

iExec Security Audit

of PoCo Smart Contracts

May 16, 2018

Produced for



by



Table Of Content

1	Foreword	1
2	Executive Summary	1
3	Scope	2
3.1	Included in the scope	2
3.1.1	Rewards and penalties	2
3.1.2	SGX	2
3.1.3	Misbehaving workers	2
3.1.4	Misbehaving schedulers	3
3.1.5	Misbehaving users	3
3.1.6	Marketplace	3
3.1.7	Miscellaneous	3
3.2	Out of scope	4
4	System Overview	5
4.1	Actors and Components	5
5	Audit Overview	6
5.1	Scope of the Audit	6
5.2	Depth of Audit	6
5.3	Terminology	6
6	Limitations	8
7	Details of the Findings	9
7.1	Security Issues	9
7.1.1	Worker rewards and penalties  No Issue	9
7.1.2	Scheduler can manipulate its reward percentage   Fixed	11
7.1.3	Scheduler can unfairly obtain kitty rewards   Addressed	11
7.1.4	Worker exploitation by scheduler  No Issue	11

7.1.5	Marketplace	✓ No Issue	12
7.1.6	Misbehaving market participant can obtain or delay results	M	12
7.1.7	Workers can be forced into pools	L	12
7.1.8	Consensus Auditability	✓ No Issue	12
7.1.9	Workers can circumvent single-pool limitation	M	13
7.1.10	Result without Payment	H ✓ Fixed	13
7.1.11	Joining and leaving a pool	✓ No Issue	13
7.1.12	Arbitrary Contracts Are Trusted	C ✓ Fixed	14
7.1.13	The worker score system can be gamed	H ✓ Addressed	14
7.1.14	Caching scores makes them outdated	L ✓ Addressed	15
7.1.15	Requirements on the result space are not documented	M ✓ Addressed	15
7.1.16	Correctness of the log function	✓ No Issue	15
7.1.17	No freshness when submitting results	L ✓ Addressed	15
7.2	Trust Issues		16
7.2.1	Scheduler has indisputable decision power	H	16
7.2.2	Scheduler is incentivized to maximize the number of workers	M	16
7.2.3	SGX-Related Trust Issues	L ✓ Addressed	16
8	Recommendations / Suggestions		18
9	Open Questions		20
10	Disclaimer		21

Foreword

We first and foremost thank IEXEC for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The IEXEC smart contracts constituting the PROOF-OF-CONTRIBUTION protocol have been analyzed under the agreed upon specification, with different tools for automated security analysis of Ethereum smart contracts and manual review. The issues listed in this report result from CHAINSECURITY's verification of this specification and should not be considered exhaustive.

While we found that IEXEC employs good coding practices and has clean, well-documented code, the current PROOF-OF-CONTRIBUTION implementation has a model that places trust in external contracts and key roles, introducing several issues.

Scope

IEXEC requested a precisely scoped audit, meant to assess the technical foundation of IEXEC's project in its current state, rather than establish its security soundness in the context of an actual deployment.

To define this scope, CHAINSECURITY listed potential points of failure and agreed with IEXEC upon them.

Issues that have been encountered while verifying this specification have been listed, even when they were not explicitly mentioned in the specification. However, this list should not be considered exhaustive with respect to the security of IEXEC's smart contracts, due to the restricted nature of this audit. In particular, CHAINSECURITY does not claim to have spent the effort required to fully audit such a project to the extent it deserves; rather, CHAINSECURITY strived to verify the points listed below, to provide a report which could serve as potential guidelines in the future, and to prepare for full audits to be carried out before the final release and deployment.

Included in the scope

Rewards and penalties

CHAINSECURITY has checked whether the following reward and penalty situations are correctly enforced.

Worker rewards and penalties

- No contribution, consensus: no penalty
- Wrong contribution: stake seized, loss of reputation
- Correct contribution, doesn't reveal, someone else revealed: penalized as when result is wrong
- Correct contribution, doesn't reveal, nobody revealed, Consensus is reopened: workers are punished (seized in `finalizeWork`) and excluded from the newly opened consensus.
- Correct contribution, doesn't reveal, nobody revealed, Consensus is not reopened: stake is unlocked when failed consensus is reclaimed
- Correct result, reveals result: stake unlocked, contribution reward, reputation gain
- Contribution, no consensus: no direct consequences, unlocked stake after timeout
- No contribution, consensus timeout: no penalty

Furthermore, CHAINSECURITY has studied the optimal strategy for an adversary with regards to the worker rewards, e.g. should the adversary use one or multiple accounts. This also includes whether worker rewards are proportional to their consensus weight.

Scheduler rewards and penalties

- Scheduler receives a correct percentage of the reward if consensus is achieved.
- Scheduler cannot significantly influence its reward by choosing the workers so that there is a big residue.

SGX

- The contracts correctly check whether the provided challenge was resolved.
- A potential bonus is only given if the challenge was resolved correctly.

Misbehaving workers

- Workers are required to commit a (possibly null) security deposit (stake) which is seized in case of bad behavior. A deposit has to be made, when joining a pool and when contributing to an order.
- The incorrect result of less than half the workers (weighted by importance) does not change the consensus result.
- Workers cannot leverage contributions made by other workers to submit a valid contribution without computing the result themselves.

- Workers cannot use the output “The user chose the wrong category” as a beneficial strategy.
- As discussed above, how does the stake act in the case of Sybil attacks? Does it provide something beyond a “secondary reputation”?
- A competing scheduler cannot deteriorate a competitor’s service (e.g. increase its latency) too significantly by registering many fake workers.
- Workers cannot make inconsistent contributions (e.g. contributions which do not match their address).
- Each participating worker needs to have the required amount of reputation and stake for an order.
- A worker can only be member of one worker pool at any time.

