QuillAudits

AUDIT REPORT

April , 2025

For

Guess.Meme

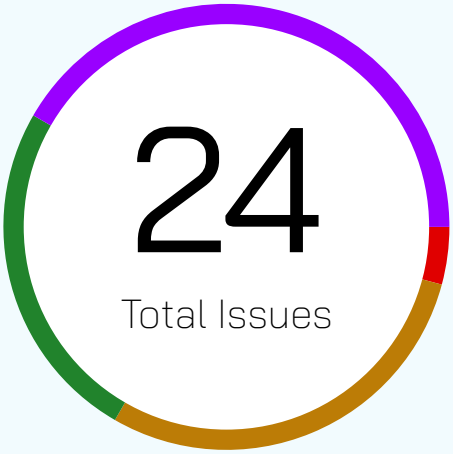
Table of Content

Executive Summary

Project name	Guessmeme
Project URL	https://guessmemelive.netlify.app/
Overview	Guessmeme uses Factory and ERC20 Token contract. The Factory is used to create sales of standard ERC20 tokens using a bonding curve where users can buy the listed tokens with ETH and sell the tokens back to the pool. Once the real token reserve reaches 0 i.e. all the tokens are sold, the contract owner can withdraw the tokens (remaining tokens from supply) and ETH.
Audit Scope	The scope of this Audit was to analyze the Guessmeme Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/sardar-khan/guess-meme-contracts/blob/main/ethereum/guess_meme_latest.sol
Contracts in Scope	guess_meme_latest.sol
Branch	main
Commit Hash	90c60ced71896b87b52dbf67da2dd1f82130ed40
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	6th March 2025 - 14th March 2025

Updated Code Received	16th April 2025
Review 2	21st April 2025
Fixed In	https://drive.google.com/file/d/1wffKrTmRG7rwR-RHtyyKF5ce1mKTmglJ1/view?usp=drive_link

Number of Issues per Severity



High	1 (4.17%)
Medium	7 (29.17%)
Low	6 (25.00%)
Informational	10 (41.67%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	0	**2**	**5**	**2**
Acknowledged	**1**	**4**	**1**	**8**
Partially Resolved	0	**1**	0	0

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, 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 analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

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 analyzed, 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 behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

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 Issues

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

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

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

The updated functionality of the max buy limit can by bypassed

Acknowledged

Path

guess_meme_latest.sol#L752-L754

Function

buyTokens()

Description

The new change added to validate if the user should be able to buy the tokens based on the max limit introduces a bug where msg.sender will be able to send his tokens to another address while buying, just to buy more amount and pass the check/validation where the current balance of msg.sender plus _amount is validated.
Example:
1.User1 bought the maximum amount of tokens that can be bought by one user (based on the maxSupplyPercentage)
2.User1 calls buyTokens() again but before that the User1 transfers some or the whole amount to another address, so that on L754 when the user's balance will be fetched, it will be 0. And the user will be able to buy an additional allowedToBuy amount of tokens.
3.This action can be repeated multiple times.

(Note: It should be noted that even if other functionality will be used where the user's bought amount will be stored in the variable for each user's address, the user can still create different addresses to trick that functionality.)

Recommendation

Remove the vulnerable functionality to validate the maximum amount bought through allowedToBuy

Medium Severity Issues

Unable to complete the bonding curve

Resolved

Path

eth_guessmeme.sol#L766

Function

withdraw(), buyTokens()

Description

The withdraw() can be called when the bonding curve is reached, i.e., when the realTokenReserves is 0. But because while buying, it checks that the user should buy less than or equal to maxSupplyPercentage percent, there are chances that realTokenReserves will never reach to 0.

let's say the maxSupplyPercentage is 80, which means the user can buy 80% of realTokenReserves amount in a single transaction. And the requirement to set isCompleted to true is when realTokenReserves == 0.
So if every time the user can only buy 80% of the remaining realTokenReserves,
while buying, reaching realTokenReserves to 0 is not always possible. while it can reach close to 0 (e.g. 1) but not 0. its because e.g. the realTokenReserves reaches 1 but the 80% of 1 would be 0 in solidity, and that's why it can't be even specified in the buyTokens() function call.

