PUBLIC

Security Audit

of iExec's Smart Contracts

April 16, 2019

Produced for

iExec

by

CHAINSECURITY

Table Of Contents

Foreword

We would first and foremost like to thank iExec for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations and results.

– ChainSecurity

Executive Summary

The audit of the iExec smart contracts focused on verifying a set of invariants, both provided by iExec and augmented by ChainSecurity. The verification work was done by a newly developed system of ChainSecurity. Due to limitations of this system in handling the experimental pragma ABIEncoderV2, ChainSecurity complemented the audit by manually verifying selected properties. We thank iExec for their valuable feedback which helped us further improve our system.

Overall, we found that iExec employs good coding practices and has clean, well-documented code. All properties verified by ChainSecurity hold. ChainSecurity raised minor security and design issues, all of which have been fixed in the latest code commit.

The audit was limited to verification of the provided invariants and did not include a complete manual code review. It is therefore possible that unintended behaviours not covered by the invariants are present in the contracts.

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 March 11, 2019.
The corresponding Git commit is: `4025775a796fe60ed33ad66554f8b83ede5a400b`.
The latest update has been received on April 15, 2019.
The corresponding Git commit is: `4d017e2468c64067addabb42ee30e59bdf4bea82`.

File	SHA-256 checksum
./CategoryManager.sol	10808812160143f40237ecccf650542d9e7e5e03c372ed2bd0a10b1c03409a37
./Escrow.sol	93f7396362480451daf056f2a0652b3bb0c4224ec3a09bb51bda5b8337361e49
./IexecClerk.sol	b5f62769e3ccc8aa4af1e724abf593b8b82d4936e1ddcf484905e2021eb6ad98
./IexecClerkABILegacy.sol	b583bdaa632e4920065fc3ed149f63c262b34bae641d6be386dd074f52e31321
./IexecHub.sol	54114b013626318b281145f43edf1fb74c581750dfc1841fd6edda7062ef1f1f
./IexecHubABILegacy.sol	6cbc7505d609d5618096f1900015155317df9a60c78db3bc0283fd9fb3958376
./IexecHubAccessor.sol	38588dd62bc65ded6b55dda161d86611678412597fcf8ffe046f2e1b11a20e35
./IexecHubInterface.sol	9a8049fbece8ceb1f5d1543fea899ad6badee50f1910d059ff2316aae4044bf2
./SignatureVerifier.sol	031d3a152e1a4eedc4d6b323266722e97c2e56996d5339ea62b4722ec3757659
./libs/IexecODBLibCore.sol	b9ec4c3e149784f14c81a88676e9b90ce528b532aaeeb9be51c6b4b0f03dd092
./libs/IexecODBLibOrders.sol	2a724aae8f0168e59e0f64810c0f6e9fba9c84fd57fcf99f9350bbd5ed4b0b73
./registries/App.sol	d34cd0e94f366f2d78ec59c88454c433381d74f51a9ca5008ba6fb3c2ae6d09a
./registries/AppRegistry.sol	892723dfd2aa4035127c30c9fad6cdd687dd3808df21b104d140f64412acab6a
./registries/Dataset.sol	610f13ce6b2ce8de91b525eff7b353ba87a944fdd0e0f767e5a9e072bda6454e
./registries/DatasetRegistry.sol	631c57dd775a86f89ad3f7232588072456830c9831ab747214b1f4cd92dfc6c1
./registries/RegistryBase.sol	78bac1c1f7edce32bc58ab4e2f18d6547185c1e4f46126f975f8f524f9a30af5
./registries/Workerpool.sol	a71a3c6d7d6a82e0b3883f9e1e6763b4155f77deda540585ff44f659689b07c5
./registries/WorkerpoolRegistry.sol	e0c27d5c87c704781633a2cfa6431b6d03a8b03ed9281a20943bbd1aeef433dd

Depth of Audit

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

- Manual verification of the system invariants.
- The invariants were checked only on the respective files. Majorly in `IexecHub` and `IexecClerk` contracts.

Terminology

For the purpose of this audit, ChainSecurity has adopted the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

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.

