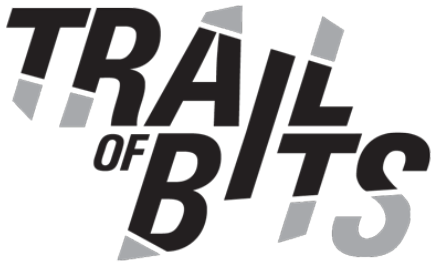

Livepeer

Security Assessment

Smart Contract and Token Protocol
March 12, 2018

Prepared For:
Doug Petkanics | *Livepeer*
doug@livepeer.org

Prepared By:
Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Chris Evans | *Trail of Bits*
chris.evans@trailofbits.com

Changelog

March 9, 2018:	Initial report delivered
March 10, 2018:	Added informational finding TOB-Livepeer-005
March 12, 2018:	Public release

Executive Summary

From February 26 through March 9, 2018, Livepeer engaged with Trail of Bits to assess the Livepeer system's smart contracts. The assessed contracts components were written in Solidity with a small amount of EVM assembly. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers.

The priority of the assessment focused on interactions between bonding management, job management, stake and earning allocation, and round progression. We directed extensive static analysis and dynamic instrumentation effort towards finding interactions that could lead to ill-gotten monetary gain or denial of service attacks against the protocol.

The code reviewed is of excellent quality, written with obvious awareness of current smart contract development best practices and utilizing well tested frameworks such as OpenZeppelin. The manager proxy and delegate controller contracts are restrictive enough to defend against any unauthorized administrative action.

The Livepeer protocol is designed to make several randomized decisions which are generally unfair and susceptible to collusion. The logic and state machines of the managers are very complex and could easily harbor new vulnerabilities as the result of a hasty future refactor. Edge-cases related to the floating-point and linked-list implementations can prevent delegated stake from being burnt after a slash. Analysis of the utility libraries for floating-point and linked-list implementations revealed some edge-case scenarios that prevent some delegated stake from burning after being slashed.

Overall the largest indicator of Livepeer's security strength is the consistency of its code. Integration and unit tests handle many edge cases that result from normal use of the protocol. Changes made in response to these findings can indirectly mitigate many exploit patterns. Extensive parameter handling and requirements also reduce the threat of malformed patterns.

As development of smart contract software continues, ensure the same level of consistency is maintained when adding features or upgrading pre-existing components. The current iteration of the smart contract protocol provides a secure foundation and meets many of the standards set by the Livepeer platform.

Appendix B and Appendix D contain references to implementation specifics that will help developing around certain areas of the code. Appendix C contains a short reference to the Slither static analyzer used in this engagement and accompanied with the final report. Appendix E discusses the challenges of pseudorandom number generation with respect to findings TOB-Livepeer-001 and TOB-Livepeer-003.

Engagement Goals & Scope

The engagement was scoped to provide a security assessment of the risk factors related to the core Livepeer smart contract implementation and ecosystem.

In particular we sought to answer the following questions:

- Is it possible for an unauthorized third party to gain administrative access to deployed Livepeer contracts?
- Are tokens managed and stored securely within the contract?
- Can participants manipulate the bonding and transcoding protocols to gain an unfair advantage?
- Is it possible to cause the contract services to enter an unrecoverable state?

The following components remained out of scope and were not examined as part of the assessment:

- The external TrueBit verification protocol for transcoded segments.
- Network protocols for peer-to-peer video streaming and playback.
- Transcoding libraries and software for desktop and mobile applications.
- The out-of-band storage and retrieval of transcoded segments on the Swarm layer.
- The Livepeer website and online media platform.

Trail of Bits conducted a detailed security analysis from the perspective of an attacker with access to the public Livepeer documentation and source code. We sought to identify risks, and scored their severity based on their likelihood and potential impact. We also sought to provide a mitigation strategy for each risk factor, whether it required a procedural change or a replacement of the solution, in whole or in part, with a more secure alternative.

Coverage

While the entire codebase was inspected for common solidity flaws, this audit focused on an in-depth analysis of the job, rounds, and bonding managers. Since the quality of the coding standards are so high, latent bugs are likely to be related to logic or concurrency.

