Audius Contracts Audit

OpenZeppelin Security - August 25, 2020

Introduction

The Audius team asked us to review and audit their smart contracts. We looked at the code and now publish our results.

We audited commit 6f3b31562b9d4c43cef91af0a011986a2580fba2 of the AudiusProject/audius-protocol/ repository. In scope are all the smart contracts in the `eth-contracts/contracts` directory, except for the `test` contracts.

All external code and contract dependencies were assumed to work correctly. Additionally, we assumed that the administrators are available, honest, and not compromised during this audit.

Update: *The Audius team made some fixes and comments based on our recommendations. They fixed all the critical, high, and medium severity issues that we reported. Below we address those fixes, which were introduced in pull requests #564 and #657. The Audius team then merged these pull requests into master, resulting in commit dac9cb31f0a2df9c25083bd3a833d285a4d947ef, which now includes the smart contract code we originally audited and the fixes that we reviewed as part of this engagement. Our analysis of the mitigations disregards any other changes to the code base.*

About Audius

Audius is a decentralized community of artists, developers, and listeners whose mission is to give everyone the freedom to share, monetize, and listen to any audio.

The system comprises service providers who maintain the availability of the content, index the content for discovery, handle authentication, monitor activity, and cache.

This system is backed by these main smart contracts deployed in the Ethereum blockchain:

- Registry: hub where all the contracts are registered.
- ServiceTypeManager: maintains the different types of service providers, their versioning, and stake requirements.
- ServiceProviderFactory: manages the registration of service endpoints and their delegate owner wallet.
- AudiusToken: ERC-20 token used for staking on service providers and rewards.
- Staking: stores staking balances for service providers.
- DelegateManager: manages delegation of stake to service providers, slashing and claiming rewards.
- ClaimFactory: mints and allocates tokens, and manages claim rounds.
- Governance: manages changes to the protocol through staked voting.

All the Audius contracts use the OpenZeppelin proxy pattern and can be upgraded through the Governance contract by voting on proposals submitted by stakers or through administrator action.

The system administrators manage the registry, which lists all the target contracts that can be proposed for Governance voting. In the Governance contract migration, the registry ownership is transferred to the Governance contract, so all contracts added to the registry are intended to go through the governance system. In this audit we assumed that contracts are extensively tested and audited before being added to the registry.

The administrators also control the `guardianAddress`, which can veto proposals made by the stakers, and directly execute any protocol changes without voting. According to the Audius team, "These safeguards will be slowly removed over time through contract upgrades to Governance itself." With pull request #616, the Audius team also allowed the `guardianAddress` to submit proposals without stake.

***Update**: While we were auditing this project, the Audius team found the following issue:*

The guardian can veto a proposal at any time from its creation until its evaluation.

However, if the `guardianAddress` waits to veto a proposal until the voting period is almost over, by making use of the `vetoProposal` function, this transaction can be frontrun by any staker calling the `evaluateProposalOutcome` function, which will modify the proposal's outcome and make the `vetoProposal` transaction fail when the requirement for an `InProgress` proposal is not met.

Also, depending on the vote count, the guardian could be tricked to not veto the proposal because the proposal does not reach the necessary votes to be executed. However, a malicious staker could vote near

the voting deadline, changing the proposal from a failure into a success, and executing it while it frontruns the `vetoProposal` call.

Consider adding a cooldown period after the voting ends to allow the guardian to veto successful but malicious proposals.

Update: *Fixed. An* `executionDelay` *is now enforced on every proposal to account for this vulnerability.*

Critical Severity

[C01] A malicious delegator can permanently lock all stake and rewards for a victim service provider and all of its honest delegators

The `DelegateManager.requestUndelegateStake` function increases the value of the `spDelegateInfo[_target].totalLockedUpStake` variable. However, if this request is cancelled via the `cancelUndelegateStake` function, the `spDelegateInfo[_target].totalLockedUpStake` variable is not decreased.

This means that a malicious delegator can delegate to a target service provider, and then call `requestUndelegateStake` and `cancelUndelegateStake` repeatedly, causing `spDelegateInfo[_target].totalLockedUpStake` to grow arbitrarily large.

If the attacker makes `spDelegateInfo[_target].totalLockedUpStake` larger than `totalBalanceOutsideStaking`, then the `claimRewards` function will always revert on line 364 of `DelegateManager.sol` when called by the victim service provider. In this way, the malicious delegator can permanently prevent the service provider from ever claiming any of their rewards.

This has additional negative security consequences.

First, since the victim service provider will not be able to claim their pending rewards, the `_claimPending` function will always return `true` after the end of the week in which the attack took place. This means that honest delegators who have delegated to the victim service provider will never be able to undelegate their stake because their calls to the `undelegateStake` function will revert on line 252.

Second, this also means that the victim service provider cannot successfully call the `ServiceProviderFactory.requestDecreaseStake` function because it will revert on line 369 of `ServiceProviderFactory.sol`. So the victim service provider also has their stake permanently locked.

Consider modifying the `cancelUndelegateStake` function so that it reduces the `spDelegateInfo[_target].totalLockedUpStake` variable by the pending `UndelegateStakeRequest` amount.

Update: *Fixed in pull request #561.*

[C02] Updating or removing a service type causes critical accounting errors

Updating or removing a service type after a service provider has registered an endpoint of that type can result in critical accounting errors. These errors can have serious effects, including preventing service providers from deregistering endpoints and preventing delegators from undelegating their stake.

When a service provider registers or deregisters an endpoint, the `ServiceProviderFactory` contract tracks the required bounds within which a service provider's stake must remain. These bounds are tracked in a `ServiceProviderDetails` **struct** in the `ServiceProviderFactory` contract, and are referred to as the `minAccountStake` and `maxAccountStake`. These bounds are determined by the `minStake` and `maxStake` of the service type's `ServiceTypeStakeRequirements` **struct**, which is tracked in the `ServiceTypeManager` contract.

However, when governance updates the `minStake` or `maxStake` of an endpoint — either via the `updateServiceType` **function** or the `removeServiceType` **function** — the `minAccountStake` and `maxAccountStake` for a given service provider that has already registered an endpoint of that type are *not* automatically updated.

