QuillAudits

AUDIT REPORT

August 2025

For

AQUA BOT

Table of Content

Executive Summary

Project Name	Aqua bot
Protocol Type	Telegram Bot
Project URL	https://aquabot.io/
Overview	Aqua Ladder – a tiered participation system where users can contribute SOL tokens within specific constraints (1 to 30 SOL per participant). The ladder consists of 10 predefined levels with varying SOL capacity, and the size of each level grows proportionally to the overall liquidity of the ladder. The random seed generation combines block timestamp and slot number for unpredictability. This seed is then used by a public algorithm that anyone can verify and execute, ensuring a fair and transparent level assignment for each participant. All SOL contributions are transferred to a designated multisig wallet, maintained collectively by team members and strategic partners. 100% of these funds will be deposited as liquidity into a Meteora DLMM pool at launch, ensuring deterministic handling of participant funds and alignment with the protocol's liquidity goals.
Audit Scope	The scope of this Audit was to analyze the Aqua Bot Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/darwindowndev/ladder-sc/tree/main/anchor-sc
Branch	Main
Contracts in Scope	src/*
Commit Hash	c443ce52e4b8b67eadadab924d8ce16018e6544e
Language	Rust
Blockchain	Solana
Method	Manual Analysis, Functional Testing, Automated Testing

Review 1	25th August 2025 - 28th August 2025
Updated Code Received	29th August 2025
Review 2	29th August 2025
Fixed In	3dd332f51c7c2e90b04ff979b8b7e0fc3b9cd92b

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity

🟥	Critical	0 (0.0%)
🟥	High	0 (0.0%)
🟧	Medium	0 (0.0%)
🟨	Low	3 (100%)
🟪	Informational	0 (0.0%)

Severity

Issues

	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	**3**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Partial requests denied because of over strict check	Low	Resolved
2	Sanitisation of resolve_ladder	Low	Resolved
3	ladder can accept contributions beyond its intended capacity	Low	Resolved

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
- 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.

Types of Severity

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

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

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.

Severity Matrix

		Impact		
		🟥 High	🟧 Medium	🟨 Low
Likelihood	🟥 High	Critical	High	Medium
	🟧 Medium	High	Medium	Low
	🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Low Severity Issues

Partial requests denied because of over strict check

Resolved

Path

anchor-sc/programs/aqua-ladder-sc/src/instructions/participate_in_ladder.rs

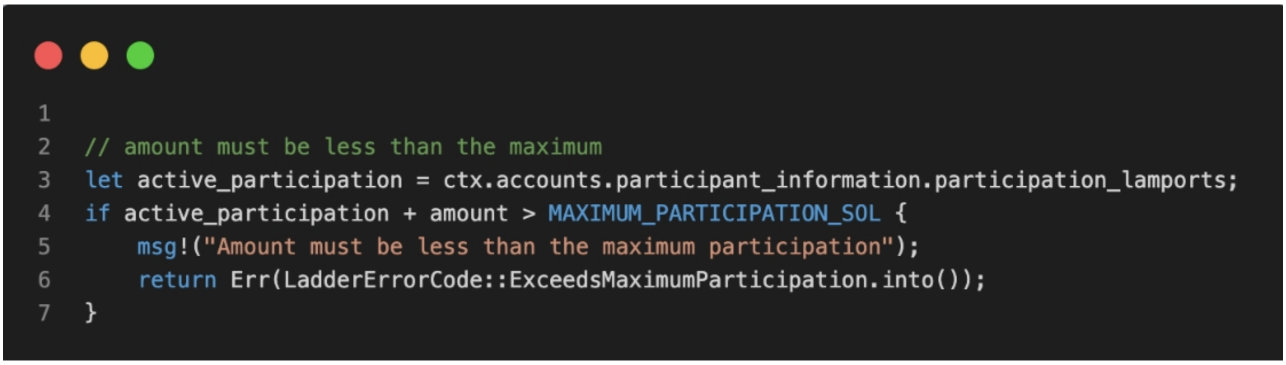
Function

`participate_in_ladder`

Description

The **`participate_in_ladder`** function strictly rejects transactions when a user attempts to participate with an amount that would exceed the maximum allowed participation, instead of accepting a partial amount up to the maximum limit.