ERC20 token implementation and genesis. Scenarios involving token ownership, transfer, and minting were assessed and tested. Usage of the OpenZepplin base templates were analyzed for attack surface exposure. The initial token release contract was verified to conform with standard ICO and crowdsale procedures. In the initial token genesis we looked for initial parameters that could trigger the end of token distribution and delegation prematurely. Proper handling of grant allocation arithmetic was inspected, but owner restrictions on critical functions prohibited any extensive tampering outside of initialization.

Token minting and inflation. The token minter contract was primarily analyzed for use cases that could manipulate inflation management to create unstable and unreasonable bonding rates. Conditional logic prevents underflowing the inflation. However, the absence of SafeMath might cause problems in the future.

Floating point arithmetic library. Percentage calculations were explored for edge cases that could cause unintentional behavior in the core contract logic. The precision limits of basic fractional operators were explored and tested for consistency and correctness against expected results.

Internal on-chain data structures. The double linked-list for transcoder pools was interrogated for bugs and situations that could corrupt the integrity of the data stored. It was also tested for resilience against unorthodox requests for rapid insertion and removal of nodes.

Job management protocols. Behavior for creation of transcoding jobs as well as claiming rewards for completed work were examined for vulnerabilities. The penalty mechanism for slashing dishonest participants was also explored for use cases of potential abuse.

Rounds Management Protocols. The mechanism for scheduling activity on the Livepeer network was inspected for issues that could lead to deadlock or miscalculation. The invariant assumptions of elapsed time, locking periods, and permissible function calls within a round were tested.

Earnings protocols. The mechanism for claiming work was inspected to see if the checks could be bypassed, *e.g.*, via a race condition. The verification methods were evaluated for determinism and their susceptibility to collusion. Fee and earnings share stability and

malleability was also covered. Finally, locking conditions were investigated to try and cause a job to disappear or become irretrievable.

Bonding protocols. Numerous bonding edge cases, race conditions, and timing sequences were investigated that are not yet exercised by the automated testing. For example, bonding to transcoders that have not yet registered, re-bonding mid-round, and transcoder re-signing and re-registration. Active transcoder election was also investigated to determine whether it is predictable or influenceable.

Contract controller interfaces and proxy contract delegation. Proxy contract delegation was reviewed briefly. No potential vulnerabilities were immediately apparent. However, there was insufficient time to complete an investigation into the possibility to abuse storage to change controller, manager, or owner addresses. Specifically, there might be the possibility that an upgrade introducing uninitialized storage pointers or tainted array lengths could enable the controller address and owner to be changed. While the base target is sparse enough that such a vulnerability does not appear possible, it does warrant further investigation.

Scalability. A small scale stress test was conducted with a dozen broadcasters, a dozen transcoders, and two dozen delegators. Other than being mindful of scalability concerns while auditing the code, no other specific effort was made into investigating scalability.

Project Dashboard

Application Summary

Name	Livepeer Protocol
Version	929182cc684410d55eb9048f47ed1ec3ab70461a
Type	Smart contracts
Platform	Solidity, Javascript

Engagement Summary

Dates	February 26 to March 9, 2018
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High Severity Issues	0	
Total Medium Severity Issues	0	
Total Low Severity Issues	3	■■■
Total Informational Severity Issues	2	■■
Total Undetermined Severity Issues	0	
Total	5	

Category Breakdown

Denial of Service	1	■
Arithmetic	2	■■
Cryptography	1	■
Undetermined	1	■
Total	4	

Recommendations Summary

Short Term

❑ **Explicitly scope the parameters for floating point arithmetic functions.** The precision limit these functions can handle should be programmatically enforced.

❑ **Limit the use of block hash and other deterministic entropy sources for pseudorandomness.** Transcoder elections are currently not fair. As soon as the Livepeer ecosystem contains enough ether to justify miner collusion, any architectural component that relies on the pseudorandom number generation (PRNG) scheme is threatened.

❑ **Have transcoder slashing always penalize a minimum amount of stake if possible.** Scenarios where a transcoder has non-zero stake but can be slashed without suffering a penalty encourages bad behavior on the Livepeer network.

❑ **Improve source code comments to describe state machine semantics.** Contracts like the Bonding Manager have very complex semantics that are not immediately transparent from the code. It would be very easy for these to be broken in a future refactor.