[1] `https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology`

We categorise the findings into four distinct categories, depending on their severities:

- L Low: can be considered less important
- M Medium: should be fixed
- H High: we strongly suggest fixing it before release
- C Critical: needs to be fixed before release

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

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit, concerns might arise or tools might flag certain security issues. After carefully inspecting the potential security impact, we assign the following labels:

- ✓ No Issue : no security impact
- ✓ Fixed : during the course of the audit process, the issue has been addressed technically
- ✓ Addressed : issue addressed otherwise by improving documentation or further specification
- ✓ Acknowledged : issue is meant to be fixed in the future without immediate changes to the code

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

For the purpose of manually and automatically verifying the systems' invariants, different labels are used for the individual property or invariant ChainSecurity checked:

- ✓ Verified : A property/invariant is verified and holds in the system.
- ✕ Does not hold : A property/invariant does not hold in the system.

Limitations

Security auditing cannot uncover all existing vulnerabilities; even an audit in which no vulnerabilities are found is not a guarantee of a secure smart contract. However, auditing enables discovery of 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 are completely unprotected against it. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. This is why we carry out a source code review aimed at determining 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.

System Overview

IEXEC provides a fully decentralized solution where providers of applications, datasets and computational power can meet users. Due to its decentralized nature and the use of smart contracts, there is no need to rely on any one single agent.

The new version of IEXEC introduces Proof-of-Contribution (PoCo). Honest contributions are ensured by staking, because bad actors will lose their stake.

User interaction happens through the IEXEC market front-end. Users buy computational resources with specific apps and, if needed, datatsets, while worker pool owners sell computational power. Payment and staking are carried out with RLC tokens. The user creating an order can set the confidence level desired; this corresponds to a minimum correctness likelihood that the result achieves. Furthermore, enforcing execution of a task in a Trusted Execution Environment is supported. However, this trusted execution does not fall under the scope of this audit. Here, we will focus on the underlying smart contract suite.

Simplified life-cycle of a deal

Note that, in contrast to the previous version, sealing a deal, including order matching, is now done off-chain. First, an order is created off-chain, then the signature of the intended order is checked on-chain in the `IexecClerk` contract. These open orders can subsequently be matched. While the `Clerk` is responsible for matching orders, in theory anyone can do so.

Once `matchOrder` is called, this marks the beginning of the execution of a task. Funds of the requester and the scheduler (worker pool owner) are locked to ensure that these parties make honest contributions. The consensus timer starts and the corresponding task must be completed before the final deadline. If a task is not completed by this time, the scheduler is punished and the requester gets their funds back. Thus, the scheduler has an incentive to ensure the task is completed on time. To start the execution, the scheduler initializes the task in `IexecHub` and designates workers to participate. Only these designated workers may participate, to ensure this, the scheduler signs and shares messages with these workers' addresses, which are later cross-checked. Contributing workers now compute the requested task and, upon completion, call `contribute`.

Only the designated workers may do so and they have to lock a certain stake to ensure honest contribution. As soon as the contributions exceed the consensus level needed by the task, the task transitions from `ACTIVE` to `REVEALING`. Contributors need to reveal some parameters in order to prove they actually computed the parameters contributed. After the `REVEAL` phase is over, the scheduler calls finalize. Successful contributions are rewarded and locked stakes are released.

There is functionality for the scheduler to reset the contribution phase. This can be done only when the task's final deadline is not reached and there is still some time left before the reveal deadline. The scheduler can only reset the contribution phase when no workers have revealed their results. When a task is reset, it is again open for workers to contribute their results.

IexecClerk

Contains functionality to sign a deal and match orders. Also includes an `Escrow` which is responsible for storing users' funds safely.

IexecHub

Handles all functionality during processing of a task from initializing to finalizing.

Apps and Datasets

Anyone may provide Applications that can do certain computations (e.g. image processing) or datasets and earn RLC tokens. These must first be approved by IEXEC.

Roles

Scheduler The owner of a worker pool. Coordinates the workers to complete a task and is responsible for leading the process in `IexecHub`.

Requester Any user requesting that a task be completed.

Worker Any worker inside any worker pool. They contribute computing power to the network.

Clerk Responsible for matching the orders.

Application provider Application providers or developers can monetize their applications and algorithms by setting a fixed fee for each single instance of software use.

Dataset provider They own valuable datasets and can make them available for use by applications.

