKF5ce1mKTmgIJ1/view?usp=drive_link">https://drive.google.com/file/d/1wffKrTmRG7rwRRHtyyKF5ce1mKTmgIJ1/view?usp=drive_link</a></u>



# Number of Issues per Severity



High	4 (50%)
Medium	3 (33%)
Low	1 (17%)
Informational	0 (0%)

		Severity			
		High	Medium	Low	Informational
Issues	Open	0	0	0	0
	Resolved	1	0	1	0
	Acknowledged	2	3	0	0
	Partially Resolved	1	0	0	0



# Checked Vulnerabilities

We have scanned the solana program for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

✓ **Signer authorization**

✓ **Account data matching**

✓ **Sysvar address checking**

✓ **Owner checks**

✓ **Type cosplay**

✓ **Initialization**

✓ **Arbitrary cpi**

✓ **Duplicate mutable accounts**

✓ **Bump seed canonicalization**

✓ **PDA Sharing**

✓ **Incorrect closing accounts**

✓ **Missing rent exemption checks**

✓ **Arithmetic overflows/underflows**

✓ **Numerical precision errors**

✓ **Solana account confusions**

✓ **Casting truncation**

✓ **Insufficient SPL token account verification**

✓ **Signed invocation of unverified programs**



# Techniques and Methods

Throughout the audit of Solana Programs, care was taken to ensure:

- The overall quality of code.
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

## Structural Analysis

In this step, we have analysed the design patterns and structure of Solana programs. A thorough check was done to ensure the Solana program is structured in a way that will not result in future problems.

## Static Analysis

Static analysis of Solana programs was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of Solana programs.





# Techniques and Methods

## Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

## Gas Consumption

In this step, we have checked the behaviour of Solana programs in production. Checks were done to know how much gas gets consumed and the possibilities of optimising code to reduce gas consumption.



# Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

## High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

## Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

## Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

## Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.



# Types of Issues

## Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

## Resolved

These are the issues identified in the initial audit and have been successfully fixed.

## Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

## Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

# High Severity Issues

## 1. The create instruction is susceptible to unlimited minting attack and pump/dump scenario

Partially Resolved

### Path

lib.rs

### Function Name

create

### Description

1. Unlimited Minting Exploits - With the dependence on admin calling the meta function, there are high chances for users to mint an unlimited amount of the tokens before admin action.

```
// 1. User calls create normally
create(
    initial_real_token_reserves: 1000,
    token_total_supply: 1000
)
// 2. Before admin calls meta, attacker
could:
mint_to_bonding_curve(additional_tokens)
// No checks to prevent this!
// Could mint unlimited tokens before
authority is revoked
```

2. Price Manipulation Attack - Users can manipulate the price as at the point of creating tokens so it is favorable for a pump and dump scenario.



```
create(  
  initial_virtual_token_reserves: 1_000_000, // Very high  
  initial_real_token_reserves: 100,          // Very low  
  token_total_supply: 1_000_000             // High  
)  
  
/ Results in:  
/ - Artificially high initial price  
/ - Scarcity illusion  
/ - Potential pump and dump
```

### Impact

1. Protocol are at risk of having more pump and dump tokens
2. Users can mint more than the standardized token supply

### Recommendation

Introduce some variables in the BondingCurve account that tracks when an initial mint has been completed and when metadata has been finalized. These variables could help stand as a check to prevent minting more than once after token creation. To keep all tokens to use a standard value when users create a token, use the global values.

```
pub struct BondingCurve {  
  ...  
  initial_mint_complete: bool,    // Prevents additional minting  
  metadata_finalized: bool,       // Tracks finalization  
}
```

### Client's Response

No one can mint more tokens as mint authority is provided to a pda and there is no public function in the contract to mint more tokens.