Long Term

❑ **Document and extend the floating point library.** Include detailed use cases of limits as well as explanations of parameter inputs and limitations. This will help developers' understanding for interacting with the library. Upstreaming these utilities to a third party framework may support the development of a robust system that will be able to handle extended precision needs in the future.

❑ **Use an external source of randomness, or none at all.** There is no safe way to use blockchain-derived randomness without risking collusion and/or unfairness.

❑ **Improve automated testing.** Create integration tests to cover all of the intricacies and edge cases of processes like bonding.

❑ **Ensure penalties are strong enough to deter misbehavior.** The transcoder slashing protocol is currently the primary mechanism that inhibits dishonest participation of processing video transcoding. Edge cases that do not proportionally punish misbehavior have the potential to negatively impact the network as a whole.

Findings summary

#	Title	Type	Severity
1	Transcoder election can be predictable and influenceable	Denial of Service	Low
2	Loss of precision for sufficiently high denominator and amount	Arithmetic	Informational
3	Pseudorandom number generation is not random	Cryptography	Low
4	Transcoders with low bonded stake can avoid slashing penalties	Arithmetic	Low
5	Bonding synchronization errors between data structures can enable stolen and locked tokens	Denial of Service	Informational

1. Transcoder election can be predictable and influenced

Severity: Low
Type: Denial of Service
Target: `BondingManager.sol`

Difficulty: High
Finding ID: TOB-Livepeer-001

Description
A malicious miner can influence the election by manipulating the job-creation block to achieve a desired hash, or simply by choosing not to publish a block that would favor an undesirable transcoder.

When a transcoder is claiming work, the JobsManager first ensures that the transcoder won the election. This is determined by confirming that:

1. the transcoder is active; and
2. the hash of the job's creation block, modulus the total stake among active transcoders, results in stake assigned to the claimant:

```
// Pseudorandomly pick an available transcoder weighted by its stake relative
to the total stake of all available transcoders
uint256 r = uint256(_blockHash) % totalAvailableTranscoderStake;
uint256 s = 0;
uint256 j = 0;

while (s <= r && j < numAvailableTranscoders) {
    s = s.add(activeTranscoderTotalStake(availableTranscoders[j], _round));
    j++;
}

return availableTranscoders[j - 1];
```

Figure 1: Election Assignment Block in `BondingManager.sol`

The Livepeer protocol specification does note that a transcoder can launch a *self-dealing attack* similar to the one described above if it is also an Ethereum miner.

Exploit Scenario
A malicious transcoder manipulates the hash of a job-creation block (*e.g.*, by transaction reordering or injecting spurious transactions) such that its desired transcoder wins the election.

Recommendation
There does not appear to be a resolution to this issue without changing the Livepeer protocol. An external source of randomness for the transcoder election would resolve this issue. While Livepeer is aware of this issue, we report it here in order to urge Livepeer to use a different, more secure, and fairer means of electing transcoders.

2. Loss of precision for sufficiently high denominator and amount

Severity: Informational
Type: Arithmetic
Target: `MathUtils.sol`

Difficulty: Low
Finding ID: TOB-Livepeer-002

Description

When using `MathUtils.percOf(amount, fracNum, fracDenom)`, if the amount is large enough where the fractional percentage is outside the precision range of the `PERC_DIVISOR` constant, then the returned value will always round down to 0.

```
    /*
     * @dev Compute percentage of a value with the percentage represented by a fraction
     * @param _amount Amount to take the percentage of
     * @param _fracNum Numerator of fraction representing the percentage
     * @param _fracDenom Denominator of fraction representing the percentage
     */
    function percOf(uint256 _amount, uint256 _fracNum, uint256 _fracDenom) internal pure
returns (uint256) {
        return _amount.mul(percPoints(_fracNum, _fracDenom)).div(PERC_DIVISOR);
    }


    /*
     * @dev Compute percentage representation of a fraction
     * @param _fracNum Numerator of fraction represeting the percentage
     * @param _fracDenom Denominator of fraction represeting the percentage
     */
    function percPoints(uint256 _fracNum, uint256 _fracDenom) internal pure returns
(uint256) {
        return _fracNum.mul(PERC_DIVISOR).div(_fracDenom);
    }
```

Figure 2: Susceptible functions in MathUtils.sol

Consider the scenario where a user attempts to calculate a fractional percentage point of large value:

```
fracNum = 1
fracDenom = PERC_DIVSOR + 1
amount  = 25000000
```

`MathUtils.percPoints` will return 0, causing the resulting quotient to always be 0, where in this case the actual value is closer to 249. Since this level of precision may be outside of

the required ceiling for Livepeer, and these functions are mostly used internally, the finding is classified as informational.