This can result in critical accounting errors. Here are two examples.

Example 1. Increasing a service type's `minStake` or `maxStake` can result in a service provider being unable to deregister an endpoint:

Suppose a service provider registers exactly one endpoint. Then suppose governance increases either the `minStake` or `maxStake` for that service type. Then when the service provider attempts to deregister their endpoint, their call to the `deregister` function will revert **on line 287** or **on line 288** due to an underflow in `.sub`.

Example 2. Decreasing a service type's `minStake` can result in a delegator being unable to undelegate all of their stake:

Suppose a service provider registers an endpoint, and suppose a delegator delegates to that service provider. Then suppose governance decreases the `minStake` for that service type — either via the `updateServiceType` **function** or the `removeServiceType` **function**. Then when the service provider deregisters the endpoint, the `deregister` function sets the `minAccountStake` to a value greater than `0` **on line 287**. This means that when the delegator tries to undelegate all of their stake, their call to the `undelegateStake` function will revert **on line 311**, because the `validateAccountStakeBalance` function **calls the** `_validateBalanceInternal` **function**, **which requires** that `amount` (which will be `0`) is at least `spDetails[_sp].minAccountStake` (which will be greater than `0`).

Consider removing the ability for governance to upgrade or remove service types. Instead, consider indicating the version of the service type in the service type's name (e.g.: `"ServiceType1-v1"`) and using a `bool` to flag whether the service type is still "active" and can be registered by service providers.

***Update**: Fixed in pull request #555. The ability to update an existing service type was removed.*

[C03] Proxy admin doesn't entirely cede upgrade control to governance address

In the `AudiusAdminUpgradeabilityProxy` contract, the inline documentation states that the proxy admin cedes upgrades control to the governance address, which is achieved by overloading the `upgradeTo` function of the parent contracts.
However, the contract does not overload the `upgradeToAndCall` function from the OpenZeppelin's `BaseAdminUpgradeabilityProxy` contract, allowing the deployer admin to bypass the governance and control the upgrades.

This is dangerous, as it gives the admin's private key holder the possibility to upgrade the whole protocol without a proper governance's consent. Furthermore, if the private key gets leaked after the deployment, it will allow any third party to modify the Audius bytecode on-chain.

Consider modifying the current implementation to discard the upgrade permissions given to the admin address. Also, consider inheriting from a lower-level abstraction rather than overwriting existing functionality from a higher-level one. Finally, consider validating that the implementation contract does not have a signature clash with the proxy.

***Update**: Fixed in pull request #657.*

High Severity

[H01] A malicious delegator can prevent all other delegators from delegating

All the delegators for a given service provider are stored in an array. These arrays are iterated over during various operations. In order to prevent these iterations from hitting the block gas limit, a `maxDelegators` value is set. Once a service provider gains this maximum number of delegators, no new delegators can delegate to that service provider. By default `maxDelegators = 175`.

Additionally, the code requires that the total number of tokens (over all service providers) delegated by an active delegator must be above some `minDelegationAmount`. This `minDelegationAmount` is not enforced *per service provider*, but over all service providers to which the delegator has delegated. Therefore it is possible for a delegator to delegate to `X` service providers using a total of `minDelegationAmount + 1e-18 * (X-1)` tokens — by delegating `minDelegationAmount` to the first service provider and delegating just `1e-18` tokens to the remaining `X-1` service providers.

As a result of these two facts, an attacker can fill up all the delegator slots for a given service provider, thereby preventing any honest delegators from delegating to that service provider. They can do this for all service providers on the platform, thereby preventing any delegators from delegating.

They can do this as follows: Suppose there are `N` service providers on the platform. The attacker can create `maxDelegators` many ethereum accounts, depositing `minDelegationAmount + 1e-18 * (N - 1)` tokens in each of them. Then, with each account, the attacker delegates `minDelegationAmount` tokens to one of the `N` service providers and `1e-18` tokens to the other

`(N−1)` service providers. As a result, all delegator slots for all service providers will be filled, and no honest delegators will be able to delegate to any service provider.

This attack costs `maxDelegators * [minDelegationAmount + 1e−18 * (N − 1)]`, which is approximately 17,500 AUD (using default values and ignoring dust amounts). To put this in perspective, this attack costs about `1.3%` of one week's worth of AUD rewards.

Consider having the `minDelegationAmount` be applied per service provider, rather than over all service providers. This would prevent a service provider's delegator slots from being filled by "dust delegators".

***Update**: Fixed in pull request #552.*

[H02] Delegators can prevent service providers from deregistering endpoints

Under some conditions, delegators may prevent service providers from deregistering endpoints. This can happen innocently or maliciously.

Consider the case where a service provider has registered more than one endpoint and that the service provider has staked the minimum amount of stake. Suppose delegators have delegated to this service provider the maximum amount of stake.

When the service provider attempts to deregister one of the endpoints, their call to the `deregister` function may fail on line 300. This is because the `_validateBalanceInternal` function requires that `_amount <= spDetails[_sp].maxAccountStake`, which may not be true if enough stake has been delegated to the service provider.

Consider adjusting the logic of the `deregister` function to handle the case where `Staking(stakingAddress).totalStakedFor(_sp) > maxAccountStake`.

***Update:** Partially fixed in pull request #570. The fix may introduce new attack vectors to the codebase. For example, in the `removeDelegator` function the stake of the delegator is unstaked instantly. This could allow a service provider to bypass a slashing process by using sybil delegators and then removing its delegated stake before the slash takes place. This will reduce the slashing punishment significantly. Consider modifying the `removeDelegator` function to set a timelock so the slashing mechanism cannot be bypassed in that way. Furthermore, allowing a service provider to call this function at anytime could introduce further incentive problems. To mitigate this, consider adding a requirement on the `removeDelegator` function so it can only be called if the sum of stakes for a service provider is bigger than `spDetails[_sp].maxAccountStake`.*

***Update:** Fixed in pull request #657. The `removeDelegator` function now implements a timelock.*

[H03] Updating the Governance registry and Guardian addresses emits no events

In the `Governance` contract the `registryAddress` and the `guardianAddress` are highly sensitive accounts. The first one holds the contracts that can be proposal targets, and the second one is a superuser account that can execute proposals without voting.