## 2. Zero Token Withdrawal Risk: Complete Status Only Triggered After Last Token Sale

**Acknowledged**

### Path

lib.rs

### Function Name

buy, withdraw

### Description

The bonding curve's "complete" status, which enables withdrawal, is only triggered when the last token is sold. This design means there will be no tokens left to withdraw when the withdraw authority calls the withdraw function.

```
// In buy instruction - Sets complete status
pub fn buy(ctx: Context<Buy>, amount: u64, max_sol_cost: u64) -> Result<()> {
    // ... other logic ...

    if ctx.accounts.bonding_curve.real_token_reserves == 0 {
        ctx.accounts.bonding_curve.complete = true; // Only set when no tokens remain

        emit!(CompleteEvent {
            mint: ctx.accounts.mint.key(),
            user: ctx.accounts.user.key(),
            bonding_curve: ctx.accounts.bonding_curve.key(),
            timestamp: Clock::get()?.unix_timestamp,
        });
    }
}

// In withdraw instruction - Expects tokens but none will exist cause all is sold
pub fn withdraw(ctx: Context<Withdraw>) -> Result<()> {
    require!(
        ctx.accounts.bonding_curve.complete,
        GuessError::BondingCurveNotComplete
    );

    // Attempts to withdraw tokens that don't exist
    let token_amount = ctx.accounts.associated_bonding_curve.amount; // Will be 0
    helpers::transfer_tokens_from_bonding_curve_to_admin(&ctx, token_amount)?;
}
```

### Impact

1. There will be no left over tokens to withdraw

### Recommendation

See issue H-4.



### 3. Possible lock of funds due to the tendency of complete status remaining active

**Acknowledged****Path**

lib.rs

**Function Name**

buy

**Description**

The bonding curve's funds can become permanently locked if all tokens are not sold. The withdraw function requires complete status, which is only set when `real_token_reserves` reaches zero, creating a potential deadlock scenario. Complete status solely depends on selling all tokens (Not 80% of tokens). There is a possibility that the bonding curve will not sell tokens until the real token reserve value becomes 0.

```
// Complete status only set when all tokens sold
pub fn buy(ctx: Context<Buy>, amount: u64, max_sol_cost: u64) -> Result<()> {
    // ... other logic ...
    if ctx.accounts.bonding_curve.real_token_reserves == 0 {
        ctx.accounts.bonding_curve.complete = true;
    }
}

// Withdrawal requires complete status
pub fn withdraw(ctx: Context<Withdraw>) -> Result<()> {
    require!(
        ctx.accounts.bonding_curve.complete, // ✗ Strict dependency
        GuessError::BondingCurveNotComplete
    );
    // ... withdrawal logic ...
}
```

**Impact**

1. SOL could be permanently locked if tokens aren't fully sold
2. No recovery mechanism for inaccessible stuck funds

**Recommendation**

Fix the code implementation to be dependent on the sale of 80% tokens. This will indirectly fix the issue H3, ensuring that there are 20% left over tokens in the contract to be withdrawn alongside SOL..

The flow is that the bonding curve will only be complete until all the tokens are bought and until its complete admin has no authority over it. Funds won't be stuck. Users can just sell their bought tokens and receive the sols.



## 4. Missing Token Account Owner Validation Check

**Resolved**

### Path

lib.rs

### Function Name

buy/sell

### Description

The buy and sell functions don't properly validate that the token accounts provided belong to the expected owners. This could allow attackers to pass in token accounts they don't own or manipulate the transaction with incorrect token accounts. There is no check during Account declaration or when users interact with these functions.

```
pub fn buy(ctx: Context<Buy>, amount: u64, max_sol_cost: u64) -> Result<()> {  
    // ✖ No validation that associated_user is owned by ctx.accounts.user  
    helpers::transfer_tokens_from_bonding_curve_to_user(  
        &ctx,  
        famount  
    )?;  
}  
  
pub fn sell(ctx: Context<Sell>, amount: u64, min_sol_output: u64) -> Result<()> {  
    // ✖ No validation that associated_user is owned by ctx.accounts.user  
    helpers::transfer_tokens_from_user_to_bonding_curve(  
        &ctx,  
        amount  
    )?;  
}
```

### Impact

1. Attackers could potentially pass in token accounts they don't own
2. Possible theft of tokens if validation is bypassed
3. Risk of tokens being sent to incorrect accounts

### Recommendation

Add quality checks in the account declaration and on the functions for runtime check





# Medium Severity Issues

## 5. Unchecked SOL Balance in Sell Function: Risk of Failed Transfers Due to Insufficient Balance

Acknowledged

### Path

lib.rs

### Function Name

sell

### Description

The sell function updates the bonding curve state and attempts transfers without first validating if the bonding curve has sufficient SOL balance. This could lead to failed transfers and inconsistent state. There is no validation of the actual SOL balance before the state update, assuming that the `real_sol_reserves` matches the actual balance in the bonding curve. In essence, it violates the checks-effects-interaction pattern with the missing checks.