Exploit Scenario
Alice deploys a smart contract to handle participation in the Livepeer network. She imports `MathUtils.sol` to evaluate calculative logic for her own transactions perhaps involving wei and gas costs. Zero results for some cases cause her to lose money or make suboptimal decisions, causing her to withdraw her participation.

Recommendation
Require `MathUtils.validPerc` to verify that the product of `fracNum.mul(PERC_DIVISOR)` is `< fracDenom`.

In the long term, examine the use cases for fixed-point arithmetic and include clear documentation for the limits and restrictions for calling into these functions. Enumerate rounding decisions and scenarios that would result in a loss of precision.

3. Pseudorandom number generation is not random

Severity: Low
Type: Cryptography
Target: `BondingManager.sol`

Difficulty: High
Finding ID: TOB-Livepeer-003

Description
Transcoder assignment elections are not fair. The PRNG scheme is not uniformly distributed. The problem is that pseudorandom numbers are generated by taking a block hash modulus with an arbitrary sum of the bonded transcoder stakes. This will not produce a pseudorandom number. Specifically, the distribution of the randomly generated value is not guaranteed to be uniform across all active transcoders.

Figure 3 is a plot of the frequency of transcoders being elected from a pool of 100 transcoders across 10,000 elections with all transcoders having equal stake. The red line is the result of a fair election produced by a cryptographically secure PRNG, while the blue line is the result of the Livepeer election.

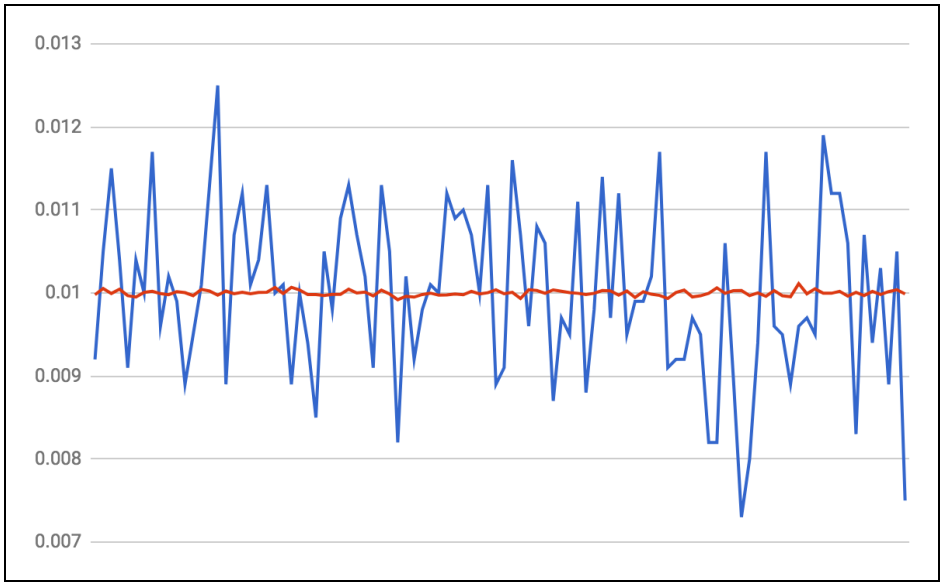

Figure 3: Red is a cryptographically secure PRNG, blue is the Livepeer PRNG

The X axis contains one bucket for each of the 100 transcoders, ordered by their index in the active transcoder set. The Y axis is the percentage of the elections in which each of the 100 transcoders were elected. As expected, the fair election results in each transcoder having a 1% probability of winning. However, the Livepeer contract election produces a very unfair distribution that is dependent on the transcoder's position in the active transcoder set, with some transcoders having over a 60% advantage relative to others.