These variables can be updated by calling `setRegistryAddress` and `transferGuardianship`, respectively. Note that these two functions update these sensitive addresses without logging any events. Stakers who monitor the Audius system would have to inspect all transactions to notice that one address they trust is replaced with an untrusted one.

Consider emitting events when these addresses are updated. This will be more transparent, and it will make it easier for clients to subscribe to the events when they want to keep track of the status of the system.

Update: Fixed in pull request #563.

[H04] No incentive for evaluating proposals with outcome other than Yes

After a voting period has ended, the `evaluateProposalOutcome` function of the `Governance` contract can be called to try to execute the target contract for proposals with a Yes quorum and to update the proposal state.

There is an incentive for approved proposals to be executed by their proposers or supporters. However, when a proposal does not reach the quorum (or is rejected), this function spends gas to update the state. It is unclear why a user would pay for this gas to clean up the proposals state. Since anybody can submit a proposal at any time, this could lead to many closed proposals with an outdated `InProgress` state. This could be confusing to voting interfaces which will have to inspect the proposal to check if they are actually open.

Consider adding an incentive for the caller of the `evaluateProposalOutcome` function, so there are better guarantees that the state of the proposals will be up-to-date.

Update: *Fixed in pull requests #575 and #609. Now, before submitting a new proposal, the status of all the proposals that can be evaluated have to be up-to-date. Note that this could make it too expensive for somebody to send new proposals if they have to evaluate many old proposals. In this case, again, only the administrators might be incentivized to evaluate all the proposals in order to unblock the system. Also note that setting the maximum number of in-progress proposals emits no event.*

[H05] Rewards calculation is incorrect when the service provider has a pending "decrease stake" request

When the service provider has no `DecreaseStakeRequest` pending, the `claimRewards` function from the `DelegateManager` contract seems to work as intended. But when the service provider has a `DecreaseStakeRequest` pending, the `claimRewards` function computes the `totalRewards` value incorrectly (setting it too high). In particular, the `claimRewards` function sets the value of the `totalRewards` variable equal to the total amount of rewards that are being paid out *plus* the amount of any `DecreaseStakeRequest` the service provider has pending.

This makes the `ServiceProviderFactory` and the `DelegateManager` contracts record incorrect (too high) values for the `spDetails[_serviceProvider].deployerStake` and `delegateInfo[delegator][_serviceProvider]`, respectively. As a result, the staking contract may become insolvent, owing more tokens than it holds.

To see that the `DelegateManager.claimRewards()` function computes `totalRewards` incorrectly, first consider this code block. The `totalRewards` variable is computed using these variables, so we begin here.

Looking at the `_validateClaimRewards` function, one can see that it first processes the claim, which mints into existence any rewards due, and stakes them for the service provider.

Then, the `_totalBalanceInStaking`, `_totalBalanceInSPFactory`, and `_totalBalanceOutsideStaking` variables are computed as follows:

The `_totalBalanceInStaking` variable gets set to `Staking(stakingAddress).totalStakedFor(_serviceProvider)`, which, in this context, is equal to:

```
_totalBalanceInStaking =
 the amount of locked SP stake from before claimRewards was called
 + the amount of unlocked SP stake from before claimRewards was called
 + the amount of delegator stake from before claimRewards was called
 + any rewards just paid out by ClaimsManager.processClaim
```

Next, `_totalBalanceInSPFactory` is initially set to the total amount of stake (locked + unlocked) that the service provider has staked. Then on line 727 it is updated by subtracting away the amount of the service provider's locked stake. So `_totalBalanceInSPFactory` ends up being:

```
_totalBalanceInSPFactory =
 the amount of unlocked SP stake from before claimRewards was called
```

Finally, `_totalBalanceOutsideStaking` gets set to `_totalBalanceInSPFactory.add(spDelegateInfo[_serviceProvider].totalDelegatedStake)`. So we have:

```
_totalBalanceOutsideStaking =
 the amount of unlocked SP stake from before claimRewards was called
 + the amount of delegator stake from before claimRewards was called
```

The `totalRewards` variable is computed as `totalBalanceInStaking.sub(totalBalanceOutsideStaking)`, which, by substituting the above values and simplifying, is equal to:

```
totalRewards =
 the amount of locked SP stake from before claimRewards was called
 + any rewards just paid out by ClaimsManager.processClaim
```

So if the service provider has any locked stake, then `totalRewards` will not accurately represent the amount of rewards that were paid out during the call to `processClaim`.

Consider adjusting the `processClaim` function so that it directly returns the amount of rewards that it paid out. Then the `totalRewards` variable can be set directly equal to the `processClaim` function's return value instead of being computed indirectly. This has the additional benefit of reducing gas costs.

***Update**: Fixed in pull request #562. The Audius team noticed that, although there are two incorrect balance calculations when a stake decrease is pending, these two errors cancel each other out. The final*

comparison between what was minted and the internal record is correct. Consequently, we downgraded the severity of this issue to high. Now the `processClaim` *function from the* `ClaimManager` *contract returns the minted rewards for the service provider directly. Note that the calculation is still complex and could be refactored for clarity. Audius' statement for this issue:*

> Summary: During rewards calculation, a pending decrease stake request results in a value for totalRewards that is incorrect – by the amount of requested decrease in stake. Further analysis into the issue exposed an even more interesting behavior, confirming that value for totalRewards is incorrect but also showing that the final value set for a claimer in ServiceProviderFactory is not higher than expected. This is because our base value for the new Service Provider stake is also skewed but in the opposite direction during calculation of the new Service Provider stake.
>
> For this reason we would like to consider re-classifying the issue from Critical to High – rewards are neither minted incorrectly nor distributed incorrectly, but tracked incorrectly during the process of claiming.
>
> Interestingly enough, we already have a test case to cover reward distribution when a decrease stake request is pending. However this was passing due to the described condition above.

[H06] AUD lending market could affect the protocol

In case an AUD token lending market appears, an attacker could use this market to influence the result of a governance's proposal, which could lead to a take over of the protocol.