Misbehaving schedulers

- Workers can observe when they are being “overused” by a scheduler to lower latencies, which would reduce their benefits.
- Schedulers cannot successfully collude with a single worker apart from the worker selection process.

Misbehaving users

- A misbehaving user cannot obtain the result without making the agreed payment.

Marketplace

- Each market order can only be consumed once
- Market orders cannot be modified by unauthorized parties
- After a market has been consumed, its parameters are immutable
- Generic ethereum security issues
- Reentrancy
- Arithmetic overflow and underflow
- Unhandled exception
- Numeric imprecisions
- Common security practices, e.g. as documented in the Solidity documentation

Miscellaneous

- The log function in SafeMathOZ is correct.
- Constraints on arrays exist and are correctly enforced
- RLC tokens are handled securely. No tokens are lost. Locked tokens cannot be withdrawn.
- Correctness of basic computation, e.g. rounding errors
- All the operations from the sequence diagram have to be atomic.
- A worker can leave a pool at any time or be evicted by the scheduler. Both have an equivalent effect of unlocking the worker’s stake.
- A leaving worker can finish leftover orders from a worker pool and will be rewarded or penalized for those as usual.
- The validity and authenticity of different contracts can be verified.
- Work orders have unique identifiers.
- Each user has a reputation according to previous contributions that is preserved when it switches worker pools.
- The policies of a worker pool (stake ratio, scheduler reward, minimum stake, minimum score) can only be changed by the scheduler.

Out of scope

- SGX integration, including attestation
- Privacy and security of the datasets and the applications
- Scheduler-related issues
- Consensus-related formulae
- Proper random selection of workers
- No proper penalty if `finalizedWork` is never called
- No proper penalty for not providing the correct result at the URI
- Secure Communication between the involved parties
- Security of dApps
- Correctness of computed result
- Malicious application providers that provide incorrect applications
- A worker's decision procedure on when to switch to another worker pool because it is treated unfairly
- A scheduler's decision procedure on when to remove a worker
- Misbehaving users which claim to have received incorrect results
- Access Control on the result URI

System Overview

In the following we describe the PROOF-OF-CONTRIBUTION protocol (PoCo), its main components and how these interact.

Actors and Components

The PROOF-OF-CONTRIBUTION protocol conceptually differentiates between three agents on the IEXEC platform, with each one of them having different incentives for participation.

- **User**
A user of the IEXEC platform can pay a fixed amount of RLC tokens to have an order executed by workers. He picks an ask order from the marketplace and provides details of the execution. Once the deal is in place and execution process completed, the results are written back to the blockchain and are retrievable by the user.
- **Worker**
A worker on the IEXEC platform can offer his services to the aforementioned users, executing their orders in return for a reward in the form of RLC tokens and an increased reputation upon successful completion. Workers subscribe to managed pools and stake a certain amount of tokens on their result contributions, which gets confiscated in case the contribution turned out to be wrong. These worker pools have different policies and participation requirements, such as a certain reputation threshold. Worker pools then submit ask orders to the marketplace, describing their resources and trust level in terms of reputation.
- **Scheduler**
Schedulers manage the worker pools, setting their policies, coordinating the order execution process and determining the consensus result. For this, a scheduler gets rewarded a percentage of the payment made by the user who submitted the job.

The actors described above interact through several components, which constitute the IEXEC platform. These are the following:

- **Marketplace**
The marketplace is a central piece of the PoCo protocol, where the scheduler, in his role as a worker pool manager, submits task orders to the orderbook which can then be consumed by a user. Such an order has a value and volume, referring to the amount of times it can be consumed, as well as a required trust level for execution, and other parameters¹.
- **Workerpool**
The worker pool is fully controlled by his owning scheduler, who makes decisions about whether or not to allow worker subscriptions and the pool policy. A policy is composed of: the percentage of the reward to stake, the percentage of the reward going to the scheduler, the minimum subscription and reputation stake, as well as the stake size to be locked when a worker joins a pool.
- **Hubs**
The Dataset-, App- and IEXEC-Hubs are separate smart contracts with distinct functionalities. The App- and Data hubs are being used by the IEXEC hub to register applications to be executed and datasets to be used by the workers. The IEXEC hub itself is the central managing instance, controlling the escrow for the agents' stakes, emitting relevant work order events and providing a link to the RLC token contract.

¹Direction (Bid, Ask, Unset, Closed), worker pool and its owner, and remaining volume.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on April 26, 2018, and updated on May 21, 2018:

File	SHA-256 checksum
App.sol	7de4b27bf57ff26c4004075fe35a2a33253bb104b20fbdfa6319867ff2461609
AppHub.sol	d2d7111009ebc07c8569ea98c4134ccee6aa45c36f2944223f8c7a65f6815bb
Dataset.sol	4984c442f1d0e172f818738465a64435d3ed6658a0e8696be25a3d4bca385270
DatasetHub.sol	5b40e7c8a4fe601bc809b4e6712305ed1bee909e1901f26983f8faa3b6bdd090
lexecAPI.sol	3592c032707152fd5b6d78791225b2fee313802c5a7b19dab37470120e523277
lexecCallbackInterface.sol	258cdfb7b93c752ddfdb8030ecc62fd1fcdcb8699e7e6c986d5b07636bfc3644
lexecHub.sol	f05a1c8855d310ad5452ac1023738edb9452730036f8fea1ddcbcd5a591ed343
lexecHubAccessor.sol	02cb46962750eea69181743d6911c683611cd5ce58870c932a8707d848348454
lexecHubInterface.sol	b8a3c910eb18fad425467d8958eb2cb9a52c58178365536da39434473f7cb9e3
lexecLib.sol	5c2be259013563ee06201b873023e8ada997c021d077cfa275ef3a76c0a3a712
Marketplace.sol	0fb28dcab57e31694aa0b0ae891c114fa22664ea5bb71dfd4f97ae93939f8b20
MarketplaceAccessor.sol	2a9fcbcc49a75b842439ffe8971ec85781f7cc0a1e916604d408e63cc31837e4
MarketplaceInterface.sol	e1c488645cfbfc8ad69afb007d6b1faf9764723a1d4dea5af6435772024f1770
OwnableOZ.sol	3f548ab4d71aa1836cc63e37a5d8412ecccb3504809a7c62d4e966fd2cd90ca7
SafeMathOZ.sol	e9d48bf5b1a6b029fd7d9dc1e98afaa40fadf416cae1a0107467d1ac113f03d0
TestSha.sol	8546e0518f4dc8e1d45b15c9119f380ee39baa131c63f09baddd6a8a067bb13e
WorkOrder.sol	275e5093baf0a5143d38d468d765803217af4e7b476972309d2d94fb9a6248e0
WorkerPool.sol	76a61b1d31a05634d13ac0269a35a3ef3abad6ab159c1e8ad6bc02cc1da80cd1
WorkerPoolHub.sol	8d60cb10e05695d0dc9b6b01517bfecdd8ec86ac1dbd5f5d2c1d5b812d376a43

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology²).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

-  - Low: can be considered as less important

²https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

-  - Medium: should be fixed
-  - High: we strongly suggest to fix it before release
-  - Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as . Finally, if during the course of the audit process, an issue has been addressed technically, we label it as , while if it has been addressed otherwise we label it as .

Findings that are labelled as either  or  are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

Details of the Findings

In this section we detail our findings, including both positive and negative findings.

Security Issues

In this section, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Worker rewards and penalties ✓ No Issue

A crucial part of IEXEC's platform is the incentive model, which is supposed to motivate workers to subscribe to worker pools managed by schedulers, then contribute the results of their computations, after which they get rewarded should their contribution be correct. Figure 1 shows the workflow of a worker participating in this process.

Workerpool.sol

W = worker

S = scheduler

c = contribution

* = correctness only determined by s

* = only new workers in the same pool can contribute, or workers with previously incorrect contributions, Punishment incurred