Exploit Scenario
A malicious transcoder registers itself to be active at such a time that it is ensured to be in a position within `activeTranscoderSet` that maximizes its probability of being selected.

Recommendation
Do not use block hashes as a source of randomness.

An alternative might be to have some deterministic metric for which transcoder to select, such as the time since it was last selected weighted by its bonded stake.

If randomness must be used and an external source of randomness is not available, then use the block hash to seed an accepted pseudorandom number generation algorithm that is guaranteed to produce a uniform distribution. Such a solution will produce a fairer election, but it will not mitigate prediction attacks like TOB-Livepeer-001.

References
- Appendix E: Pseudorandom number generation in smart contracts

4. Transcoders with low bonded stake can avoid slashing penalties

Severity: Low
Type: Arithmetic
Target: `BondingManager.sol`

Difficulty: Low
Finding ID: TOB-Livepeer-004

Description
When a transcoder is slashed, the Job Manager sets a percentage of the bonded delegated stake to burn as a penalty. This is passed into `slashTranscoder` as `_slashAmount`. Based on this parameter, there will always be a maximum *n* number bounded stake that will not be burned for a `_slashAmount` of `PERC_DIVSOR/`*n*`+1.`

```
function slashTranscoder(
        address _transcoder,
        address _finder,
        uint256 _slashAmount,
        uint256 _finderFee
    )
        external
        whenSystemNotPaused
        onlyJobsManager
    {
        Delegator storage del = delegators[_transcoder];

        if (del.bondedAmount > 0) {
            uint256 penalty = MathUtils.percOf(delegators[_transcoder].bondedAmount,
_slashAmount);

            // Decrease bonded stake
            del.bondedAmount = del.bondedAmount.sub(penalty);
```

Figure 4: Penalty calculation does not check for 0

Slashed transcoders will be ejected from the registered pool. Since the bonded stake is neither subtracted nor burned, it remains available and can be withdrawn or rebonded to a different address.

Exploit Scenario
Alice registers transcoders on the Livepeer network that adjust their bonded stake to match the minimum tokens that cannot be penalized. As long as they remain in the transcoder pool, there is a pseudo-random chance for them to be selected and claim work. If they are penalized when selected for verification, Alice can re-register or withdraw the bonded stake and reintroduce a new transcoder into the pool.

Recommendation
Subtract the total amount of bonded stake if the penalty is 0 but `delegators[_transcoder].bondedAmount` is `> 0`. Alternatively, have a minimum penalty value and take the maximum between this and the calculated penalty.

In the long term, have integration tests that ensure penalized participants cannot evade the deterrent. In addition, awarding finders' fees for users' participation in slashing will incentivize abuse by malicious actors. Ensure that internal tests reflect scenarios in which transcoders may be unfairly reported.

References
- When a transcoder is slashed, the Job Manager sets a percentage of the bonded delegated stake to burn as a penalty via the `setMissedVerificationSlashAmount` and `setDoubleClaimSegmentSlashAmount` functions.

5. Bonding synchronization errors between data structures can enable stolen and locked tokens

Severity: Informational
Difficulty: Undetermined
Type: Denial of Service
Finding ID: TOB-Livepeer-005
Targets: `BondingManager.sol` and `EarningsPool.sol`

Description
Delegated stake is stored in two different data structures. If the data structures ever get out of sync, delegates will be able to claim earnings that are not owed to them, artificially reduce transcoders' bonded stake, and lock other delegates' tokens and bonding ability. This is because certain bonding checks validate against one data structure, while others validate against the other.

Delegated stake is stored in two different data structures within the Bonding Manager: `delegators` and `transcoderPool`. The former maps delegators' bonded stake to their delegates, while the keys of the latter store the sums of the delegated stake for the transcoders. If there is any way to break the synchronization between these data structures—such that the transcoder stake summations in `transcoderPool` are erroneous—then delegators would be able to remove stake from transcoders. When a delegator re-bonds to a new address or unbonds itself completely, if at that time its old delegated stake is for a registered transcoder, then its stake will be subtracted from that transcoder's key in `transcoderPool` according to the value in `delegators`.

```
    if (transcoderStatus(del.delegateAddress) == TranscoderStatus.Registered) {
        // Previously delegated to a transcoder
        // Decrease old transcoder's total stake
        transcoderPool.updateKey(
            del.delegateAddress,
            transcoderPool.getKey(del.delegateAddress).sub(del.bondedAmount),
            address(0), address(0)
        );
    }
```