An attacker would only need to stake tokens for a brief moment without waiting for the `votingPeriod` to request an unstake. This aggravates the attack, as the attacker would only need to take a loan for the number of blocks established by the `decreaseStakeLockupDuration` variable.

The only prerequisite that an attacker needs for this attack is to have sufficient collateral, which could be trivial if a lending market of AUD tokens exists while AUD price is still low enough.

Consider countermeasures for these type of attacks, and have plan for how to react when a lending market for AUD is created.

***Update**: Fixed. As described in the updates of "[H08] Endpoint registration can be frontrun" and "[H09] Slash process can be bypassed",* `decreaseStakeLockupDuration` *is already significantly larger than* `votingPeriod` *+* `executionDelay`*. Audius's statement about this issue:*

> The above is no longer possible with our enforced relationship between decreaseStakeLockup and votingPeriod + delay. An attacker may still stake tokens immediately prior to a proposal, but the relationship between the two variables means they

are still subject to a slash operation. This is because an attacker cannot unstake without waiting at at least one votingPeriod + executionDelay time difference.

[H07] The quorum requirement can be trivially bypassed with sybil accounts

While the final vote on a proposal is determined via a token-weighted vote, the quorum check in the `evaluateProposalOutcome` function can be trivially bypassed by splitting one's tokens over multiple accounts and voting with each of the accounts. Each of these sybil votes increases the `proposals[_proposalId].numVotes` variable. This means anyone can make the quorum check pass.

Consider measuring quorum size by the percentage of existing tokens that have voted, rather than the number of unique accounts that have voted.

Update: *Fixed in pull request #574. However, note that the* `setVotingQuorumPercent` *function is not validating that the value set is between 0 and 100.*

[H08] Endpoint registration can be frontrun

An honest service provider's call to the `ServiceProviderFactory.register` function can be frontrun by a malicious actor in order to prevent any honest user from being able to register any endpoint.

The attacker can monitor the mempool for any calls to the `register` function, then frontrun them with their own call to the `register` function using the same `_endpoint` parameter.

This registers the endpoint under the attacker's account so that the honest user's attempt to register their endpoint will fail on line 163.

There is a cost to this attack. In particular, the attacker must stake the `minStake` or else line 199 will revert. This stake may be at risk of being slashed until the attacker has deregistered the endpoint and removed their stake. Since it takes at least ten blocks for an attacker to remove their stake after deregistering an endpoint, there is a window of opportunity for governance to slash the attacker. However, given the nature of the attack, it is not clear that it could be detected and punished within the ten blocks (about 2.5 minutes) lockup duration.

If `minStake` is small enough and/or the probability of getting detected and slashed is low enough, then this attack would have a low expected cost. Since these are currently unknowns, it is conservative to classify this issue as high severity.

To prevent a malicious service provider to register another service provider's endpoint first, consider hashing the endpoint and the service provider's address (`msg.sender`) together to create the endpoint's `bytes32` identifier and then use it in a commit/reveal scheme during the registration process.

Update*: Fixed in pull request # 573, where the lockup period was changed from 10 blocks to 1 week. While the registration process may still be frontrun, there will be enough time for such behavior to be detected and punished via slashing.*

[H09] Slash process can be bypassed

There are two ways for any address to be slashed. The first one is by a governance's proposal, and the second one is by a transaction performed by the guardian.

For governance to decide to slash a service provider, a proposal must be submitted to the contract, stakers must vote on it and achieve a majority, and then it has to be executed. This process takes several blocks to complete.

The `votingPeriod` establishes how long a governance proposal is open for voting.

Similarly, the `decreaseStakeLockupDuration` variable establishes the minimum length of time a service provider must wait before removing their stake.

If `decreaseStakeLockupDuration` is less than or equal to the `votingPeriod`, it will be possible for a malicious service provider to remove their stake before it can be slashed by the Governance protocol.

Since the guardian is expected to be removed once the system is fully operational — meaning that slashing a malicious service provider using the guardian account will not be possible — consider setting the `decreaseStakeLockupDuration` so it is much greater than the `votingPeriod`. This would ensure that a malicious service provider can always be slashed via governance.

***Update**: Fixed in pull request #657. The `_updateDecreaseStakeLockupDuration` function enforces that the `decreaseStakeLockupDuration` value is greater than the voting period plus an execution delay.*

[H10] A service provider can deceive its delegators

Service providers can earn AUD tokens by allowing delegators to delegate their stake into its account. The `deployerCut` value establishes the percentage of the rewards that the service provider collects from the delegators' rewards after each round.

A service provider can modify the `deployerCut` variable by calling the `updateServiceProviderCut` function at any time. It is also possible for them to even set it as 100, which would mean that delegators will not get rewards after the funding round has been completed.

If a service provider makes use of this function just before executing the `claimRewards` function from the `DelegateManager` contract, delegators will receive fewer or no rewards from that round, contrary to what they initially expected.

Consider requiring that a change to the `deployerCut` variable undergoes a timelock for a period of time greater than the `fundingRoundBlockDiff` value. This would allow delegators to have enough time to move their stake to another service provider.

***Update**: Fixed in pull request #657. Adjusting the `deployerCut` value now requires waiting for a timelock.*

[H11] A service provider can prevent their delegators from undelegating their stake

A service provider can prevent their delegators from undelegating their stake. This may happen maliciously or unintentionally, as follows.

Suppose a service provider has registered one or more endpoints and has staked the minimum amount of required stake. Then suppose one or more delegators have collectively staked an additional `X` tokens for this service provider, where `X <= spDetails[_sp].minAccountStake - minDeployerStake`, so that `totalStakedFor(_sp) = spDetails[_sp].minAccountStake + X`.

Next, consider what happens if the service provider decreases its stake by `X`:

The service provider's call to the `requestDecreaseStake` function will succeed, because the call to the `_validateBalanceInternal` function will not revert (all three `require` statements will be satisfied).

Their subsequent call to the `decreaseStake` function will succeed for the same reason.

At this point, `totalStakedFor(_sp) = spDetails[_sp].minAccountStake`. This means that any attempt by a delegator to undelegate a positive number of tokens via the `undelegateStake` function will revert on line 311, because the first `require` statement in the `_validateBalanceInternal` function will not be satisfied.