Best Practices in iExec's project

Good-quality projects follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfilment of general guidelines.

Avoiding code duplication is a good example of good engineering practice which increases the potential of any security audit.

We would now like to list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when iExec's project meets the criterion when the audit started.

Hard Requirements

These requirements ensure that the iExec's project can be audited by ChainSecurity.

- ✓ The code is provided as a Git repository to allow reviewing of future code changes.
- ✓ Code duplication is minimal, or justified and documented.
- ✓ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with iExec's project. No library files are mixed with iExec's own files.
- ✓ The code compiles with the latest Solidity compiler version. If iExec uses an older version, the reasons are documented.
- ✓ There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to iExec.

- ✓ There are migration scripts.
- ✓ There are tests.
- ✓ The tests are related to the migration scripts and a clear distinction is made between the two.
- ✓ The tests are easy to run for ChainSecurity, using the documentation provided by iExec.
- ☐ The test coverage is available or can be obtained easily.
- ✓ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ✓ The project only contains audit-related files, or, if this is not possible, a meaningful distinction is made between modules that have to be audited and modules that ChainSecurity should assume are correct and out-of-scope.
- ✓ There is no dead code.
- ✓ The code is well-documented.
- ✓ The high-level specification is thorough and enables quick understanding of the project without any need to look at the code.
- ✓ Both the code documentation and the high-level specification are up-to-date with respect to the code version ChainSecurity audits.
- ✓ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☐ Functions are grouped together according either to the Solidity guidelines[2], or to their functionality.

[2] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Verified Properties

CHAINSECURITY investigated selected customer-specific invariants manually. The verified and violated properties are listed below:

Properties related to the `IexecClerk` contract

The properties for the `IexecClerk` contract, which are verified manually, are outlined below.

1.1 Deposits increases `stake` ✓ Verified

In IEXEC contracts, each entity has to stake some RLC tokens in the `IexecClerk` contract. Staking is performed using deposit functions. Depositing an RLC token using deposit-specific functions present in `IexecClerk` increases the `stake` of a user. The functions that directly increase the `stake` of the function call are as follows:

- `IexecClerk.deposit()`
- `IexecClerk.depositFor()`
- `IexecClerk.depositForArray()`

The user calling the above function provides the number of RLC tokens to the function calls, e.g. the number of tokens is X, then the stake will be increased by X amount.

A user can increase their own `stake` by calling the `IexecClerk.deposit()` function and a user can increase their or someone else's `stake` by calling the `IexecClerk.depositFor()` or `IexecClerk.depositForArray()` function.

1.2 `stake` increases indirectly ✓ Verified

There are some process-specific indirect function calls that increase the `stake` of any passed-in account. The functions that indirectly increase the `stake` amount are as follows:

- `IexecClerk.unlockContribution()`
- `IexecClerk.unlockAndRewardForContribution()`
- `IexecClerk.successWork()`
- `IexecClerk.rewardForScheduling()`
- `IexecClerk.failedWork()`

1.3 Withdraw decreases `stake` ✓ Verified

An entity is allowed to withdraw their RLC token stake from the `IexecClerk` contract. This process decreases the `stake` of a particular user by X amount, whereby X is the amount that the user provided in the `IexecClerk.withdraw()` function call. A user is only allowed to withdraw their own staked RLC token and decreases the `stake` amount.

1.4 `stake` decreases indirectly ✓ Verified

There are some process-specific indirect function calls that decrease the `stake` of any passed-in account. This can be performed by calling the `IexecClerk.lockContribution()` function.

1.5 Deposit increases `IexecClerk` contract RLC token balance ✓ Verified

An entity deposits an RLC token to the `IexecClerk` contract to increase staking. This process increases the `stake` for that entity and increase the RLC token balance of the `IexecClerk` contract.

The contract provides specific functions for depositing the RLC token to the contract:

- `IexecClerk.deposit()`
- `IexecClerk.depositFor()`

- IexecClerk.depositForArray()

However, direct transfer of the RLC token to the IexecClerk would also increase the RLC token balance for the IexecClerk contract:

- RLC.transfer()
- RLC.transferFrom()

The only ways to increase the RLC token balance are outlined above. There are no other possible ways to increasing the RLC token balance of an IexecClerk contract.