Audit team's comment

Since the current functionality to check the user is buying in the allowed limit is changed based on the user's current balance, this issue is set to resolved.

Recommendation

To resolve this issue withdraw() can be modified to check if at least some percent of the realToken-Reserves are sold then the bondingCurve[_token].isCompleted can be set to true. this can be any number like 99%. So the code in withdraw() will check that if the 99% of initial realTokenReserves (793100000 *10**18) is sold then the bondingCurve[_token].isCompleted can be set to true.
The modified code would look like this:

```
 function withdraw(address _token) external onlyOwner {
 require(isToken[_token], "Token doesn't exist!");

 uint256 onePercentOfInitial = (793100000 *10**18) / 100;
 if(bondingCurve[_token].realTokenReserves<=onePercentOfInitial){
 bondingCurve[_token].isCompleted = true;
}

 require(
 bondingCurve[_token].isCompleted,
"Bonding curve not completed yet"
);
 uint256 ethToTransfer = bondingCurve[_token].realEthReserves;
 uint256 tokensToTransfer = IERC20(_token).balanceOf(address(this));
 uint256 feeTokens = (bondingCurve[_token]._totalSupply * 1) / 100; //1% tokens fee
 bondingCurve[_token].virtualTokenReserves = 0;
 bondingCurve[_token].virtualEthReserves = 0;
 bondingCurve[_token].realEthReserves = 0;
 payable(msg.sender).transfer(ethToTransfer);
 IERC20(_token).transfer(msg.sender, tokensToTransfer - feeTokens);
 IERC20(_token).transfer(tokenFeeRecipient, feeTokens);
 emit Withdraw(_token, ethToTransfer, tokensToTransfer);
}
```

Add slippage checks

Acknowledged

Path

eth_guessmeme.sol#L724, eth_guessmeme.sol#L749

Function

buyTokens(), sellTokens()

Description

In buyTokens() the ETH amount that the user needs to pay for buying a certain _amount of tokens can vary if other buyTokens() transactions execute before it.
If other buyTokens() transactions execute before the user's buyTokens() transaction then the amount of eth user needs to pay for buying the same amount of _amount would be more than what was calculated before.
In this case, care can be taken by sending more amount eth as the buyTokens() allows sending msg.value greater than ethToBuy + fee.
(Note: As mentioned in another issue the extra dust ETH amount should be sent back to the user/msg.sender.)
In sellTokens() the amount of ETH the user gets after selling _amount tokens can vary if other sellTokens() transactions execute before the user's call where the user was expecting to receive the specific ETH amount calculated with sellQuote().
Because of this, it's important to provide the slippage limit while selling tokens so that it can be checked if the amount is not much less than what was expected to be received.

Recommendation

This slippage limit (or the limit for a minimum amount to receive) can be passed by the user while calling sellTokens(). So the sellTokens() will check that if the ethToReceive amount is less than what is expected amount by the user and will revert.
The code to check it would look like this:

```
function sellTokens(address _token, uint256 _amount, uint256 _expectedMinEthToReceive) external
{
 require(isToken[_token], "Token doesn't exist!");
 require(_amount > 0, "Can't sell zero tokens");
 require(!bondingCurve[_token].isCompleted, "Bonding curve completed");
 uint256 ethToReceive = sellQuote(_token, _amount);
 require( (_expectedMinEthToReceive <= ethToReceive),"INSUFFICIENT_AMOUNT" );
 uint256 fee = (ethToReceive * feeBasisPoints) / 10_000;
 IERC20(_token).transferFrom(msg.sender, address(this), _amount);
 bondingCurve[_token].virtualTokenReserves += _amount;
 bondingCurve[_token].realTokenReserves += _amount;
 bondingCurve[_token].virtualEthReserves -= ethToReceive;
 bondingCurve[_token].realEthReserves -= ethToReceive;

 payable(feeRecipient).transfer(fee);
 payable(msg.sender).transfer(ethToReceive - fee);
 emit TokensSold(msg.sender, _token, _amount);
}
```

Here it will check that the ethToReceive that the user will receive is equal to or greater than _expectedMinEthToReceive.