Consider requiring that `spDetails[_sp].deployerStake >= spDetails[_sp].minAccountStake` when validating balances. This would put the burden of maintaining the `minAccountStake` on the service provider, thus removing this vulnerability.

***Update**: Partially fixed in pull request #577. Now the minimum stake for a service provider must come from the service provider itself instead of using the delegators' stake. Nevertheless, if governance decides to slash a service provider and its staked balance ends up between `0 < SPBalance < spDetails[_sp].minAccountStake`, the delegators for that service provider will not be able to undelegate their stake. That way, the malicious service provider could stake only `spDetails[_sp].minAccountStake`, handle a bigger delegated stake value, and perform a malicious action that will be slashed to prevent delegators from undelegating when the new requirement reverts.*

***Update**: Fixed in pull request #657. The `undelegateStake` function no longer calls the `validateAccountStakeBalance` function.*

[H12] Unresponsive service provider locks delegator stake

A service provider may become unresponsive (e.g., by losing their keys, dying, etc.). If this happens, the service provider will not call the `claimRewards` function. After one week the `_claimPending` function will always return `true`. So delegators will not be able to undelegate their stake because the `undelegateStake` function will revert on line 252.

Consider refactoring the `claimRewards` function to allow anyone can call it — passing in a service provider address as a parameter. This way, the rewards-claiming process can be moved forward by anyone, not just the service provider. This would protect delegators from an unresponsive service provider.

***Update:** Fixed in pull request #556.*

Medium Severity

[M01] Complicated state updates

When stake balances are modified (through `delegateStake`, `requestUndelegateStake`, `cancelUndelegateStake`, `undelegateStake`, and `slash`), multiple operations are executed to increase or decrease the values of the state variables related to the updated stake status. This is error prone, as shown by the critical issue *"A malicious delegator can permanently lock all stake and rewards for a victim service provider and all of its honest delegators"* where one of the values was not correctly updated.

A similar pattern is implemented to track the number of votes for `Governance` proposals.

Consider encapsulating these operations into separate functions, one for each type of state update. This way it will be clearer to review that the operations are complete, consistent, and complementary. Some duplication can be removed, and these functions can be thoroughly tested in isolation.

Consider formal verification to prove that these critical state variables will always behave as expected and keep the system in a consistent state.

Update: *Fixed in pull request #539. Most of the logic was encapsulated in new internal functions, such as the* `_updateDelegatorStake` *and the* `_updateServiceProviderLockupAmount` *functions of the* `DelegateManager` *contract, and the* `_decreaseVoteMagnitudeNo` *and the* `_increaseVoteMagnitudeYes` *functions of the* `Governance` *contract.*

[M02] Inconsistently checking initialization

When a contract is initialized, its `isInitialized` state variable is set to `true`. Since interacting with uninitialized contracts would cause problems, the `_requireIsInitialized` function is available to make this check.

However, this check is not used consistently. For example, it is used in the `getVotingQuorum` function of the `Governance` contract, but it is not used in the `getRegistryAddress` function of the same contract. There is no obvious difference between the functions to explain this difference, and it could be misleading and cause uninitialized contracts to be called.

Consider calling `_requireIsInitialized` consistently in all the functions of the `InitializableV2` contracts. If there is a reason to not call it in some functions, consider documenting it. Alternatively, consider removing this check altogether and preparing a good deployment script that will ensure that all contracts are initialized in the same transaction that they are deployed. In this alternative, it would be required to check that contracts resulting from new proposals are also initialized before they are put in production.

Update: *Fixed in pull requests #587 and #594. The* `_requireIsInitialized` *check has been added to all the externally accessed functions of contracts that inherit from* `InitializableV2`.

[M03] Lack of event emission after sensitive changes

In several parts of the code there are sensitive functions that lack event emissions. This can make it difficult for users to be aware of important changes that take place.

Here are some examples:

- The `processClaim` function in the `ClaimsManager` contract does not emit an event in the case where there are no rewards.
- The `setGovernanceAddress`, `setStakingAddress`, `setServiceProviderFactoryAddress`, `setDelegateManagerAddress`, `updateFundingAmount`, and `updateFundingRoundBlockDiff` functions from the `ClaimsManager` contract do not emit events.
- The `addServiceType` and `removeServiceType` functions from the `ServiceTypeManager` contract do not emit events.
- The `updateDelegateOwnerWallet`, `updateEndpoint`, `updateServiceProviderCut`, and `updateDecreaseStakeLockupDuration` functions from the `ServiceProviderManager` contract do not emit events.

Consider adding events in these cases to make it easier to track important contract changes.

Update: *Fixed in pull request #583.*

[M04] Lack of input validation

The `upgradeContract` function in the `Registry` contract does not check if the `_newAddress` is the zero address. If passed the zero address, the function will perform the same functionality as the `removeContract` function but emit a `ContractUpgraded` event.

The `setServiceVersion` function in the `ServiceTypeManager` contract does not check if the inputted `_serviceType` type exists, allowing callers to add versions to a nonexistent service type.

The `updateServiceType` function in the `ServiceTypeManager` contract does not check if the new `_serviceTypeMax` is zero or not (which would make the `serviceTypeIsValid` function always return `false`, even for valid service types), or if `_serviceTypeMax < _serviceTypeMin` (which would break endpoint registration and deregistration).

The `setGovernanceAddress` function in the `ServiceTypeManager` contract does not check if the new `_governanceAddress` corresponds to a real governance address (for example, by calling an `isGovernanceAddress` function on the `_governanceAddress` contract). If the `_governanceAddress` is set to an incorrect address, control over the contracts may be permanently lost.

The `addServiceType` function in the `ServiceTypeManager` contract does not enforce that `_serviceTypeMax > 0`. This means it is possible to add the same service type to the `validServiceTypes` array multiple times.

The `setVotingPeriod` function in the `Governance` contract does not check if the `_votingPeriod` is zero. If it is, then it would not be possible to vote on any future proposal.

Consider adding input checks to each of these functions to reduce possible errors.