In the current implementation, when a user attempts to participate in a ladder with an amount that would exceed the maximum participation limit (`` `MAXIMUM_PARTICIPATION_SOL` ``), the transaction is completely rejected with an **`ExceedsMaximumParticipation`** error.



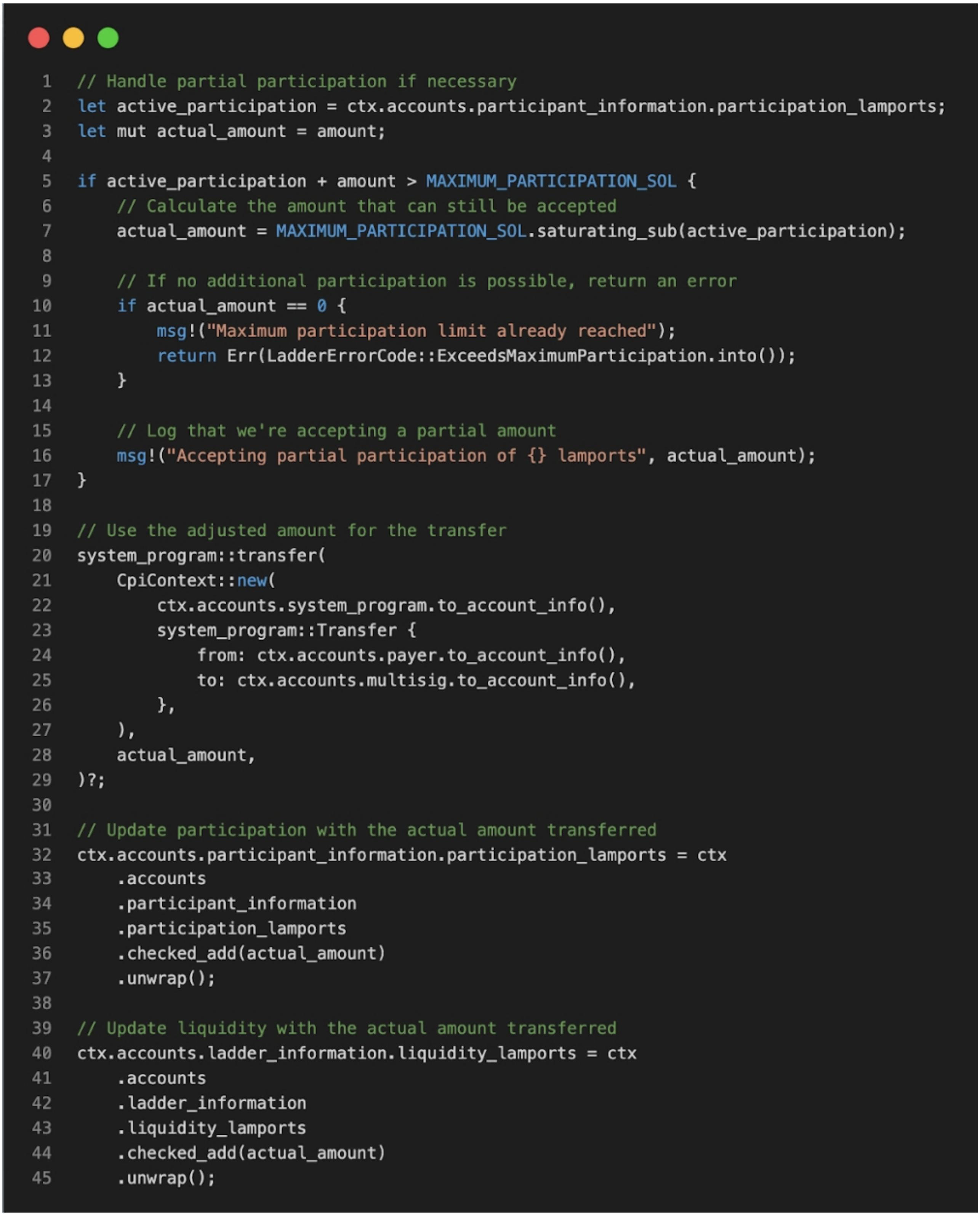
```

// amount must be less than the maximum
let active_participation = ctx.accounts.participant_information.participation_lamports;
if active_participation + amount > MAXIMUM_PARTICIPATION_SOL {
    msg!("Amount must be less than the maximum participation");
    return Err(LadderErrorCode::ExceedsMaximumParticipation.into());
}
```

