Zellic

Audius

Smart Contract Security Assessment

October 11, 2022

Prepared for:

Raymond Jacobson and Roneil Rumburg

Tiki Labs Inc.

Prepared by:

Ayaz Mammadov and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Tiki Labs Inc. from September 19th to October 7th, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

We applaud Tiki Labs Inc. for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Audius.

Zellic thoroughly reviewed the Audius codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

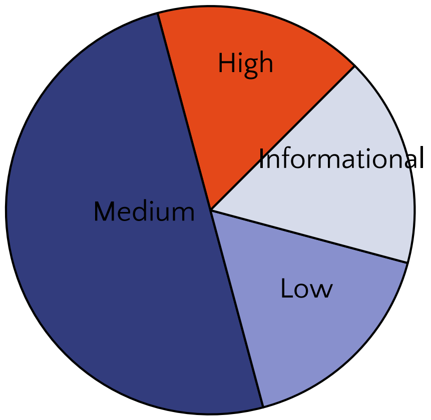
Specifically, taking into account Audius's threat model, we focused heavily on issues that would break core invariants such as the Governance proposals, both in terms of user interaction with them and in the way they execute. Moreover, we paid special attention to the way the protocol staking happens, such that all the contracts that are supposed to be able to stake can do so in a seamless manner and that the stakeholders are always secured.

During our assessment on the scoped Audius contracts, we discovered six findings. Of the six findings, one was of high severity, three of medium severity, one of low severity, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Tiki Labs Inc.'s benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	3
Low	1
Informational	1



2 Introduction

2.1 About Audius

Audius focuses on the sector of music streaming services and gives monetary power and ownership back to artists and creators. With the support of blockchain and decentralization, Audius represents a platform for streaming and sharing music that aims to change the way fans and music creators interact, making sure to protect music ownership and return it to creators.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Audius Contracts

Repository	https://github.com/AudiusProject/audius-protocol
Versions	`bda973856efc8b9144606d7952ea6a59e0d2bcca`
Programs	• AudiusAdminUpgradeabilityProxy.sol • WormholeClient.sol • erc20/AudiusToken.sol • TrustedNotifierManager.sol • ServiceTypeManager.sol • IWormhole.sol • DelegateManagerV2.sol • Migrations.sol • Governance.sol • DelegateManager.sol • GovernanceV2.sol • InitializableV2.sol • registry/Registry.sol • ServiceProviderFactory.sol • EthRewardsManager.sol • ClaimsManager.sol • Staking.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 19, 2022	Start of primary review period
October 7, 2022	End of primary review period

3 Detailed Findings

3.1 Voting can potentially be influenced via restaking

- **Target**: GovernanceV2
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

Currently, the magnitude of a vote is determined when it is submitted, based on the total stake of the user that submits the vote.

```
function submitVote(uint256 _proposalId, Vote _vote) external {
    // ...

    address voter = msg.sender;

    // ...

    // Require voter has non-zero total active stake
    uint256 voterActiveStake = _calculateAddressActiveStake(voter);
    require(
        voterActiveStake > 0,
        "Governance: Voter must be address with non-zero total active
    stake."
    );

    // Record vote
    proposals[_proposalId].votes[voter] = _vote;

    // Record voteMagnitude for voter
    proposals[_proposalId].voteMagnitudes[voter] = voterActiveStake;

    // ...
}


function _calculateAddressActiveStake(address _address)
    private view returns (uint256) {
```

```
    ServiceProviderFactory spFactory
    = ServiceProviderFactory(serviceProviderFactoryAddress);
    DelegateManager delegateManager
    = DelegateManager(delegateManagerAddress);

    // Amount directly staked by address, if any, in
    ServiceProviderFactory
    (uint256 directDeployerStake,,,,,)
    = spFactory.getServiceProviderDetails(_address);

    // Amount of pending decreasedStakeRequest for address, if any, in
    ServiceProviderFactory
    (uint256 lockedDeployerStake,)
    = spFactory.getPendingDecreaseStakeRequest(_address);
    // active deployer stake = (direct deployer stake - locked deployer
    stake)
    uint256 activeDeployerStake
    = directDeployerStake.sub(lockedDeployerStake);

    // Total amount delegated by address, if any, in DelegateManager
    uint256 totalDelegatorStake
    = delegateManager.getTotalDelegatorStake(_address);
    // Amount of pending undelegateRequest for address, if any, in
    DelegateManager
    (,uint256 lockedDelegatorStake, )
    = delegateManager.getPendingUndelegateRequest(_address);
    // active delegator stake = (total delegator stake - locked delegator
    stake)
    uint256 activeDelegatorStake
    = totalDelegatorStake.sub(lockedDelegatorStake);

    // activeStake = (activeDeployerStake + activeDelegatorStake)
    uint256 activeStake = activeDeployerStake.add(activeDelegatorStake);

    return activeStake;
}
```