Update: *Fixed in pull request #569. Checks were added for each of the reported occurrences. For the one found in the* `addServiceType` *function, the* `ServiceTypeManager` *contract does not explicitly enforce that* `_serviceTypeMax` *has to be bigger than zero, nevertheless, it needs to be bigger than the* `_serviceTypeMin` *variable, which cannot be inferior than zero.*

[M05] Only active stakers can evaluate proposals

When a vote is complete, an account with active stake at the time the proposal was submitted has to call the `evaluateProposalOutcome` function of the `Governance` contract.

This function requires nothing specific from a staker, so it is not clear why this caller limitation is implemented. If no staker is interested in evaluating the proposal, then this prevents other potentially interested accounts to do the evaluation.

Consider allowing any account to evaluate proposals. Alternatively, if there is a reason why this action has to be limited to stakers, consider documenting it.

Update: *Fixed in pull requests #572 and #585. Now any account can evaluate proposals.*

[M06] Voting period and quorum can be set to zero

When the `Governance` contract is initialized, the values of `votingPeriod` and `votingQuorum` are checked to make sure that they are greater than 0. However, the corresponding setter functions `setVotingPeriod` and `setVotingQuorum` allow these to variables to be reset to 0.

Setting the `votingPeriod` to zero would cause spurious proposals that cannot be voted. Setting the `quorum` to zero is worse because it would allow proposals with 0 votes to be executed.

Consider adding the validation to the setter functions.

Update: *Fixed in pull request #568.*

[M07] Semantic overloading in the No outcome of proposals

When a proposal fails to meet support of the majority of stake, its outcome is set to `No`. Also, when a proposal is vetoed by the guardian account, its outcome is set to `No`.

This is a semantic overload, giving the `No` outcome two different meanings. It could be confusing for callers of these contracts and may open the door to regressions in future updates to the code.

Consider adding an extra `Veto` outcome, to clearly differentiate between the two states.

Update: *Fixed in pull request #579.*

[M08] Service providers and delegators can mistakenly soft-lock their own stake

If a service provider calls the `requestDecreaseStake` function and passes in a `_decreaseStakeAmount` of `0`, the call will succeed and a `DecreaseStakeRequest` will be queued up.

If the service provider then attempts to cancel that request using the `cancelDecreaseStakeRequest` function, the call will revert at line 402 because the `_decreaseRequestIsPending` function returns `false` when `decreaseStakeRequests[_serviceProvider].decreaseAmount = 0`.

If the service provider tries to call the `decreaseStake` function, the call will revert at line 420 for the same reason.

To get out of this "soft-lock" situation, the service provider must call the `requestDecreaseStake` function again, passing in a `_decreaseStakeAmount` greater than `0`, and then cancel that new `DecreaseStakeRequest`. This is outside the normal UX flow and may not be easy to discover.

Similarly, if a delegator calls the `requestUndelegateStake` function and passes in an `_amount` of `0`, the call will succeed and a `UndelegateStakeRequest` will be queued up.

If the delegator then attempts to cancel that request using the `cancelUndelegateStake` function, the call will revert at line 226 because the `_undelegateRequestIsPending` function returns `false` when `undelegateRequests[_delegator].amount == 0`.

If the delegator tries to call the `undelegateStake` function, the call will revert at line 244 for the same reason.

To get out of this "soft-lock" situation, the delegator must call the `requestUndelegateStake` function again, passing in an `_amount` greater than `0`, and then cancel that new `UndelegateStakeRequest`. As before, this is outside the normal UX flow and may not be easy to discover.

Consider requiring that the `requestDecreaseStake` only accepts a `_decreaseStakeAmount` greater than zero, and consider requiring that the `requestUndelegateStake` function only accepts an `_amount` greater than `0`.

Update: *Fixed in pull request #567.*

[M09] Ties in governance proposals are approved

When the stake in favor a governance proposal is equal to the stake against it, the proposal is considered approved and can be executed. In cases when critical updates to the system are so contentious to the point of reaching a tie, the safest course of action is to prepare a new proposal that can be discussed further and be more easily approved later.

Consider executing proposals only when they have a majority of supporting votes.

Update: *Fixed in pull request #576.*

[M10] Some state variables are not set during initialize

The Audius contracts can be upgraded using the unstructured storage proxy pattern. This pattern requires the use of an initializer instead of the constructor to set the initial values of the state variables. In some of the contracts, the initializer is not initializing all of the state variables.

For example, in the `initializer` function of the `ClaimsManager` contract the `stakingAddress` state variable is not set. It can be set later by calling `setStakingAddress`. However, this means that it is possible to call the `processClaim` function and some others without any staking address set, because they only check that the contract has been initialized.

The same happens in the `initializer` function of the `Governance` contract, which does not set the `stakingAddress` either.

Consider setting all the required variables in the initializer. If there is a reason for leaving them uninitialized consider documenting it, and adding checks on the functions that use those variables to ensure that they are not called before initialization.

Update: *Fixed in pull request #589.*

Low Severity

[L01] Governance proposal description only stored in log

When a new governance proposal is submitted, it requires a description. This description is not saved to the contract storage, only to the events log when `ProposalSubmitted` is `emitted`.

This makes it hard for a user of the contract to get the description. They would have to gather this information from the logs, which are not accessible to other on-chain contracts.

Consider saving the description in the contract storage, together with the rest of the proposal information.

Update: *Fixed in pull request #550.*

[L02] Duplicated `newProposalId` calculation

In lines 187 and 213 of `Governance.sol`, the same addition is executed twice.

To avoid duplication, consider replacing the second calculation with the value of `newProposalId`.

Update: *Fixed in pull request #544.*

[L03] Duplicated staker check

In lines 236 and 340 of `Governance.sol`, the same check for active stake at a particular block number is executed.

To avoid duplication, and to improve encapsulation and readability, consider implementing a function `wasStakerAt` in the `Staking` contract, and calling this function in the `Governance` contract

instead of reimplementing the check.

In the same `Governance` contract, there is another check to see if the account currently has some stake. For readability and consistency with the previous suggestion, consider also adding a function `isCurrentStaker` the `Staking` contract.