Instead of rejecting the entire transaction, a more user-friendly approach would be to accept a partial amount that brings the user's total participation up to the maximum limit.

Remediation

Modify the implementation to accept partial participation amounts when the requested amount would exceed the maximum limit:



```rust
// Handle partial participation if necessary
let active_participation = ctx.accounts.participant_information.participation_lamports;
let mut actual_amount = amount;

if active_participation + amount > MAXIMUM_PARTICIPATION_SOL {
    // Calculate the amount that can still be accepted
    actual_amount = MAXIMUM_PARTICIPATION_SOL.saturating_sub(active_participation);

    // If no additional participation is possible, return an error
    if actual_amount == 0 {
        msg!("Maximum participation limit already reached");
        return Err(LadderErrorCode::ExceedsMaximumParticipation.into());
    }

    // Log that we're accepting a partial amount
    msg!("Accepting partial participation of {} lamports", actual_amount);
}

// Use the adjusted amount for the transfer
system_program::transfer(
    CpiContext::new(
        ctx.accounts.system_program.to_account_info(),
        system_program::Transfer {
            from: ctx.accounts.payer.to_account_info(),
            to: ctx.accounts.multisig.to_account_info(),
        },
    ),
    actual_amount,
)?;

// Update participation with the actual amount transferred
ctx.accounts.participant_information.participation_lamports = ctx
    .accounts
    .participant_information
    .participation_lamports
    .checked_add(actual_amount)
    .unwrap();

// Update liquidity with the actual amount transferred
ctx.accounts.ladder_information.liquidity_lamports = ctx
    .accounts
    .ladder_information
    .liquidity_lamports
    .checked_add(actual_amount)
    .unwrap();
```

Sanitisation of resolve_ladder

Resolved

Path

anchor-sc/programs/aqua-ladder-sc/src/instructions/resolve_ladder.rs

Function Name

`handle_resolve_ladder`

Description

The **`handle_resolve_ladder`** function lacks validation checks before resolving a ladder, specifically not verifying if the ladder cap has been reached and not validating that the ladder is not already finished.

The **`handle_resolve_ladder`** function is responsible for finalizing a ladder by setting its state to finished and generating a randomized seed for determining results. However, the function proceeds with the resolution process without any preliminary checks to ensure:

1. The ladder cap has been reached, which should be a prerequisite for resolution
2. The ladder is not already finished (**`` `is_ladder_finished` ``** is not already true)

The function currently:
- Generates a random seed using block timestamp and slot
- **`Sets is_ladder_finished`** to true
- Records the closing timestamp
- Does not perform any validation checks before proceeding

This allows the admin to potentially resolve a ladder prematurely before the cap is reached or to re-resolve an already finished ladder, which could lead to unexpected behavior or exploitation.