1.6 Withdraw decreases IexecClerk contract RLC token balance ✓ Verified

An entity can withdraw its portion of the stake in RLC tokens from the IexecClerk contract. The call to IexecClerk.withdraw() reduces the stake of the caller and transfers the requested amount of RLC tokens to its address. This process reduces the RLC token balance from the IexecClerk contract.

This is the only possible way of reducing the balance of the RLC token in the IexecClerk contract.

1.7 Value moves from stake to locked ✓ Verified

An entity's RLC token stake is locked for it to perform certain actions. The same amount is deducted from stake and increased in locked. This is performed using the following function calls:

- IexecClerk.lockContribution() function internally calls IexecClerk.lock() and moves the value from stake to locked for a given account.
- IexecClerk.matchOrders() function internally calls IexecClerk.lock() for the requester and scheduler, which moves values from stake to locked.

The only two function calls that deduct the X amount from stake and increase the X amount in locked are outlined above.

1.8 Value moves from locked to stake ✓ Verified

After some operations, an entity's RLC token stake is unlocked. When unlocking takes place, the same amount is deducted from locked and increased in stake for the specific account. This is performed using the following function calls:

- IexecClerk.unlockContribution() function internally calls IexecClerk.unlock() and moves the value from locked to stake for a given account.
- IexecClerk.failedWork() function internally calls IexecClerk.unlock() for the requester account, which reduces the amount for locked and increases it for stake.

The only two function calls that deduct the X amount from locked and increase the X amount in stake are outlined above.

However, there are still some functions that move an amount from locked to stake. But the amount is not the same, there would be a slightly increased amount for stake as the Reward Kitty is also credited while the following function calls are being performed:

- IexecClerk.successWork() function internally calls IexecClerk.unlock() for the scheduler account, which reduces the same amount in locked and increases it in stake, but at the end it checks for the Reward Kitty; if it is present in the pool, a portion of the Reward Kitty is also given to the scheduler. This process increase their stake balance further.
- IexecClerk.unlockAndRewardForContribution() function internally calls IexecClerk.unlock() and IexecClerk.reward() functions. The unlock() call would reduce the same amount from locked and increases it in stake. However, the reward() call would further increase the stake balance of the given account.

1.9 Reward kitty increases upon failedWork call ✓ Verified

The reward kitty is a special account in which RLC tokens are credited when a task is failed. Once a task has failed consensus, the stake of the pool worker (scheduler) of that task is seized and credited to the reward kitty address. The reward kitty account is tracked using address(0) address.

Once IexecClerk.failedWork() is called, the reward kitty account (address(0)) is staked with poolstake and further locked using the IexecClerk.lock() call.

It is not possible to deposit RLC tokens in the IexecClerk contract for the reward kitty, as IexecClerk.depositFor checks for address(0) and reverts the transaction. Hence, the reward kitty only increases using IexecClerk.failedWork().

1.10 Deal remains unchanged after creation ✓ Verified

Anyone is allowed to call the IexecClerk.matchOrders() function. A deal is created when the function IexecClerk.matchOrders() is called. Once it has been created, the deal will never be changed.

1.11 IexecClerk.token never changes ✓ Verified

Upon creation of the IexecClerk contract, the RLC token address is passed to the constructor. The address gets stored in IexecClerk.token and is never updated by the contract. Hence, the value will never change once set by the constructor.

1.12 IexecClerk.IexecHub never changes ✓ Verified

Upon creation of the IexecClerk contract, the IexecHub contract address is passed to the constructor. The address gets stored in IexecClerk.IexecHub and is never updated by the contract. Hence, the value will never change once set by the constructor.

1.13 IexecClerk.EIP712DOMAIN_SEPARATOR never changes ✓ Verified

Upon creation of the IexecClerk contract, the hash of some fixed parameters is calculated and gets stored in the IexecClerk.EIP712DOMAIN_SEPARATOR state variable. The value is never updated by the contract. Hence, the value will never change once set by the constructor.