Update: Partially fixed in pull request #580. The `isStaker` *function was introduced to the* `Staking` *contract. The check for active stake at a block number was not refactored. Audius' statement for this issue:*

> After evaluation, the suggested function `wasStakerAt` may offer a slight improvement in readability but sacrifice other benefits. For example, if `wasStakerAt` is a function returning a `bool` for a specific block:
>
> `wasStakerAt(spAddress, blockId) returns (bool)`
>
> In order to replace current usage of `totalStakedForAt` per audit feedback, another call would have to be made to retrieve the stake at a given block – this is a strictly worse outcome despite marginal improvement in readability.
>
> Alternatively, if `wasStakerAt` is a function returning `(uint, bool)` tuple indicating stakedAmount at a given block:
>
> `wasStakerAt(spAddress, blockNum) returns (uint, bool)`
>
> The above pattern is returning a boolean that can be trivially generated by evaluating the stake at a given `block > 0`, which is identical to our current use of `totalStakedForAt > 0`.
>
> Finally, there is an option to create a function that returns `(uint)` the stake at a current block but also does additional validation in the form of require statements:
>
> ```
> wasStakerAt(spAddress, blockNum) returns (uint):
>     require(valueAtBlock > 0)
>     ...
>     return valueAtBlock
> ```
>
> However, we felt that this solution actually ends up less readable since the function `wasStakerAt` implies a returned boolean value but instead reverts on failure.
>
> For the above reasons, we have decided to forgo implementing the suggested wasStakerAt helper function in favor of our existing > 0 check for value at a specific block.

[L04] Lack of indexed parameters in events

Some events are defined with no indexed parameters. For example, `ClaimProcessed` in the `ClaimsManager` contract and `RegisteredServiceProvider` in the `ServiceProviderFactory`

contract.

Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Update: Partially fixed in pull request #614. Some important events still lack of indexed parameters.

Update: Fixed in pull request #657.

[L05] Contracts have artifacts leftover from earlier test versions

The whitepaper defines the native token's symbol as "AUDS" and its name as "Audius". However, the symbol used in the `AudiusToken` contract is "TAUDS" and its name is "TestAudius".

In the `ServiceTypeManager` contract there are two events called `Test` and `TestAddr` that are not used in the code.

Consider changing the name and symbol to the ones defined in the whitepaper, and consider removing the unused test events.

Update: Partially fixed. The test events were deleted in pull request #583. Audius' statement for the token issue:

> This was done on purpose so we wouldn't accidentally deploy test contracts to mainnet with the token symbol AUDS. The plan was to change the name and symbol before the final deploy of the contracts as to not pollute the ERC-20 landscape with the token name and symbol.

Update: Fixed in pull request #657.

[L06] Misleading comments, docstrings, and typographical errors

Some docstrings and inline comments were found to be erroneous. In particular:

- Line 412 of `ServiceProviderFactory.sol` says "Must have called requestDecreaseStake and waited for the lockup period to expire", but it is possible that the `DecreaseStakeRequest` was queued up by the `deregister` function instead of the `requestDecreaseStake` function.
- Line 486 of `ServiceProviderFactory.sol` says that the third parameter is `_oldEndpoint` but it is actually `_newEndpoint`.
- Line 123 of `Staking.sol` says `historry` but it should say `history`.
- Line 231 of `Staking.sol` has an incorrect `@notice` tag for the `undelegateStakeFor` function (it duplicates the one for the `delegateStakeFor` function).

Consider fixing these errors before deploying the contracts.

Update: *Fixed in pull requests #546 and #672.*

[L07] Outdated Solidity version in use

An outdated Solidity version, 0.5.16, is currently in use. As Solidity is now under a fast release cycle, ensure that the latest version of the compiler is used at the time of deployment (presently 0.5.17 in the 0.5.x branch).

Since OpenZeppelin version 3.0.0, the 0.6.x branch of Solidity is supported. Consider updating both OpenZeppelin and Solidity to their latest stable branches.

Update: *Fixed in PR#602. The project is now using the latest Solidity release from the 0.5 branch. Consider updating the* `pragma` *statement on all contracts to require this latest version or a newer one.*

[L08] Votes can be overwritten by calling the same function for the first vote

In order to vote in a proposal, stakers call the `submitProposalVote` of the `Governance` contract. If they want to update their vote, they call the same function.

This is giving two different responsibilities to the same function, which makes the code for the function more complicated than it should. It could also lead to mistakes by careless callers that might not notice they are updating their vote.

Consider splitting this function in two, to add a new `updateVote` function that takes care of updating an existing vote.

Update: *Fixed in pull request #596.*

README file is missing important information

The README.md files of the Audius roject have little information about what how to use, test, and contribute to the different components of the system. README files on the root of git repositories are the first documents that most developers often read, so they should be complete, clear, concise, and accurate.

Consider following Standard Readme to define the structure and contents for the README file.
Also, consider including an explanation of the core concepts of the repository, the usage workflows, the public APIs, instructions to test and deploy it, and how the code relates to other key components of the project.

Furthermore, it is highly advisable to include instructions for the responsible disclosure of any security vulnerabilities found in the project.
Consider adding a method for secure and encrypted communication with the team, like an email address with its GPG key.

Update: Fixed in pull request #601.

[L10] Proposed target code can change

When an approved governance proposal is evaluated, the `Governance` contract checks that the registered address has not changed. However, with the `create2` opcode it is possible to modify the code of a contract while keeping the same address, so this check is not enough to verify that the voted proposal is the same as the one that will be executed.

This is not a critical issue because the registry is managed by the trusted administrators of the system, and we started this audit with the assumption that they will thoroughly review all the contracts before adding them to the registry. It would be easy to spot a contract that self-destructs and opens the door to a `create2` deceiving attack.

For extra-safety and transparency, consider also registering the code hash of the contracts added to the registry, and verify that it has not changed before executing an approved proposal.

Update:: Fixed in pull request #605.

Notes & Additional Information

[N01] Inconsistent style for error messages

Throughout the code, when a `require` statement fails an error message is returned. These error messages are not following a consistent style. For example, in line 128 of `Governance.sol` a simple message is returned. In line 245 of `Governance.sol` a formatted message is returned, starting with the name of the contract and of the function. In line 23 of `InitializableV2.sol` a message in all capital letters is returned.