### Recommendation

Add check in sell operation to confirm that the bonding curve has sufficient SOL to perform txs.

### Client's Response

Not possible, the bonding curve is completed when all tokens are sold. The sols acquired at that time can not be withdrawn until the curve is complete so there is no way one can manipulate the sols, they will always be on a bonding curve and can be withdrawn after it's completed.



# Medium Severity Issues

## 6. Rent set aside for storage is never redeemed

**Acknowledged**

### Description

In the pub fn withdraw() after the bonding curve has been completed and data cleared, it does not refund the rent set aside to pay for storage, note: The amount of SOL that can be earned from closing an account is proportional to how large the account was.

### Recommendation

Redeem rent fee and close accounts.

### Client's Response

Not possible, the bonding curve is completed when all tokens are sold. The sols acquired at that time can not be withdrawn until the curve is complete so there is no way one can manipulate the sols, they will always be on a bonding curve and can be withdrawn after it's completed.

## 7. max\_allowed\_amount will be bypassed by users

**Acknowledged**

### Description

Each token created has a max\_supply\_percent that should not be bypassed, however the current check allows users to bypass the max\_allowed\_amount:

```
Let max_allowed_amount: u64 = (ctx.accounts.bonding_curve.token_total_supply
    * ctx.accounts.bonding_curve.max_supply_percent as u64)
```

```
require!(
    (ctx.accounts.associated_user.amount + famount) <= max_allowed_amount,
    GuessError::MaxSupplyExceeded
);
```

notice the check  $(ctx.accounts.associated\_user.amount + famount) \leq max\_allowed\_amount$  the issue here is the check is not on the total sold but the amount the user already has however this is wrong as users who have exceeded the max allowed, can transfer their amount out before calling buy tokens, passing the max\_allowed, then transfer their amount back in. This is possible because the max\_allowed was not checked on the total sold but on the account current balance.



**Recommendation**

add a mutable totalSold variable to the bonding\_curve and check on that

```
Let mac_allowed_amount: u64 = (ctx.accounts.bonding_curve.token_total_supply  
    * ctx.accounts.bonding_curve.max_supply_percent as u64)
```

```
require!(  
    (ctx.accounts.bonding_curve.token_total_sold + famount) <=  
    max_allowed_amount,  
    GuessError::MaxSupplyExceeded  
);  
ctx.accounts.bonding_curve.token_total_sold += famount // do opposite on sell
```

**Client's Response**

I have discussed with the Solana developer its user base not transaction base. If max\_percentage is set to 80%, each user can hold up to 80% of the tokens—no more.



# Low Severity Issues

## 8. No event emission for the withdraw operation

**Resolved****Path**

lib.rs

**Function Name**

withdraw

**Description**

The withdraw instruction, which handles the final withdrawal of SOL and any remaining tokens after bonding curve completion, lacks event emission for transparency and off-chain tracking.

**Impact**

1. Difficult to track protocol withdrawals
2. No off-chain monitoring of admin actions
3. Missing historical data on withdrawals for analysis

**Recommendation**

Add event emission and proper tracking.



# Functional Tests

Some of the tests performed are mentioned below:

- ✓ Should test for all possible reverts on each function
- ✓ Should initialize global state account and compare with created tokens
- ✓ Should allow users buy and sell into a created bonding curve
- ✓ Should test that recognized authority can invoke permissioned function

# Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.



# Closing Summary

In this report, we have considered the security of Guessmeme. We performed our audit according to the procedure described above.

Issues of 4 - High, 3- Medium and 1- Low severity issues were found ,out of those Guessmeme team resolved a few and acknowledged others.

# Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.



# About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over \$30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

**7+**

Years of Expertise

**1M+**

Lines of Code Audited

**\$30B+**

Secured in Digital Assets

**1400+**

Projects Secured

**Follow Our Journey**

# AUDIT REPORT

---

April 2025

For



**Guess.Meme**



Canada, India, Singapore, UAE, UK

[www.quillaudits.com](http://www.quillaudits.com)

[audits@quillaudits.com](mailto:audits@quillaudits.com)