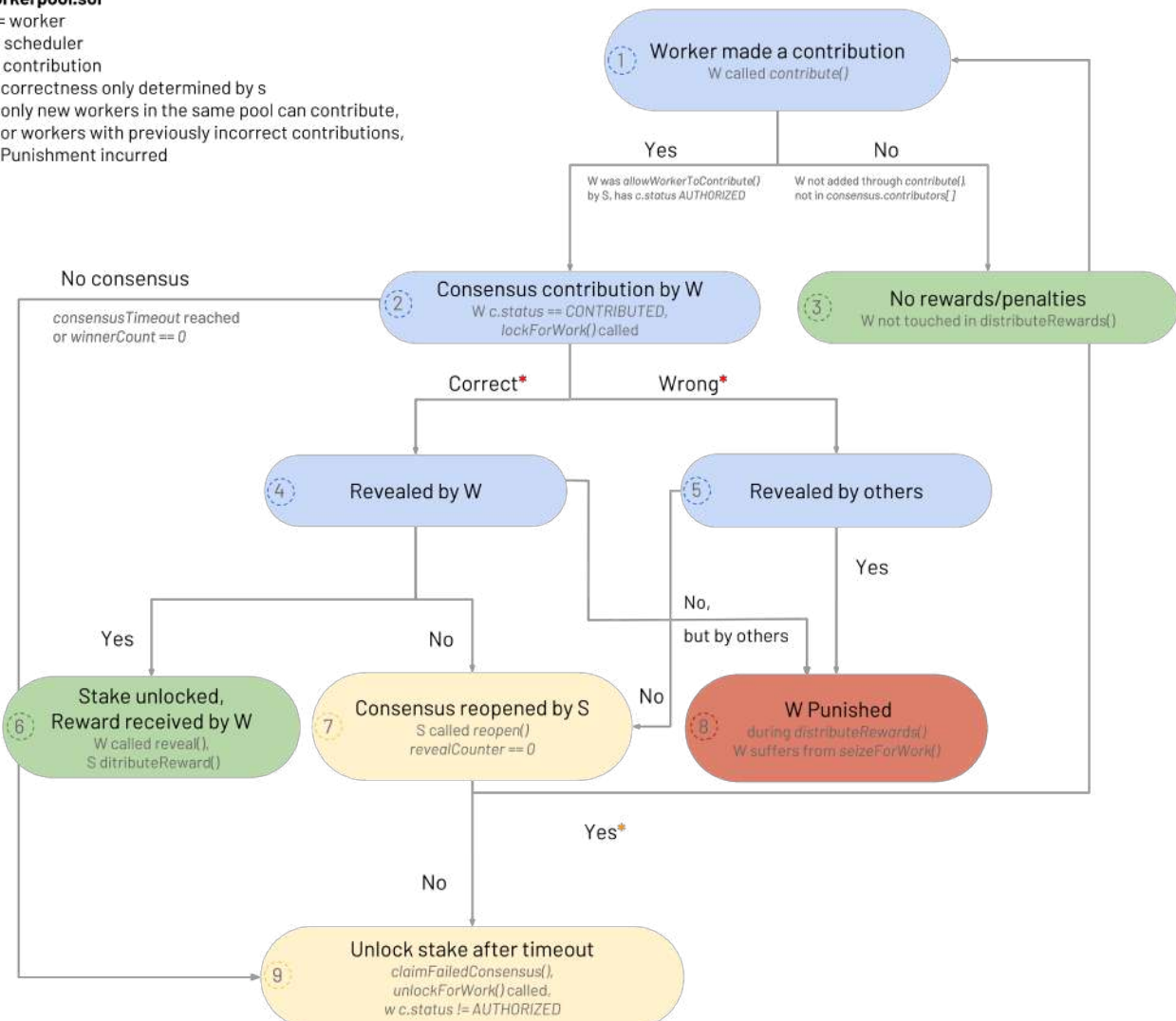


Figure 1: Worker perspective on participating in an order execution

The diagram should be read from top to bottom. We now consider an exemplary workflow.

A worker makes a correct contribution by calling `contribute()` (1), which is accepted if the scheduler previously allowed this worker to contribute. During the contribution, the worker's stake is locked (2). The consensus contribution of this worker is correct and he needs to reveal it (4). However, this does not happen,

but all other workers in this pool reveal their results. This leads to a punishment of the worker who did not reveal his result, meaning that he loses his stake and some reputation (8).

Following the specification, the behaviour list below has been verified through manual review and the exhaustive test suite provided by IEXEC, which offers a high test coverage ³.

- A non-contributing worker is neither rewarded nor punished if a consensus is reached.
- A non-contributing worker is neither rewarded nor punished if no consensus is reached.
- A worker contributing a wrong result loses his submitted stake and some reputation if consensus is reached.
- A worker contributing a correct result that he does not reveal loses his stake and some reputation when consensus is reached.
- A worker contributes a correct result and does not reveal it, but neither does any of the other workers. The scheduler reopens consensus and as a consequence all workers that contributed the correct result but did not reveal it cannot participate in the next round and all lose their stake and some reputation.
- A worker contributes a correct result and does not reveal it, but neither does any of the other workers. The scheduler does not reopen consensus and once the user ordering the job claims a failed consensus after the timeout, all stakes of workers participating in this pool are unlocked.
- A worker contributing and revealing a correct result will regain his stake, increase his reputation and get a reward upon completion of consensus.
- A worker contributing and revealing a result in a worker pool that does not achieve consensus does not gain a reward or get punished and his stake is unlocked after a timeout.
- A worker has a reputation according to previous contributions that is preserved when he switches worker pools and can only be manipulated by a malicious scheduler.
- The policies⁴ of a worker pool to which a worker subscribed can only be changed by the current scheduler.

We note that the correctness of a worker's contribution is determined solely by the scheduler, which is extensively discussed in the trust section. Reputation loss is set to the minimum between 50 and the current worker reputation. Meanwhile, reputation is only increased in steps of 1.

Each worker's reward should be proportional to his consensus weight. This is indeed the case and can be seen by the formula encoded in the smart contract. First, the bonus for each worker depends on whether or not SGX is used.

$$b = \begin{cases} 3 & \text{if worker uses SGX} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

Then each worker's consensus weight is calculated, influenced by his bonus and score, which is also referred to as his reputation.

$$w_{worker} = \max(0, \log(s_{worker} \times b)) \quad (2)$$

The reward of all workers combined is calculated based on the scheduler's reward ratio policy (his reward percentage) and the pool reward, consisting of the payment for that order and all stakes lost. Finally, a single worker's reward is a weighted fraction of that.

$$r_{workers} = \frac{r_{pool} \times (100 - p_{scheduler})}{100} \quad (3)$$

$$r_{worker} = \frac{w_{worker} \times r_{workers}}{w_{total}} = \frac{w_{worker} \times r_{pool} \times (100 - p_{scheduler})}{100 \times w_{total}} \quad (4)$$

Hence, we can conclude that a worker's reward is directly proportional to his consensus weight.

³100% coverage of all statements, functions and lines and 81.73% of branch coverage in the WorkerPool.sol contract. This result can be verified by running `npm run coverage` on the provided codebase

⁴These include stake ratio, scheduler reward, minimum stake, minimum score

Scheduler can manipulate its reward percentage H ✓ Fixed

A scheduler can call `changeWorkerPoolPolicy` after workers submitted their contributions and revealed them, arbitrarily increasing his reward percentage up to 100%. While the workers get notified about this by a corresponding event, there is no action left for them except to unsubscribe from the worker pool, which would leave them with no rewards at all.

Storing the `m_schedulerRewardRatioPolicy` in consensus in the `emitWorkOrder` function would resolve this issue.

Likelihood: High

Impact: Medium

Post-Audit Fix: IEXEC adapted the proposed solution and the scheduler reward is now stored with the consensus object.

Scheduler can unfairly obtain kitty rewards M ✓ Addressed

Whenever a user calls `claimFailedConsensus` through the IEXEC hub, the `workerPoolStake` is seized from the scheduler who failed to deliver a consensus result and then payed out as a bonus to the next scheduler who successfully finalized a work order.

Another scheduler can, on purpose, give itself a very long-running order it never finishes. However, once he observes the `WorkOrderClaimed` event, indicating a claim on a failed consensus, he finishes it to get the kitty reward. This can be repeated an unlimited number of times.