1.14 Only IexecHub is allowed to call ✓ Verified

In the IexecClerk contract, there are a few functions that can only be called by the IexecHub contract. These functions are:

- IexecClerk.lockContribution()
- IexecClerk.unlockContribution()
- IexecClerk.unlockAndRewardForContribution()
- IexecClerk.seizeContribution()
- IexecClerk.rewardForScheduling()
- IexecClerk.successWork()
- IexecClerk.failedWork()

The above functions are restricted with the onlyIexecHub modifier, which checks the msg.sender address with IexecHub. The IexecHub is never changed once it has been set by the constructor.

1.15 m_presigned is only updated once ✓ Verified

When an App order, a Dataset order, a Worker pool order and a Request order are signed by the corresponding entity, it updates m_presigned mapping. Once a key present in m_presigned mapping is set to true, it never resets back to false.

1.16 Scheduler stake is locked ✓ Verified

A scheduler's RLC token stake is locked once a deal has been created. This process locks the configured percentage of the worker's pool price from the schedule's RLC token stake.

1.17 No-one is allowed to withdraw from the reward kitty ✓ Verified

The reward kitty account is maintained using a special address (address(0)). The functions that change the reward kitty are as follows:

- IexecClerk.successWork(): The function takes a portion of the amount and rewards it to deal.workerpool.owner. This function can only be called from IexecHub.finalise, which can only be called by the scheduler.
- IexecClerk.failedWork(): The function only increases the reward kitty.

However, there is a way to directly withdraw an RLC token from the reward kitty.

1.18 No-one is allowed to directly deposit to the reward kitty ✓ Verified

The reward kitty account is maintained using a special address (`address(0)`). The function `IexecClerk.depositFor` checks for the `address(0)` when depositing the RLC token. Hence, none of the functions `depositFor` and `depositForArray` deposits to the reward kitty account.

1.19 Reward kitty is not burned ✓ Verified

The reward kitty account is maintained using a special address (`address(0)`).

- The reward kitty increases when work has failed; the scheduler's amount is credited to the reward kitty account.
- The portion of amount from the reward kitty is distributed to the scheduler upon `successWork`.

Hence, the reward kitty amount is never burned.

Properties related to the `IexecHub` contract

The properties for the `IexecHub` contract, which are verified manually, are outlined below:

2.1 Contribution status can only change in a specific order ✓ Verified

The contribution status of a task along with its worker is changed only in a specific order using the following function calls:

- The contribution status changes only from `UNSET` to `CONTRIBUTED` when the `IexecHub.contribute()` function is called.
- The contribution status changes only from `CONTRIBUTED` to `PROVED` when the `IexecHub.reveal()` function is called.
- The contribution status changes only from `CONTRIBUTED` to `REJECTED` when the `IexecHub.reopen()` function is called.

2.2 Task status can only change in a specific order ✓ Verified

The task status of a task can change only in a specific order using the following function calls:

- The task status changes only from `UNSET` to `ACTIVE` when the `IexecHub.initialise()` function is called.
- The task status changes only from `ACTIVE` to `FAILED` when the `IexecHub.claim()` function is called.
- The task status changes only from `ACTIVE` to `REVEALING` when the `IexecHub.contribute()` function is called.
- The task status changes only from `REVEALING` to `ACTIVE` when the `IexecHub.reopen()` function is called.
- The task status changes only from `REVEALING` to `FAILED` when the `IexecHub.claim()` or `IexecHub.claimArray()` function is called.
- The task status changes only from `REVEALING` to `COMPLETED` when the `IexecHub.finalise()` function is called.

2.3 When a task status is `UNSET` it must be `UNSET` previously ✓ Verified

If a task status is `UNSET` then it was previously `UNSET`. The `UNSET` status is set by default and never set again from any of the functions.

2.4 Contribution status changes from `CONTRIBUTED` to `PROVED` only when the task status is `REVEALING` ✓ Verified

Once contributions by the workers are complete and consensus is achieved, the workers call the `IexecHub.reveal` function. This function changes the contribution status from `CONTRIBUTED` to `PROVED` for a worker's contributions. To call this function, the task status must be in the `REVEALING` state.