Impact

As currently designed, there exists no checks on whether the staking/unstaking locking period is greater than the voting period. Imagine the following scenario:

1. User A votes "YES" on a proposal, then unstakes their share and transfers it to user B.

2. User B stakes, then votes "YES" on the same proposal, effectively pumping the voting weight.

3. The process could repeat over and over, as long as the staking/unstaking locking periods fit in the voting period of the proposal.

Recommendations

As discussed with the Audius team, we determined that currently the contracts are secure, since the staking lockup period is greater than the voting period. This means that despite the fact that theoretically the attack may be possible under specific circumstances (e.g., locking period of staking is way less than the voting period of proposal), it is impossible to perform it as per the current state of the contracts.

The fix, as proposed by the Audius team, would be to enforce that the unstake period is always greater than the voting period of a proposal.

Remediation

The issue has been addressed in pull request 4358.

3.2 Initialize check is missing from some functions

- **Target**: DelegateManager(V2), WormholeClient
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Medium
- **Impact**: Medium

Description

The `_requireIsInitialized` function is available in contracts that inherit the `InitializableV2` contract and is used to ensure that the child contract has been initialized before performing any other function call. Currently, the `cancelRemoveDelegatorRequest` in `DelegateManagerV2` and `DelegateManager` and `transferTokens` in `WormholeClient` miss this important check.

Impact

There are no direct security implications of these instances of omitting the `_requireIsInitialized` check; however, the functions that should implement it and currently do not would revert.

Recommendations

In order to keep a consistent code design and follow best practices over all the contracts and their functions, we recommend adding the `_requireIsInitialized` function call in the two functions mentioned above.

```
// DelegateManagerV2.sol, DelegateManager.sol

function cancelRemoveDelegatorRequest(address _serviceProvider,
    address _delegator) external {
    _requireIsInitialized();

    require(
        msg.sender == _serviceProvider || msg.sender == governanceAddress,
        ERROR_ONLY_SP_GOVERNANCE
    );
    require(
        removeDelegatorRequests[_serviceProvider][_delegator] ≠ 0,
        "DelegateManager: No pending request"
    );
    // Reset lockup expiry
    removeDelegatorRequests[_serviceProvider][_delegator] = 0;
    emit RemoveDelegatorRequestCancelled(_serviceProvider, _delegator);
}

// WormholeClient.sol

function transferTokens(
    address from,
    uint256 amount,
    uint16 recipientChain,
    bytes32 recipient,
    uint256 arbiterFee,
    uint deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public {

    _requireIsInitialized();
    // ...
```

Remediation

The issues have been addressed in pull request 4360.

3.3 Stake contract address should not change once set

- **Target**: Project-wide
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Medium
- **Impact**: Medium

Description

Currently, the address of the staking contract is stored in a variable called stakingAddress and can be set via the setStakingAddress function, an action that can only be performed by the governanceAddress. There is no check put in place, however, on whether the stakingAddress has been previously set or not.

Impact

Changing the staking address after users have already interacted with it may result in a significant confusion between the user and the contracts they are supposed to interact with. This is mainly because the accounts mapping, which stores the amounts staked by each user, would not reflect what the user has staked in the initial Staking contract.

Recommendations

We strongly recommend that once set, the stakingAddress should not be changeable.

```
function setStakingAddress(address _stakingAddress) external {
    _requireIsInitialized();

    require(stakingAddress == address(0), ERROR_STAKING_ALREADY_SET);
    require(msg.sender == governanceAddress, ERROR_ONLY_GOVERNANCE);
    stakingAddress = _stakingAddress;
    emit StakingAddressUpdated(_stakingAddress);
}
```

Remediation

The issues have been addressed in pull request 4362.

3.4 Unused allowance

- **Target**: ClaimsManager
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The `initiateRound` functions approves a transfer; however, this allowance is not used by the `safeTransfer`.

```
function initiateRound() external {
    ...
    audiusToken.mint(address(this), recurringCommunityFundingAmount);

    // Approve transfer to community pool address
    audiusToken.approve(communityPoolAddress,
    recurringCommunityFundingAmount);

    // Transfer to community pool address
    ERC20(address(audiusToken)).safeTransfer(communityPoolAddress,
    recurringCommunityFundingAmount);
    ...
```

Impact

This allows `communityPoolAddress` to receive twice the allotted claims from the `claimsManager`. Currently this does not pose an active security issue as `EthRewardsManager` is only managed by governance; however, if the `communityPoolAddress` changed, this could result in a more severe vulnerability.