Likelihood: Medium

Impact: Medium

Post-Audit Fix: IEXEC is aware of this possibility and concludes that, considering an attacker would have to create market orders, find or run accomplice workers and pay gas for the dry run of the POCO just to retrieve small amounts of value, this is not a significant issue.

Worker exploitation by scheduler ✓ No Issue

The scheduler has an inherently stronger position than the worker. However, while he can manipulate certain policy elements (as mentioned in a separate issue on page 11), workers cannot be completely exploited by a scheduler to deliver free results.

A worker can adapt the following optimal procedure to avoid providing free work:

- Wait for the event `AllowWorkerToContribute` before doing any work.
- Estimate revenues based on `m_consensus[workOrder].poolReward` and `m_schedulerRewardRatioPolicy`. However this is only reliable once the issue allowing a scheduler to change his reward percentage is fixed.
- Simulate a call to `contribute()` privately and verify that it succeeds. It might fail, for instance if the enclave challenge was incorrect.
- At this point, the worker has revealed no information, therefore it is not being used. Proceed with the protocol. See the worker penalties and rewards in figure 1.
- If own `resultHash` is not the winner result, do not reveal own result.

We remark the following limitations:

- A `resultHash` can be contributed, but no rewards paid, if the scheduler picks another `resultHash`, or calls `reopen`.
- Results can be revealed, but no rewards paid, if the scheduler stalls and `claimFailedConsensus` is not called.
- There can be value in the `resultHash`, even without revealing the result, such as detecting success in crypto-currency mining.

Marketplace

The marketplace is implemented in three different contracts:

- Marketplace
- MarketplaceAccessor
- MarketplaceInterface

We verified that:

- Each ASK market order can only be consumed `volume` times.
- ASK market orders cannot be modified by unauthorized parties and only workpool owners can create or close their corresponding orders.
- After a ASK market order has been consumed its parameters are immutable.

Misbehaving market participant can obtain or delay results

Because of the costly computations in the loop of the `WorkerPool` contract during the scheduler's call to `distributeRewards`, where the contract iterates twice over all contributors of a given consensus, this function will exceed the block gas limit⁵ once more than 120 workers participate.

An attacker can prevent the finalization of a work order by spawning and subscribing enough workers to exceed the block limit, which will require $121 - n_{realWorkers}$ workers with sufficient stake and score according to the pool policy.

Even under the assumption that worker selection is performed randomly, a worker pool with more worker will make it easier for an attacker to fill it up to exceed the 120 participant threshold.

Under the assumption that in such a worker pool at least one worker revealed his contribution, the only way to resolve this deadlock is for `claimFailedConsensus` to be called. (Note that, as described on page 18 `claimFailedConsensus` is also limited in the maximum size it can handle.) This results in all stakes, including the resources the attacker dedicated for this delay, to be unlocked again. More so, the current scheduler's stake gets confiscated and will supposedly be used as a bonus for the next well finalized task, giving the attacker additional incentive (see page 11), since he could reinstall himself as the next scheduler in addition to purposely delaying the result in the previous round, where he could have already obtained the results from other workers for free.

Likelihood: Low

Impact: High

Workers can be forced into pools

An attacker can publicly register a worker pool by calling `createWorkerPool` through the `iExecHub` contract, hence being able to subscribe arbitrary workers to his pool by calling `registerToPool`, as long as they are not already registered somewhere else.

This call locks workers' deposits and registers their affectation to this malicious pool, blocking a part of their funds and the possibility to do work in other pools, without workers' consent and cost for the attacker. The workers will only be able to stop this once the call to `unsubscribeFromPool()` succeeds.

Likelihood: Medium

Impact: Low

Consensus Auditability

The question of whether or not schedulers can successfully collude with a single worker apart from the worker selection process, boils down to the question of whether any such conspiracy can be detected. Under the assumption that it is not possible to deploy arbitrary `workOrder` contracts and all of them run the official code, the consensus can be fully audited.

Once the event `FinalizeWork` is observed for a particular `workOrder`, anyone can rely on the fact that no further events, payments, or state changes will happen to this `workOrder`. Therefore, they can start reviewing the sequence of past events (`AllowWorkerToContribute`, `Contribute`, `Reopen`, `RevealConsensus`, `Reveal`), and verify whether the consensus protocol was followed by the scheduler and workers.

This is confirmed by `m_status = COMPLETED` in `WorkOrder.setResult`. All functions involving a `workOrder` require that `m_status != COMPLETED`. The `COMPLETED` status is final and hence the round is finished.

⁵Assuming a gas limit of 8 million.

Workers can circumvent single-pool limitation M

The specification requires that workers should only be able to work in a single pool at a time. However, the current design allows workers to practically be part of multiple pools at the same time.

The overall idea is that a worker can leave a pool immediately after `allowWorkerToContribute` has been called. Then, the worker remains unassigned and only joins a new pool as that pool is about to receive a new work order.

Here is a concrete example:

- Worker W is part of pool A and is randomly chosen as an allowed contributor for order O_1
- W leaves pool A, but can still contribute and reveal for O_1 and get the reward
- Pool B gets a new work order O_2 through `buyForWorkOrder`
- W front-runs this transaction to join pool B
- W gets randomly chosen as an allowed contributor for order O_2
- W leaves pool B, but can still contribute and reveal for O_2 and get the reward
- Pool C gets a new work order O_3 through `buyForWorkOrder`
- W front-runs this transaction to join pool C
- ...

We note that this strategy is in theory extendible to an arbitrary number of pools.