Send dust/extra eth amount back in buyTokens()

Acknowledged

Path

eth_guessmeme.sol#L724

Function

buyTokens()

Description

buyTokens() allows sending msg.value that is greater than or equal to ethToBuy + fee.
This means an extra amount can be sent than what is required to buy _amount. Which will help users to handle some slippage (if ETH price for token goes up than what was calculated).

However, since the extra amount will not be utilized every time. It is necessary to send the extra/dust ETH amount back to the msg.sender.

Recommendation

Send the extra amount of ETH back to the msg.sender at the end of function, before emitting the event. The code to send extra/dust amount would look like this:

```
if (msg.value > (ethToBuy + fee)) {
 payable(msg.sender).transfer(msg.value - (ethToBuy + fee));
}
```

private variables will lack privacy on the public blockchain

Resolved

Path

eth_guessmeme.sol#L265-L277

Function

name(), symbol()

Description

Currently, the assumption that the name() and symbol() functions will return name and symbol only when the block.timestamp is greater than _reveal timestamp is incorrect as even private variables can be read in various ways e.g. By checking the previous token creation transaction and by fetching the storage slot value, etc.
That's why it's important to verify and change the logic to avoid this issue.

Audit team's comment

The functionality of returning the real name and symbol based on reveal timestamp is removed and the updateMetadata() will be used at the time of revealing the token details.

Recommendation

Consider verifying and changing the logic to avoid this issue.

Use multisig wallet to mitigate centralization concern

Partially Resolved

Path

eth_guessmeme.sol

Function

-

Description

Since some critical functionality like withdrawals of the fund is managed by a single EOA/owner address. It's important that a multi-sig should be used as an owner for managing the functionality. Additionally, it should be noted that some functions like withdrawStuckEth(), transferStuckTokens() create centralization concerns as they grant direct access to the owner to remove the ETH and tokens without any restrictions.

Audit team's comment

Partially resolved by removing withdrawStuckEth() and transferStuckTokens(). The withdraw() still remains in the contract which is required for owner withdrawals

Recommendation

Use multisig as an owner to manage critical functionality.

Restrict the minimum token amount while buying and selling

Acknowledged

Path

eth_guessmeme.sol#L724, eth_guessmeme.sol#L749

Function

buyTokens(), sellTokens()

Description

While buying, a very small _amount can lead the ethToBuy to 0 and eventually fee calculated based on ethToBuy would be 0, the minimum limit should be added while buying the tokens with buyTokens(). This limit can be added by calculating ethToBuy and fee for a certain _amount and checking that it would not be 0.

E.g. In buyTokens() it can be checked that if _amount is 1000000000000000 (i.e. 0.001e18) the ethToBuy and fee are non-zero for hardcoded reserve amounts. so based on that the limit can be set for the _amount to be greater than or equal to 1000000000000000.

Similarly, while selling with sellTokens() very small amount can lead ethToReceive to go close to 0. The limit can be added by calculating ethToReceive and fee, for a certain _amount and checking that it would not be 0.

E.g In sellTokens() if _amount is 1000000000000000 (i.e. 0.001e18) the ethToReceive and fee is non-zero for hardcoded reserve amounts. so based on that the limit can be set for the _amount to be greater than equal to 1000000000000000.

Recommendation

Set the limit in both buyTokens() and sellTokens() to check that the entered _amount is greater than or equal to 1000000000000000 and if not the transaction should revert.
The limit can be assigned to the variable and can have its setter method so the amount can be adjusted according to future requirements.

The code for adding a limit would look like this:
require(_amount >= 1000000000000000,"INVALID_AMOUNT");

Care should be taken for an incomplete bonding curves

Acknowledged

Path

eth_guessmeme.sol

Function

-

Description

According to the code bondingCurve[_token].isCompleted is set to true, if the bondingCurve[_token].realTokenReserves becomes 0 on L739 while the user buys tokens.

It should be noted that it's not guaranteed that all the tokens will be sold in the specific period and hence the ETH accumulated by selling those tokens can't removed as the bonding curve is not complete because the realTokenReserves has not reached 0.

Recommendation

Consider verifying the logic and add the required changes to remove the ETH for the above mentioned scenario.

If this is implemented, there should be certain restrictions in that functionality E.g. the code will allow removal of ETH only when a certain token sale is inactive for let's say 6 month period. To know if the sale for the specific token is inactive for a certain period, every time the last active timestamp can be updated in the buy functionality. Which then can be used later in the emergency withdrawal functionality that will be used in these scenarios.