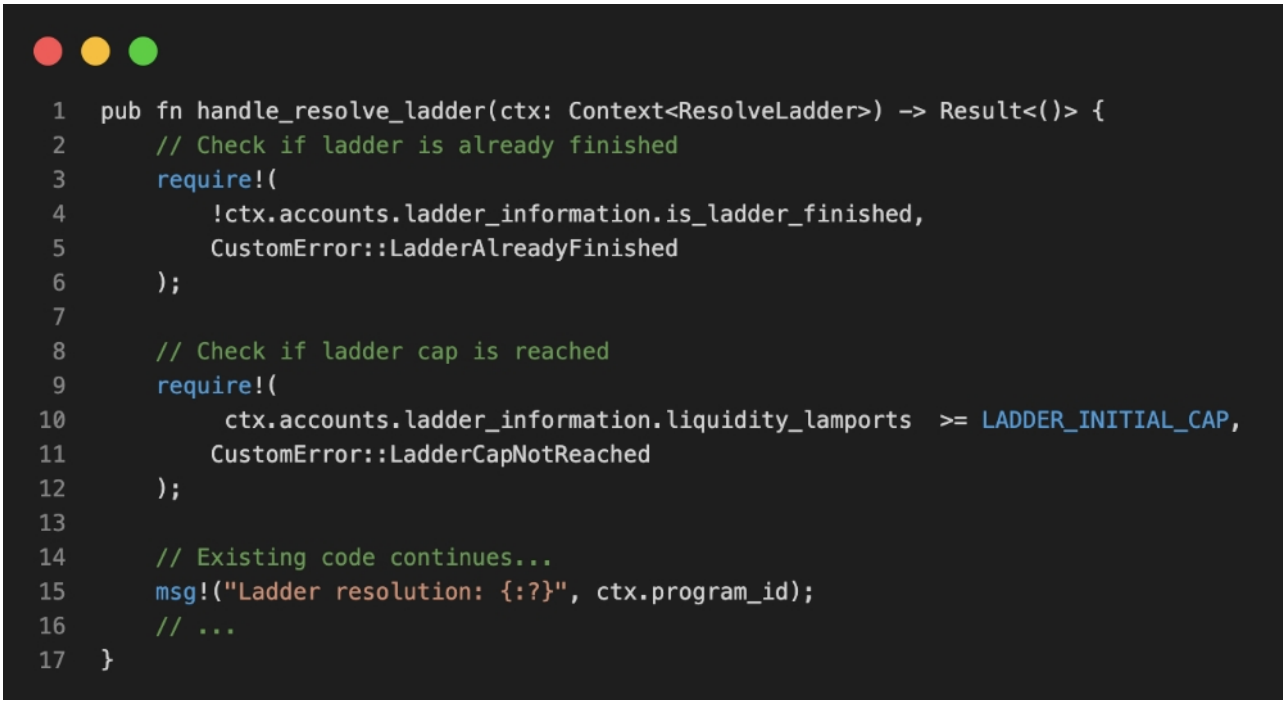
Impact

This vulnerability could allow:

- Premature resolution of ladders before all participants have joined
- Potential re-resolution of already completed ladders, which could overwrite previous results
- Manipulation of outcome timing to potentially affect the randomization factor
- Undermining user trust in the fairness of the ladder resolution mechanism

Remediation

Add the following validation checks at the beginning of the **`handle_resolve_ladder`** function:



```rust
pub fn handle_resolve_ladder(ctx: Context<ResolveLadder>) -> Result<()> {
    // Check if ladder is already finished
    require!(
        !ctx.accounts.ladder_information.is_ladder_finished,
        CustomError::LadderAlreadyFinished
    );

    // Check if ladder cap is reached
    require!(
         ctx.accounts.ladder_information.liquidity_lamports  >= LADDER_INITIAL_CAP,
        CustomError::LadderCapNotReached
    );

    // Existing code continues...
    msg!("Ladder resolution: {:?}", ctx.program_id);
    // ...
}
```

ladder can accept contributions beyond its intended capacity

Resolved

Path

anchor-sc/programs/aqua-ladder-sc/src/instructions/participate_in_ladder.rs

Function Name

`handle_participate_in_ladder`

Description

The handle_participate_in_ladder function lacks a check to ensure that the total participation does not exceed the ladder's initial cap (`` `LADDER_INITIAL_CAP` ``), allowing users to continue participating even after the cap has been reached.