Likelihood: Medium

Impact: Medium

Result without Payment H ✓ Fixed

A malicious consumer could try to obtain the result without having to pay for the completion of the order. During the finalization of an order an optional callback is made:

```
109         {  
110             // optional dappCallback call can be done  
111             require(IexecCallbackInterface(m_callback).workOrderCallback(  
112                 this,  
113                 _stdout,
```

WorkOrder.sol

If the consumer makes sure that this callback will never succeed then the order can never be finalized. Hence, the consumer never has to pay as it can claim a failed consensus after the timeout.

Even though the callback fails, the consumer can observe all the relevant parameters inside the contract execution on the blockchain (e.g. on etherscan). Therefore, the consumer receives the result without having to make a payment.

Likelihood: Medium

Impact: High

Post-Audit Fix: IEXEC fixed this issue by removing the callback from the general workflow. It can now be triggered separately, but does not act as an automatic callback any longer. Therefore, the issue does not persist.

Joining and leaving a pool ✓ No Issue

Workers are expected to go from one pool to another, depending on how well schedulers perform. This possibility creates a healthy competition between pools, but it also makes the implementation more complicated, given that IEXEC has to handle all the possible cases, e.g. what if a worker leaves in the middle of an order?

CHAINSECURITY found no issue in how IEXEC handles these cases. In particular, a worker can leave a pool at any time or be evicted by the scheduler. Both have an equivalent effect of unlocking the worker's stake. A leaving work can finish leftover orders from a worker pool and be rewarded or penalized for those as usual. This holds under the assumption that the pool is implemented as designed in the `Workerpool1.sol` contract and is not some arbitrary contract.



IEXEC has implemented several contracts meant to fulfill separate roles. However, the current implementation implicitly relies on every contract being used as is, without modification, especially not modifications with malicious intent.

In particular, the interface of the `WorkOrder` contract is used in the `claimFailedConsensus` function of the `IexecHub` contract.

```
function claimFailedConsensus(address _woid) public returns (bool)
{
    WorkOrder workorder = WorkOrder(_woid);
    require(workorder.m_requester() == msg.sender);
    WorkerPool workerpool = WorkerPool(workorder.m_workerpool());

    IexecLib.WorkOrderStatusEnum currentStatus = workorder.m_status();
    require(currentStatus == IexecLib.WorkOrderStatusEnum.ACTIVE || currentStatus == IexecLib.
        WorkOrderStatusEnum.REVEALING);
    // Unlock stakes for all workers
    require(workerpool.claimFailedConsensus(_woid));
    workorder.claim(); // revert on error

    uint256 value = marketplace.getMarketOrderValue(workorder.m_marketorderId()); //
        revert if not exist
    address workerpoolOwner = marketplace.getMarketOrderWorkerpoolOwner(workorder.m_marketorderId()
    ); // revert if not exist
    uint256 workerpoolStake = value.percentage(marketplace.ASK_STAKE_RATIO());

    require(unlock (workorder.m_requester(), value.add(workorder.m_emitcost()))); // UNLOCK THE
        FUNDS FOR REINBURSEMENT
    require(seize (workerpoolOwner, workerpoolStake));
    // put workerpoolOwner stake seize into iexecHub address for bonus for scheduler on next well
        finalized Task
    require(reward (this, workerpoolStake));
    require(lock (this, workerpoolStake));

    emit WorkOrderClaimed(_woid, workorder.m_workerpool());
    return true;
}
```

This function can be called by anybody (since it is a public function), with any argument representing the address of a work order. No check is made on this address, and the code will execute provided that the corresponding function is implemented. In particular, this will also work if the work order was never on the market place to begin with. Given that this function relies on the value of `workorder.m_marketorderId()` to retrieve parameters from the marketplace, a malicious work order can easily reuse a preexisting value to prevent failures.

An attacker can thus unlock arbitrary amounts of an arbitrary address (defined by `workorder.m_requester()`), and seize funds of any address that ever designated a scheduler.

Although this is the most serious vulnerability that CHAINSECURITY found with respect to interacting with arbitrary contracts, CHAINSECURITY suspects that there may be more, and thus recommends IEXEC to make sure that either the interfaces used are always safe being trusted, or that the outputs of said interfaces are checked before being used blindly.

Likelihood: High

Impact: High

Post-Audit Fix: IEXEC fixes this issue by registering legitimate work orders inside the `m_woidRegistered` variable. Only such work orders can be used in any subsequent function calls.

As these work orders are created according to IEXEC's template, the issue does not persist.



A worker's score is shared among all worker pools in which he is ever involved. A worker wanting to get a higher share of the rewards could artificially inflate its own score, by creating its own pool and submitting lots of small work orders. For every completed order, its score increases.

If that worker later completes an order in a regular pool, its reward can easily be three or four times larger due than usual due to the inflated score.

Note, that a worker can arbitrarily inflate its score using this technique, however, due to logarithm involved in reward calculation the benefits are diminishing.

Likelihood: High

Impact: Medium

Post-Audit Fix: IEXEC states that if some suspicious reputation gains are observed for a worker the scheduler can blacklist him and not feed him any work. The burden is thus on the scheduler to do this due diligence.

Caching scores makes them outdated

The score of a worker is only checked when it is registered to a pool, so a diminishing score would not automatically lead to the eviction from the pool, and the score used in the reward system is not updated to take the latest value into account. Similarly, if the score requirement for a pool is raised, existing members below the requirement will not be automatically ejected.

In practice, this would be countered by some due diligence from the scheduler, who would evict workers manually; however some mechanisms could be put in place to automate these steps in the smart contracts themselves.