2.5 Values must be set, when a task in `REVEALING` state ✓ Verified

When a task status changes from `ACTIVE` to `REVEALING`, some of the variables of that task must be set and the conditions below must hold:

- `IexecHub.m_tasks[task_id].consensusValue != 0`
- `IexecHub.m_tasks[task_id].revealDeadline != 0`
- `IexecHub.m_tasks[task_id].winnerCounter != 0`
- `IexecHub.m_tasks[task_id].revealCounter == 0`

2.6 Only the owner is allowed to call `IexecHub.attachContracts()` ✓ Verified

The function `IexecHub.attachContracts()` is called by the owner of the contract to attach the other contracts to the `IexecHub` contract. The function is only allowed to be called by the current owner of the `IexecHub` contract. However, once the `IexecClerk` address is set to a non-zero (`0x0`) address, the call to `attachContracts()` is not allowed again.

2.7 Active tasks have certain variables set and some unset ✓ Verified

When a task is in the `ACTIVE` state, some of its variables must be set to to non-zero values and some must be set to zero or the default value (`0x0`). In this case, the following holds:

- `IexecHub.m_tasks[task_id].dealid != 0`
- `IexecHub.m_tasks[task_id].idx != 0`
- `IexecHub.m_tasks[task_id].timeref != 0`
- `IexecHub.m_tasks[task_id].contributionDeadline != 0`
- `IexecHub.m_tasks[task_id].finalDeadline != 0`
- `IexecHub.m_tasks[task_id].consensusValue == 0x0`
- `IexecHub.m_tasks[task_id].winnerCounter == 0`
- `IexecHub.m_tasks[task_id].contributors.length == 0`
- `IexecHub.m_tasks[task_id].revealDeadline == 0`
- `IexecHub.m_tasks[task_id].results == 0`

2.8 Result variables set when contribution status is not `UNSET` ✓ Verified

When a specific contribution's status is not in the `UNSET` state, then its result-specific fields must have non-zero values set. In this case, the following holds:

- `IexecHub.m_contributions[task_id][worker].resultHash != 0x0`
- `IexecHub.m_contributions[task_id][worker].resultSeal != 0x0`

2.9 Task's `finalDeadline` greater than `contributionDeadline` ✓ Verified

When a task is initialised, two deadlines are set for that particular task. The task's `finalDeadline` is always greater than the `contributionDeadline`.

2.10 Only `initialise` creates a task ✓ Verified

When the `IexecHub.initialise()` function is called with the `dealid` and `idx` parameters that have not been used before, it would create and initialise a new task. The status of the task would be set to `ACTIVE`. It also sets the following variables of a task:

- `IexecHub.m_tasks[0x1].dealid`
- `IexecHub.m_tasks[0x1].idx`
- `IexecHub.m_tasks[0x1].timeref`
- `IexecHub.m_tasks[0x1].contributionDeadline`
- `IexecHub.m_tasks[01].finalDeadline`

2.11 `initialise` never overwrites an existing task ✓ Verified

Once a task is initialised using the `initialise` function call, it is not overwritten again. This is prevented because, for the initialised task, its status is changed from `UNSET` to `ACTIVE`. Hence, it is not possible to re-initialise an existing task.

2.12 Only the current owner can change the owner ✓ Verified

The `IexecHub` indirectly inherits from the `Ownable` contract. Hence, the `owner` variable can be changed to new owner. The current owner is the only address that can set the new owner address by calling the `IexecHub.transferOwnership()` function. They can also call the `IexecHub.renounceOwnership()` function to leave the ownership permanently.

2.13 Only reveal increases a task's `revealCounter` ✓ Verified

When the `IexecHub.reveal()` function is executed by the worker to reveal their work result, the function increases the `IexecHub.m_tasks[task_id].revealCounter` by one to update the count of the results that have been revealed.

2.14 `reopen` resets some fields ✓ Verified

When the `IexecHub.reopen()` function is called, it changes the state of the provided task from `REVEALING` to `ACTIVE`. It also resets some of the task's variables:

- `IexecHub.m_tasks[task_id].consensusValue == 0x0`
- `IexecHub.m_tasks[task_id].revealDeadline == 0`
- `IexecHub.m_tasks[task_id].winnerCounter == 0`

2.15 The `locked` value decreases ✓ Verified

The `locked` value decreases for an account when the following functions are called:

- `IexecHub.finalise()`: The function internally calls `IexecClerk.successWork` which `seizes` the `deal.requester`'s stake and `unlocks` the `deal.workerpool.owner` stake, which in tern decreases the `locked` value.
  - Also, this function internally calls `IexecHub.distributeRewards`, which further calls `IexecClerk.seizeContribution` and decreases the locked value.
  - This function also calls `IexecClerk.unlockAndRewardForContribution` and decreases the locked value.
- `IexecHub.claim()`: The function internally calls `IexecClerk.failedWork` which `seizes` the token from the `deal.workerpool.owner`'s stake. It also `unlocks` the `deal.requester` stake.
  - This function also makes a call to `IexecClerk.unlockContribution`, which also decreases the `locked` value.