Figure 5: Transcoder stake deletion in BondingManager.sol

The *eligibility* to claim earnings is based off of a delegator's `delegateAddress` in the `delegators` data structure, but the actual earnings calculation is based off of the delegated amount in the `transcoderPool` data structure. Moreover, if the erroneously reduced stake is sufficiently high, it can lock other delegators' stakes to that transcoder, as is demonstrated by the following exploit scenario. Finally, if the original delegator claims earnings *before* any other delegators bonded to the same transcoder—which will automatically happen when the transcoder either unbonds or re-bonds—then the original

delegator will receive earnings intended for the other delegators, since the rewards pool did not take into account the original phantom stake. When the other delegators subsequently attempt to claim earnings, a SafeMath assertion will fail when each delegator attempts to subtract their claims from the earnings pool, effectively locking their earnings and preventing the delegator from ever unbonding.

Note: The severity of this finding is classified as "Informational" because we were not able to exploit it. Therefore, it does not pose any immediate security risk. However, a future modification to the Bonding Manager contract could easily expose it.

Exploit Scenario

Alice wants to reduce the stake of Bob's transcoder. Alice has bonded 1,337 delegated stake to Bob's transcoder, Bob has bonded 1,000 of his own stake, and Carol has also bonded 2,000 stake to Bob's transcoder.

Assume that a synchronization error *does* exist that prevented Alice's 1,337 stake from appearing in the `transcoderPool` data structure. Therefore, from the perspective of the `transcoderPool`, Bob's transcoder will only have 3,000 bonded stake, not the correct amoung: 4,337.

Malicious Delegator Can Claim Additional Earnings

As long as Alice claims her earnings for Bob's transcoder's claimed work relatively early, she will get an undue increase in her reward. This is because the earnings pool's claimable stake is based off of the erroneous basis of 3,000 bonded stake. Therefore, Alice will receive 1,337/3,000 = ~45% of the reward pool instead of the 1,337/4,337 = ~31% that she actually deserves.

Malicious Delegator Can Artificially Reduce Transcoders' Bonded Stake

Alice eventually re-bonds her stake to a different address or alternatively un-bonds herself completely. This automatically calls `claimEarnings`, which will once again give Alice an undue share of the rewards. After Alice completes her bonding change, Bob's transcoder's address in the `transcoderPool` datastructure will have its bonded amount reduced by Alice's 1,337, resulting in an erroneous bonded stake of 1,663.

Other Delegators' Bonded Tokens Can Be Locked

Carol decides to re-bond to a different transcoder. However, she will be unable to because the claimable stake in the earnings pool is less than Carol's stake of 2000, causing an assertion error in `autoClaimEarnings`. Even if that were not the case, the safe subtraction in Figure 4 would also fail an assertion because `del.bondedAmount = 2000` but Bob's transcoder's value in the `transcoderPool` is currently `1663`. This effectively locks Carol's delegated stake, preventing her from unbonding, re-bonding, or claiming her stake.

Recommendation
Improve source code comments to provide better context for the interdependency between data structures and their semantics.

Consider improving the automated integration tests to check for bonding edge cases.

A. Vulnerability classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Arithmetic	Related to arithmetic calculations
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code quality recommendations

The following recommendations are not associated with specific vulnerabilities, however, they enhance readability and may prevent the introduction of vulnerabilities in the future.

Apply SafeMath operators consistently

- `require()` calls in contracts that check arithmetic conditions should also employ SafeMath to ensure consistent logic
  - `JobsManager.sol::setVerification*`
  - `GenesisManager.sol::setAllocations`
- Ensure that modified data structure fields are protected against potential overflows introduced by future code
  - In `SortedDoublyListLL.sol` the `remove()` and `insert()` calls increment and decrement `self.size`.
- Inflation management in `Minter.sol::setInflation` is protected by conditional logic, but the inflation decrease should use SafeMath just in case.