Likelihood: Low

Impact: Low

Post-Audit Fix: IEXEC is aware of this situation and states that the score in the reward system should not be updated to provide auditability of the sarmenta formula computed by the workerpool and that the scheduler can handle this issue.

Requirements on the result space are not documented

A yes or no question⁶ has only two possible results, encoded in only two different hashes. In these conditions, reverting the hash is possible for a lazy worker, who would then follow the majority and submit a contribution without computing the actual order. This breaks the concept of confidence which is ingrained in IEXEC's system.

The same applies to any question which has a small set of possible answers, and can be generalized to questions for which answers are likely to be in a small subset of all possible answers.

Overall, CHAINSECURITY was not able to find the assumptions on the concrete requirements on the result space which users have to take into account before submitting a work order. These should be made explicit, and automatically enforced if possible.

We suspect the likelihood of this issue to be low, but it greatly depends on the use cases of IEXEC's platform, which are to predict a priori.

Likelihood: Low

Impact: High

Post-Audit Fix: IEXEC acknowledged the lacking documentation and committed to provide the missing information.

Correctness of the log function

We tested the implementation of the log function (more precisely, the ceiling of the logarithm function with base 2), which was taken from the Internet⁷.

The computations are correct up to at least 2^{228} , which seems to match the use of IEXEC.

No freshness when submitting results

There is no freshness involved when submitting results. This means that, if two different work orders have the same results, a lazy worker can see that some of the hashes submitted to the last order are the same as for the first, and thus reuse the result which was previously revealed without having to compute it.

Likelihood: Low

Impact: Medium

Post-Audit Fix: IEXEC will instruct application providers to make sure that their applications' results are unique (e.g. by containing an internal nonce). Then, the issue becomes irrelevant.

⁶such as the satisfiability of a logic formula

⁷<https://ethereum.stackexchange.com/questions/8086/logarithm-math-operation-in-solidity>

⁸according to independent computations

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into IEXEC.

Scheduler has indisputable decision power

Since the calculation of the worker consensus for the result of an order is done completely off-chain, only the scheduler decides on-chain what result is considered correct. This happens when the scheduler calls `revealConsensus` and passes the argument `_consensus` in the `Workerpool` contract.

The consensus protocol can be audited after the fact, in particular after observing the event `FinalizeWork` for a work order anyone can be certain that no further events, payments, or state changes will happen to this order. However, there is no mechanism for the workers to dispute the chosen outcome. Hence, if a scheduler is malicious, correct results would have been computed and revealed by the workers, but they still lose their stake and reputation with no direct punishment for such a scheduler.

Even if the consensus calculation is too expensive to be put on-chain and executed for each order, there could be a separate contract doing the computation. While this would be costly, it could be called by somebody not agreeing who would pay the gas, and who would get rewarded if indeed the scheduler took the wrong decision. There may also be other (better) solutions.

Likelihood: Medium

Impact: High

Scheduler is incentivized to maximize the number of workers

In `distributeRewards()`, the scheduler is awarded the default reward as set by `schedulerRewardRatioPolicy`, plus whatever residue accrues from rounding down during the calculation of the worker rewards:

```
workerReward = workersReward.mulByFraction(m_contributions[_woid][w].weight, totalWeight)
```

WorkerPool.sol

If the number of tokens is much larger (i.e. several orders of magnitude) than the number of workers, the residue is only a relatively small part of the scheduler's reward. For example, if the total reward is 100 000 tokens and there are 100 workers, the residue will be between 0 and 99 tokens, which is small compared to the scheduler's default reward of 1000 tokens (if `schedulerRewardRatioPolicy == 1`).

However, if the reward is only 10,000 tokens or if there are 1000 workers, the residue could be almost as large (99 percent) as the scheduler's default reward. If the reward is only 10,000 and there are 1000 contributing workers, the residue could be several times as large as the default scheduler reward. This means that for smaller rewards, the residue represents a bigger part of the scheduler's reward, and the scheduler will be incentivized to maximize the number of contributing workers (i.e. more than might be needed to form a good consensus) in order to increase the size of the residue.

Likelihood: Medium

Impact: Medium

SGX-Related Trust Issues

IEXEC encourages the use of Intel's Software Guard Extensions (SGX) as they can help to protect the confidentiality of both dataset and application.

With regards to the use of SGX, there are the following trust issues:

1. *Trust into SGX and Intel:* Especially, in the light of recent low-level CPU issues, including Meltdown and Spectre, all involved parties have to trust the SGX architecture to be correct and not to be vulnerable to relevant side channel attacks. Furthermore, they need to trust Intel, which is in charge of key generation and key management.
2. *Trust into enclave:* The enclave which is holding the private key associated with IEXEC's SGX challenge has to be designed in such a way that the private key can not be exported or leaked. This property needs to be verified by all parties. If the key could be extracted, the verification system would collapse.
3. *Scheduler needs to correctly attest SGX:* All workers need to trust the scheduler to correctly attest the SGX. This includes the verification that the right enclave is running. If the scheduler would collude with one of the workers it could stealthily omit this attestation step.

4. *Freshness issues:* As discussed above, there are freshness-related issues with the commit-reveal process of IEXEC's consensus. If a result appears multiple times the previous signature of an SGX can be reused.

Likelihood: Low

Impact: Medium

Post-Audit Fix: IEXEC acknowledges the difficulties related to the secure usage of SGX and the importance of a correct implementation, however notes that this is outside the scope of the PROOF-OF-CONTRIBUTION protocol.