Low Severity Issues

Add the max limit for changeFeeBasisPoints()

Resolved

Path

eth_guessmeme.sol#L821

Function

changeFeeBasisPoints()

Description

Currently, 100% of the fee can be set through feeBasisPoints using changeFeeBasisPoints() function. It should be capped to a specific number to improve users' transparency so that the fee won't exceed a specific percentage.

Recommendation

Add the fee limit for entered _feeBasisPoints. The code to add restriction would look like this: require(_feeBasisPoints <= 3000, "INVALID_FEE"); this will allow setting fee only upto 30% for example.

Floating pragma

Resolved

Path

eth_guessmeme.sol

Function

-

Description

Multiple contracts are using version (^0.8.0 and ^0.8.26) with floating pragma instead of locking to a specific version. floating pragmas allow the contract to be compiled with any version greater than or equal to the specified version. If the contract wasn't thoroughly tested with that version, this can introduce possible bugs.

Additionally, Only one solidity version can be used for the contract.

Recommendation

Remove the ^ symbol to consider using a fixed solidity version with which the contracts are thoroughly tested.

Add restrictions to avoid token dumping

Acknowledged

Path

eth_guessmeme.sol

Function

-

Description

Users can sell the tokens after buying even while the sale is live.
We encourage your team to analyze and understand the implications of this scenario and add certain restrictions for the users before they start moving the funds.

Recommendation

Time restriction can be added where only after a certain timestamp is reached the users can move their bought tokens.

Assumption about withdraw() fee subtraction logic

Resolved

Path

eth_guessmeme.sol#L779

Function

withdraw()

Description

In withdraw() on L779 while subtracting feeTokens amount from tokensToTransfer, it's assumed that it will be always lesser than tokensToTransfer so there won't be any arithmetic errors while subtraction.

It is assumed that feeTokens would be always lesser than tokensToTransfer because feeTokens is 1% of _totalSupply. Given that realTokenReserves is 793100000 *10**18 and _totalSupply would be 1000000000e18. when the withdraw() is executed the realTokenReserves will have already reached 0. That means 1000000000e18 - 793100000e18 = 206900000e18 is the current remaining amount that the contract holds. which is around 20.69% of the 1000000000e18.

So on L774 when calculating 1% of 1000000000e18 it would be 10000000e18 which is less than the remaining 20.69% (206900000e18). Hence there should not be any underflow error in the tokensToTransfer - feeTokens subtraction on L779.

Note: Because depending on the _totalSupply (if it's different than 1 billion) entered, while withdrawing the remaining amount in the contract (tokensToTransfer) will vary. if the tokensToTransfer is less than feeTokens, it can create the arithmetic error.

Recommendation

The mentioned scenario should be taken into account, and if there will be different supply and realTokenReserves will be set, the required changes should be done in the withdraw() to handle the case where the tokensToTransfer would be less than the feeTokens.

Hardcode the totalsupply

Resolved

Path

guess_meme_latest.sol

Function

-

Description

As discussed internally, the _totalSupply will be 1 billion for every token sale that is created. Still, it's good practice to set the hardcoded value to the _totalSupply to avoid any unexpected results as mentioned in other issues.

Recommendation

Hardcode the _totalSupply.

sellQuote rounds up the result

Resolved

Path

guess_meme_latest.sol

Function

sellQuote()

Description

The sellQuote() returns the ETH amount that the users will receive for selling tokens. This function rounds up the ETH to receive.

So if the user returns the same bought token amount, the formula will return a slightly higher ETH value with 1 wei, as compared to what was required while buying.

sellQuote() should never round up because in sell, the eth returned is the eth amount users will receive from the protocol.

Recommendation

Change the formula where the sellQuote() won't round up the result. The updated code would look like this:
uint256 ethOutput = (_amount * virtualEthReserves / (virtualTokenReserves + _amount));

Informational Severity Issues

Unused variable

Acknowledged

Path

eth_guessmeme.sol#L652

Function

-

Description

tokenCount is declared on L652 is increamented in createToken() on L720. This variable is then never used anywhere and the variable is declared as private so no one can directly know how many tokens are created as there would be no getter method created for private variable.