Recommendations

Remove the approval, or use `safeTransferFrom` instead of `safeTransfer`.

Remediation

The issue has been addressed in pull request 4359.

3.5 Inconsistent usage of SafeMath

- **Target**: Project-wide
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Solidity version 0.5 does not have inbuilt overflow or underflow protections. As a consequence of this, SafeMath should be used in areas where overflow or underflow are not the intended behavior, such that the operations revert safely. As an example, underflow protection should be implemented in the function below:

```
function _removeFromInProgressProposals(uint256 _proposalId) internal {
     ...
    inProgressProposals[index]
    = inProgressProposals[inProgressProposals.length - 1];
    inProgressProposals.pop();
}
```

Impact

In the areas affected, we only noted reverts; however, future commits could change the behaviour of certain affected functions, leading to more severe vulnerabilities.

Recommendations

Use SafeMath wherever overflow is not intended behavior.

Remediation

The issue has been fixed in pull request 4361.

3.6 Governance should be transferred in two steps

- **Target**: Project-wide
- **Category**: Business Logic
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

The governance plays a fundamental role in the logic of all the contracts. It is important that whenever there might occur a transfer in the governance ownership, this transfer happens in two steps.

Impact

Currently, the contracts implement a simple transfer of ownership, which is prone to human error. For this reason, a wrong address may be introduced when using the `setGovernanceAddress`, thus losing the ownership of the contract.

```
function setGovernanceAddress(address _governanceAddress) external {
    _requireIsInitialized();

    require(msg.sender == governanceAddress, ERROR_ONLY_GOVERNANCE);
    _updateGovernanceAddress(_governanceAddress);
    emit GovernanceAddressUpdated(_governanceAddress);
}

function _updateGovernanceAddress(address _governanceAddress) private {
    require(
        Governance(_governanceAddress).isGovernanceAddress() == true,
        "ClaimsManager:  _governanceAddress is not a valid governance
    contract"
    );
    governanceAddress = _governanceAddress;
}
```

Recommendations

We recommend implementing a two-step governance ownership transfer function, such that any issues that may arise while transferring the governance will be mitigated.

```
function setGovernanceAddress(address _governanceAddress) external {
    _requireIsInitialized();

    require(msg.sender == governanceAddress, ERROR_ONLY_GOVERNANCE);
    toBeGovernanceAddress = _governanceAddress;
}

function acceptGovernanceRole() external {
    _requireIsInitialized();

    require(msg.sender == toBeGovernanceAddress, ERROR_ONLY_GOVERNANCE);
    _updateGovernanceAddress(toBeGovernanceAddress);
    emit GovernanceAddressUpdated(_governanceAddress);
}

function _updateGovernanceAddress(address _governanceAddress) private {
    require(
        Governance(_governanceAddress).isGovernanceAddress() == true,
        "ClaimsManager: _governanceAddress is not a valid governance
    contract"
    );
    governanceAddress = _governanceAddress;
}
```

Remediation

The issue has ben acknowledged by the Tiki Labs team. Upon further investigation, the team has decided not to mitigate it as of the time of writing this report.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Gas inefficiency can result in prohibitively expensive gas costs

There are several areas in the code that use linear array scans to find an element instead of a mapping. In Solidity, a storage read has a static gas cost, and as a consequence of this, linear array scans are O(n) and storage reads are very expensive.

As a result of the aforementioned, there are several edgecases in which it becomes prohibitively expensive to carry out certain operations. For example,

- calling `delegateStake` close to maxDelegators results in gas costs around 1.1 million gas, due to the `_DelegatorExistsForSP` check using arrays instead of mappings.
- calling `vetoProposal/evaluateProposalOutcome` close to maxInProgressProposals also becomes expensive due to the `_removeFromInProgressProposals` using arrays instead of mappings.
- calling `slash` in `DelegateManager/V2` on a service provider with the default maximum number of delegators (175) costs approximately 3.4 million gas.
- calling `submitProposal` in `GovernanceV2.sol` with close to maxInProgressProposals costs around 800 thousand gas.

5 Audit Results

At the time of our audit, the code was deployed to mainnet evm.

During our audit, we discovered six findings. Of these, one was of high risk, three of medium risk, one of low risk, and one was a suggestion (informational). Tiki Labs Inc. acknowledged all findings and implemented fixes.

5.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.