Recommendations / Suggestions

- The `IexecLib.sol` contract contains structs that are routinely instantiated by the Hub, Market and Workerpool contracts. Considering the frequent usage, gas costs optimizations have a significant impact. We recommend reordering the `Category` struct by placing the `string` name and `string` description fields first, followed by `uint256` `catid` and `uint256` `workClockTimeRef` saving roughly 50000 gas on transaction and deployment costs per struct.
- `allowWorkersToContribute()` and `claimFailedConsensus()` in `WorkerPool.sol` will also result in deadlock if the number of contributing workers is too high (266 in this case. However, it would be best to stay under 200, in order to have a 25 percent margin of safety). This is because they both contain loops performing expensive storage operations.
- `tx.origin` should only be used when strictly necessary. In the few places where `IEXEC` uses it, it seems to only be an implicit argument, which makes the code harder to reason about for no real gain. The functions in which it is used, the create functions, could also be changed completely: `IexecHub.sol` only provides public interfaces to functions implemented elsewhere; these interfaces are currently only meant to enforce that a certain `marketplaceAddress` is used.

```
function createWorkerPool(
    string _description,
    uint256 _subscriptionLockStakePolicy,
    uint256 _subscriptionMinimumStakePolicy,
    uint256 _subscriptionMinimumScorePolicy)
external returns (address createdWorkerPool)
{
    address newWorkerPool = workerPoolHub.createWorkerPool(
        _description,
        _subscriptionLockStakePolicy,
        _subscriptionMinimumStakePolicy,
        _subscriptionMinimumScorePolicy,
        marketplaceAddress
    );
    emit CreateWorkerPool(tx.origin, newWorkerPool, _description);
    return newWorkerPool;
}
```

IexecHub.sol

```
function createWorkerPool(
    string _description,
    uint256 _subscriptionLockStakePolicy,
    uint256 _subscriptionMinimumStakePolicy,
    uint256 _subscriptionMinimumScorePolicy,
    address _marketplaceAddress)
external onlyOwner /*owner == IexecHub*/ returns (address createdWorkerPool)
{
    // tx.origin == owner
    // msg.sender == IexecHub
    // At creating ownership is transferred to tx.origin
    address newWorkerPool = new WorkerPool(
        msg.sender, // iexecHubAddress
        _description,
        _subscriptionLockStakePolicy,
        _subscriptionMinimumStakePolicy,
        _subscriptionMinimumScorePolicy,
        _marketplaceAddress
    );
    addWorkerPool(tx.origin, newWorkerPool);
    return newWorkerPool;
}
```

WorkerPoolHub.sol

These interfaces could be removed, while the create functions could instead use the `marketplaceAddress` which would be retrieved directly from the owner (variable `m_owner`), considered as an `IexecHub`.

- Some assumptions are made about the computation costs, but they should be tested more thoroughly. In particular, writing to the storage is expensive, a lot more than simple computations. Comments seem to indicate that the `workerWeight` variable is stored into the contribution to avoid recomputations, but if this is to save gas, this is misguided.

```
workerBonus = (c.enclaveChallenge != address(0)) ? 3 : 1; // TODO: bonus sgx = 3 ?
workerWeight = 1 + c.score.mul(workerBonus).log();
totalWeight = totalWeight.add(workerWeight);
c.weight = workerWeight; // store so we don't have to recompute
```

WorkerPool.sol

- Some variables are unused, and see to be in the contracts only to keep a log of events. This functionality is better served by proper events, which can be emitted cheaply. Examples of such variables are `stdout`, `stderr` and `m_uri` in the `WorkOrder` contract.
- In `MarketplaceAccessor.sol`, the variables `marketplaceAddress` and `marketplaceInterface` are effectively the same. `address(marketplace)` can be used instead. The same applies to `IexecHub.sol` and `IexecHubAccessor.sol`.
- In `WorkerPool.sol`, `finalizeWork` could use an `assert` to guarantee that `consensus.winnerCount` is never null. `assert` that never fails are cheap, so gas is not an issue for them.
- `removeWorker` can be optimized in terms of gas consumption:
 - The storage writes could be omitted if the removed worker was also the last worker.
 - `m_workerIndex[worker]` could be cleared.

Open Questions

- Each order currently requires deploying a smart contract along with its code. Could `structs` be used instead?
- Overall, a lot of state is kept in many different contracts. Could some of this state be reset, i.e. set to 0, to save on gas costs?⁹
- If `seize()` is ever called without a corresponding call to `reward()` (to add the tokens that were deleted by `seize()` to a different account), then it destroys tokens (because they are deleted from `_user`'s account without being added anywhere else). Is this the intended behavior?
- It has been mentioned that, under certain assumptions, “using a confidence threshold of 90% or less would probably allow a worker to achieve consensus alone”¹⁰. It could be interesting for users to know which threshold is high enough to prevent this, even under the given assumptions.
- It seems that for the fastest category workers have only 60 seconds to reveal their contribution. Couldn't this be a problem if the scheduler intentionally calls `revealConsensus` when the network is very busy and then immediately calls `reopen` to seize the workers' stakes?
- The fact that CHAINSECURITY was able to find issues regarding arrays could be related to the small amount of testing done in that regard: tests all use less than 10 different accounts, which does not seem realistic in IEXEC's case. More testing with larger pools (for example) could highlight this kind of issues before the next audit. Are the current tests realistic in terms of what IEXEC expects from its users?



⁹See R_{sclear} in the yellow paper

¹⁰<https://medium.com/iex-ec/poco-series-2-on-the-use-of-staking-to-prevent-attacks-2a5c700558bd>

Disclaimer

UPON REQUEST BY IEXEC, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..