Properties related to both `IexecHub` and `IexecClerk` contracts

The following are the properties for `IexecHub` and `IexecClerk` contracts that are verified manually:

3.1 The `locked` value increases ✓ Verified

The `locked` value increases for an account when the following functions are called:

- `IexecClerk.matchOrder()`: Function locks the stake for `deal.requester` and `deal.workerpool.owner`.
- `IexecHub.contribute()`: Function calls `IexecClerk.lockContribution` which locks the stake.
- `IexecHub.claim()`: Function internally calls `IexecClerk.failedWork` which increases the `locked` for the reward kitty.

3.2 The `stake` value increases ✓ Verified

The `stake` value increases for an account when the following functions are called:

- `IexecClerk.deposit()`: The function deposits an RLC token and increases the `stake`.
- `IexecClerk.depositFor()`: The function deposits an RLC token for a specific address and increases the `stake`.
- `IexecClerk.depositForArray()`: The function deposits RLC tokens for specific addresses and increases the `stake` for each of them.
- `IexecHub.finalise()`: The function internally calls `IexecClerk.successWork`, which unlocks and rewards. This increases the `stake` of `deal.workerpool.owner`.
- `IexecHub.claim()`: The function internally calls `IexecClerk.failedWork`, which `unlocks` the `deal.requester` and increases their `stake`.

3.3 The `stake` value decreases ✓ Verified

The `stake` value decreases for an account when the following functions are called:

- `IexecClerk.withdraw()`: The function decreases the `stake` as it withdraws RLC tokens from the contract.
- `IexecClerk.matchOrder()`: The function locks the `stake` and decreases it.
- `IexecHub.contribute()`: The function internally calls `IexecClerk.lockContribution` which decreases the `stake` and locks it.

Security Issues

This section relates to our investigation into security issues. It is meant to highlight times when we found specific issues, but also mentions what vulnerability classes do not appear, if relevant.

Old version of Solidity compiler used M ✓ Fixed

In many of the contracts, the compiler experimental feature `pragma experimental ABIEncoderV2` is used. The Solidity version 0.5.7[3] has some important bug fixes related to `ABIEncoderV2`.

ChainSecurity recommends that the latest version of the Solidity compiler `0.5.7` should be used.

Likelihood: Medium

Impact: Medium

Fixed: iExec fixed the problem by upgrading to solidity compiler version `0.5.7`. iExec is aware about this issue and waiting for native support for `ABIEncoderV2` in solidity compiler version `0.6.0`.

[3] https://solidity.readthedocs.io/en/v0.5.7/bugs.html

Trust Issues

This section mentions functionality that is not fixed inside the smart contract and hence requires additional trust in IEXEC, including in IEXEC's ability to deal with such powers appropriately.

CHAINSECURITY has no concerns to raise in this category of the report.

Design Issues

This section lists general recommendations about the design and style of iExec's project. These recommendations highlight possible ways for iExec to improve the code further.

Unused `EIP712DOMAIN_SEPARATOR` variable L ✓ Acknowledged

The `IexecClerk` and `IexecClerkABILegacy` contracts both have an `EIP712DOMAIN_SEPARATOR` state variable defined.

However, the `IexecClerk` contract inherits from `IexecClerkABILegacy`, hence the `EIP712DOMAIN_SEPARATOR` state variable is defined twice in the `IexecClerk` contract.

ChainSecurity recommends to avoid duplication of variables.

Acknowledged: iExec has written client application using Web3j Java library. The library does not support `ABIEncoderV2` compiled ABI and crashes the client. Hence, it is the workaround to have support for Web3j.

Disclaimer