The handle_participate_in_ladder function performs several validations before allowing a user to participate in the ladder:
- It checks if the ladder is initialized
- It checks if the ladder is not finished
- It verifies that the participation amount is greater than the minimum allowed
- It ensures that a participant's total contribution doesn't exceed the maximum individual participation limit

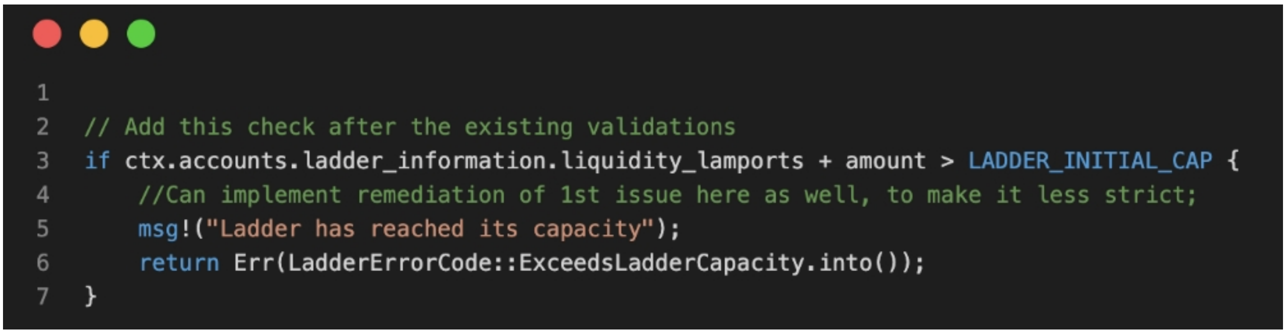
However, the function does not check if the total liquidity in the ladder (`` `ladder_information.liquidity_lamports` ``) plus the new participation amount would exceed the ladder's initial cap (`` `LADDER_INITIAL_CAP = 2222 * LAMPORTS_PER_SOL` ``).
This oversight allows the ladder to accept contributions beyond its intended capacity.

Impact

- The contract may accumulate more funds than intended, violating the business logic of the ladder system
- If other parts of the system rely on the cap being enforced (e.g., for calculating rewards or determining ladder phases), those calculations could be incorrect
- Potential over-allocation of resources or rewards that were designed with the cap in mind
- Participants may join with incorrect expectations about the pool size and their relative position.

Remediation

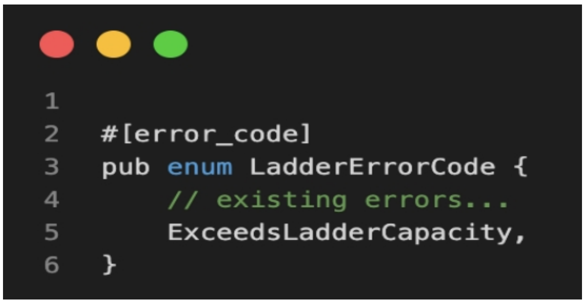
Add an explicit check to ensure that the total participation doesn’t exceed the ladder’s initial cap:



```rust

// Add this check after the existing validations
if ctx.accounts.ladder_information.liquidity_lamports + amount > LADDER_INITIAL_CAP {
    //Can implement remediation of 1st issue here as well, to make it less strict;
    msg!("Ladder has reached its capacity");
    return Err(LadderErrorCode::ExceedsLadderCapacity.into());
}
```

Additionally:
1. Add a new error code to the LadderErrorCode enum:



```rust

#[error_code]
pub enum LadderErrorCode {
    // existing errors...
    ExceedsLadderCapacity,
}
```

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 Aqua Bot. We performed our audit according to the procedure described above.

Issues of Low severity were found. Aqua Bot team resolved all the issues mentioned.

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 seven years of expertise, we've secured over 1400 projects globally, averting over $3 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.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2025

For

AQUA BOT

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com