Improve Comments and Test Coverage in the Bonding Manager

- Bonding, unbonding, transcoder registration, and transcoder resigning are all very complex processes. There are many edge cases that are not currently covered in the unit and integration tests. For example, we recommend adding tests for cases when delegators delegate stake to unregistered transcoders which will later be registered.
- The current implementation of the Bonding Manager is somewhat brittle. Since the semantics of the `delegators` and `transcoderPool` datastructures are so complex, improve the documentation of these interactions' purpose, potential side effects, and any other functions or mechanisms in which they are relevant.

C. Slither

Trail of Bits has included our Solidity static analyzer, Slither, with this report. Slither works on the Abstract Syntax Tree (AST) generated by the Solidity compiler and detects some of the most common smart contract security issues, including:

- The lack of a constructor
- The presence of unprotected functions
- Uninitialized variables
- Unused variables
- Functions declared as constant that change the state
- Deletion of a structure containing a mapping

Slither is an unsound static analyzer and may report false positives. The lack of proper support for inheritance and some object types (such as arrays) may lead to false positives.

Usage

Launch the analysis on the Soldity file:

```
$ python /path/to/slither.py file.sol
```

Ensure that import dependencies and libraries, such as OpenZepplin, can be found by the solc compiler in the same directory.

D. Storage mapping deletion pattern in SortedDoublyLL

The Linked List implementation included in the utility libraries is the data structure in charge of managing candidate and reserve transcoder pool members. This implementation enforces the pool's size. This protects against edge cases that involve a mismatch in reported size and actual size of the list. However, it means that the only solution to decreasing pool size is to create a new one.

If multiple transcoder pools are used in the future, then storing this data in an array of structures could lead to a situation where a `Data` struct entry is deleted but the underlying `Node` mappings remain.

The following code demonstrates a scenario where old mappings from a deleted transcoder pool are included in new one:

```
    // Information for a node in the list
       struct Node {
           uint256 key;
           address nextId;
           address prevId;
       }
       // Information for the list
       struct Data {
           address head;
           address tail;
           uint256 maxSize;
           uint256 size;
           mapping (address => Node) nodes;
       }
    Data first_pool;
    Data second_pool;
    Data third_pool
    Data[] data_list;

    data_list.push(first_pool);
    data_list.push(second_pool);

    delete data_list[0];
    Data_list[0] = third_pool;
```

Figure 4: `third_pool` replaces `first_pool` but the underlying mappings will not get deleted

The Solidity documentation states that:

> *delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.*

As result, all the node information for `addresses->Nodes` persists. Since many checks use `key`, `prevId`, and `nextId`, it may be possible to massage the internal storage structures by reusing "deleted" space to create a pool that favors adversarial transcoders. For example, this pattern is exhibited in `contains()`, which checks if `node.key > 0` to determine if the linked list contains that entry. Since deleting the original structure does not recursively delete these fields, the linked list will erroneously report that it contains the node.

In order to effectively address this issue, ensure that calls to non-existent pools are reverted. In the long term, ensure that unit tests exhaustively check that underlying storage data is consistent with deletion logic.

E. Pseudorandom number generation in smart contracts

Pseudorandom number generation on the blockchain is generally unsafe. There are a number of reasons for this, including:

1. **The blockchain does not provide any cryptographically secure source of randomness.** Block hashes in isolation are cryptographically random, however, a malicious miner can modify block headers, introduce additional transactions, and choose not to publish blocks in order to influence the resulting hashes. Therefore, miner-influenced values like block hashes and timestamps should never be used as a source of randomness.
2. **Everything in a contract is publicly visible.** Random numbers cannot be generated or stored in the contract until *after* all lottery entries have been stored.
3. **Computers will always be faster than the blockchain.** Any number that the contract could generate can potentially be precalculated off-chain before the end of the block.

Even if miners are trusted or a specific situation is deemed appropriate for using block hashes as a source of randomness (*e.g.*, using the block hash of a *subsequent* block to a transaction), one still must be extremely careful about how randomness is generated. For example, say a contract needs to randomly select a winner from a set of *n* addresses that were submitted during a previous block. Since the current block's hash is random and assumed to be unpredictable and non-influenceable—an admittedly large assumption — then a naïve approach might be to calculate the winner like so:

```
winner = entries[blockHash % entries.length];
```

The trouble is that taking the modulus of a random number *does not* produce another random number.