Recommendation

Consider making it public so the getter method can be used to know it’s value.

Note regarding hardcoded reserves

Acknowledged

Path

eth_guessmeme.sol#L714-L716

Function

-

Description

It should be noted that the reserves are hardcoded in the contract that means every time a new token sale is created, the same reserve values will be used.

Recommendation

This should be acknowledged.

Add validation in sellTokens()

Acknowledged

Path

eth_guessmeme.sol#L749

Function

sellTokens()

Description

Validation should be added to check if the contract has enough amount of ETH available when the user sells a certain amount of tokens back to the contract to avoid unexpected failing.

Recommendation

Add validation as suggested.

Variables can be declared as constant and immutable

Acknowledged

Path

eth_guessmeme.sol

Function

-

Description

initialVirtualEthReserves set the value of 1900000000000000000 and is never changing hence it can be declared as a constant.
feeRecipient and tokenFeeRecipient can be declared as immutable variables.

Recommendation

Consider making variables constant and immutable as suggested.

Note regarding withdraw functionality

Acknowledged

Path

eth_guessmeme.sol

Function

withdraw()

Description

This should be noted that for every created token sale/pool only the protocol owner can withdraw the tokens and not the creator of that pool.

Recommendation

This should be acknowledged.

Note about ownership of the smart contract

Acknowledged

Path

eth_guessmeme.sol

Function

-

Description

It should be noted that when deploying this smart contract the ownership is transferred to the deployer of the smart contract, unlike the current OpenZeppelin ownable implementation where you get the option to pass the initialOwner argument. So the deployer would be able to call the authorized functions like withdraw(), changeFeeBasisPoints(), withdrawStuckEth(), transferStuckTokens().

Recommendation

This can be acknowledged after noting the behavior.

Pause functionality can be added

Acknowledged

Path

eth_guessmeme.sol

Function

buyTokens(), sellTokens()

Description

While the pausing functionality should be normally avoided. Given that the contracts are non-upgradeable, it would be a good idea to add pausing functionality for buy and sell functionality to ensure the safe withdrawal of funds by the owner in any unexpected event.

Recommendation

The project team should independently think about adding pause functionality as it will also introduce the centralization concern if added.

Change the comment to match the feeBasisPoints value

Resolved

Path

guess_meme_latest.sol

Function

-

Description

feeBasisPoints is set to 100. and the comment mentions 0.01 %.

The fee calculation works like this:
(ethToBuy * feeBasisPoints) / 10_000;
Lets say ethToBuy=1e18 and feeBasisPoints = 100, then:
= (1e18 * 100) / 10_000
= 1000000000000000000000 / 10_000
= 10000000000000000 (it is 1 % of the 1e18 and not 0.01 % of 1e18(ethToBuy))

Recommendation

Change the comment to 1%.

Change the variable name to avoid confusion

Resolved

Path

guess_meme_latest.sol#L673

Function

-

Description

variable name initialRealEthReserves should be changed to initialRealTokenReserves. initial-RealEthReserves name is not correct as the real eth reserve would be 0 initially.

Recommendation

Verify and change the variable name to initialRealTokenReserves.

Note about updateMetadata

Acknowledged

Path

guess_meme_latest.sol

Function

updateMetadata()

Description

updateMetadata() updates/reveals the token name and symbol. This should be noted that the updateMetadata() is owner only function and for every token that is created by users, the owner of the factory contract can change the name and symbol, and not the user who created that token.

Recommendation

This should be acknowledged.

Functional Tests

Some of the tests performed are mentioned below:

- ✔ Bonding curve should be completed
- ✔ User should be able to buy tokens
- ✔ User should be able to sell tokens
- ✔ User should be able to calculate buy fee
- ✔ Only owner should be able to change the fee basis points
- ✔ Owner should be able to withdraw stucked ethers
- ✔ Owner should be able to withdraw stucked tokens
- ✔ Multiple users should be able to buy tokens
- ✔ Multiple users should be able to sell tokens
- ✔ Users should be able to create tokens using the factory
- ✔ Owner should be able to update the token metadata

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 1 - high,7- medium, 6- low, and 10- informational severity 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 secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

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

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com