Consider using a consistent style for all error messages.

Update: Partially fixed in pull request #597. Most of the messages are now following the style `{Name of contract}: message`*. Nevertheless, a few are still inconsistent like the one in line 44 of* `AudiusAdminUpgradeabilityProxy.sol` *or the one in line 352 of Governance.sol.*

Update: Fixed in pull request #657.

[N02] Lack of explicit visibility in public state variables

The public state variables are implicitly using the default visibility. For example, in lines 39 to 42 of `Staking.sol` and in line 12 of `AudiusAdminUpgradeabilityProxy`.

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

Update: Fixed in pull request #592. Now state variables and constants have explicit visibility declared. Consider enforcing this with a linter.

[N03] Named return variables

There is an inconsistent use of named return variables across the entire code base. Consider removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both explicitness and readability of the code, and may also help reduce regressions during future code refactors.

Update: *Fixed in pull request #600. The name of single return variables was deleted. The Audius team decided to keep the names in functions with multiple return variables. Consider documenting the rationale for this decision in the code style file, and enforcing it with a linter. Audius' statement for this issue:*

> We chose to continue returning named variables for functions returning tuples so that non-contract clients can easily access the returned values – for example, in the function 'getServiceProviderDetails' a client can simply access 'object.numberOfEndpoints' instead of having to call "object['3']". While there may be a (very) slight decrease in contract readability, the other benefits that named values provide compelled us to use them.
>
> For single return values a named return variable is unnecessary as the function name provides sufficient context as to what is being returned. For example, 'getGovernanceAddress' returns an address that represents governance – a named return variable would offer no additional information and reduce readability. In certain cases we have updated NATSPEC comments with return value description for ensured clarity.

[N04] Renaming suggestions

To favor explicitness and readability, several parts of the source code may benefit from better naming. Our suggestions are:

In the `Governance` contract:

- `No` to `Rejected`.
- `Yes` to `Approved` or `ExecutedSuccessfully`.
- `TxFailed` to `ExecutedWithFailure`.
- `startBlockNumber` to `submissionBlockNumber`.
- `signature` to `functionSignature`.
- `submitProposalVote` to `submitVote`.

In the `ClaimsManager` contract:

- `fundBlock` to `fundedBlock`.

- `fundingAmount` to `fundedAmount`.
- `_address` to `_stakingAddress`.
- `_spFactory` to `_serviceProviderFactoryAddress`.
- `_delegateManager` to `_delegateManagerAddress`.

In the `ServiceProviderFactory` contract:

- `UpdateEndpoint` to `EndpointUpdated`.
- `sp` to `serviceEndpoint`.

Update: Partially fixed in pull request 595. Some functions or variables' names that derive from the suggestions are still using the old nomenclature such as in the `getLastFundBlock` *function from the* `ClaimsManager` *contract.*

Update: Fixed in pull request 657.

[N05] Declaring two variables for some addresses

In some contracts two variables are declared for addresses, one for the instantiated contract at that address and one for just the address. For example, `tokenAddress` and `audiusToken` of the `ClaimsManager` contract, and `registry` and `registryAddress` of the `Governance` contract.

It is not clear why the two variables are declared. With the address it is possible to get the deployed contract instance, and from the deployed contract instance it is possible to get the address, so there seems to be unnecessary duplication.

Consider refactoring the code to use only variable for the address or for the contract instance, to slightly reduce the number of lines of code to maintain and the number of variables to keep track of. If there is a reason to declare both, consider documenting it.

Update: Fixed in pull request #590. Only the instance variables are now stored, all addresses' variables have been removed.

[N06] Not declaring `uint` as `uint256`

In most parts of the code variables are declared as `uint` instead of `uint256`. For example, in lines 52 to 54 of `ClaimsManager.sol`

To favor explicitness and consistency, consider declaring all instances of `uint` as `uint256`.

Update: Partially fixed in pull request #593. There are still cases, such as in the `ClaimsManager` *contract, where the variables are defined as* `uint` *instead of explicitly make use of the* `uint256` *type.*

Update: Fully fixed in pull request #657.

Conclusions

3 critical and 12 high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.

Update: *The Audius team fixed all the critical, high, and medium severity issues that we reported.*

Additional security recommendations

Note*: The following recommendations were made after the Audius team had already addressed the issues from our initial audit.*

Audius has a lot of moving parts: A delegated staking system, a rewards program, a registration system, a voting and governance system, and a slashing mechanism. The code to implement all these moving parts is complex, and there are a lot of interactions between the various systems. We have done our best to raise as many valuable security issues as we could find during our time-limited engagement. However, we cannot guarantee that we've found all the bugs or enumerated all the risks in the Audius system.

Given the complexity of this system, and the large number of users that may interact with it, consider applying the following recommendations to further reduce the attack surface, mitigate risk, and get more eyes on the code:

- Beta testing: Consider engaging a community of early adopters to put the system under test with conditions as close to mainnet as possible.
- Bug bounty: Consider implementing a bug bounty program to get more eyes on the code, and to incentivize hackers to contribute with the system instead of attacking it.
- Future reaudit: Given the high number of critical and high severity issues found during this audit, the number of changes that were made as a result, and the complex interactions between the various systems, we suggest the Audius team to analyze the results of the beta testing and bug bounty to decide if the code should be reaudited with a fresh set of eyes.
- Security contact info: To make it easier for independent security researchers to contact Audius with any issues they may find, consider adding security contact info to the `audius-protocol` repo and/or the Audius website.

Upgrades Plugin

Open Source Tools

Relayer

Monitor

UI Builder

Safe Utils

Access Control

The Open Source Stack

Blockchains

Ethereum

Arbitrum

Base

OP Mainnet

ZKsync

Linea

Services

Smart Contract Security Audit

Blockchain Infrastructure Audit

ZKP Audit

Solutions

Ecosystem Stack

Financial Institutions Stack

Careers

Brand Kit

Trust

Terms & Policies

Terms of Service

Privacy Policy

Trademark Guidelines

AGPL License

Resources

Research

Documentation

Stats

Forum

Ethernaut CTF