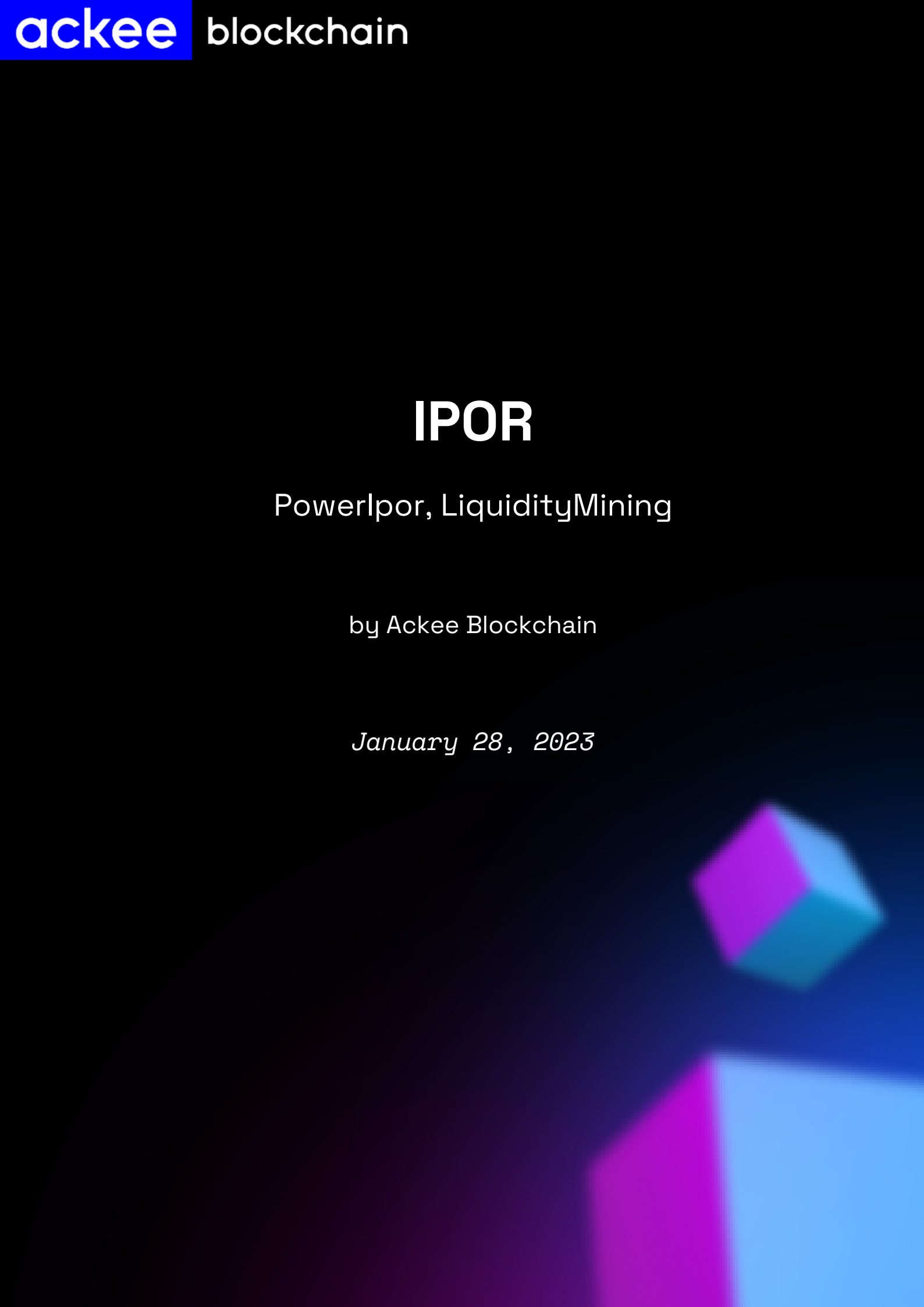
ackee blockchain

IPOR

PowerIpor, LiquidityMining

by Ackee Blockchain

January 28, 2023



Contents

1. Document Revisions

1.0	Final report	November 9, 2022
1.1	Fix review 1.1 report	November 21, 2022
1.2	Fix review 1.2 report	December 23, 2022
1.3	Protocol naming update	January 27, 2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Medium	-
	Low	Medium	Medium	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Lukáš Böhm	Lead Auditor
Miroslav Škrabal	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

IPOR (Inter-Protocol Offered Rate) protocol works as a weighted index average of several different borrowing and lending sources. Handling and selecting the most relevant sources would be done via IPOR Decentralized Autonomous Organization (DAO) to achieve a complete decentralized system. The transparent mathematical formulas calculate a weighted average.

Revision 1.0

IPOR team engaged Ackee Blockchain to perform a security review of the Ipor protocol parts, specifically `IporToken` and Ipor mining (`LiquidityMining` and `PowerIpor` contracts), within a period between October 17 and November 9, 2022 and the lead auditor was Lukáš Böhm. The audit has been performed on the commit `01c08c3`. At the client's request, the report was divided into two parts. This report covers `LiquidityMining` and `PowerIpor` contracts.

We began our review using static analysis tools, namely Slither and Woke. We then took a deep dive into the logic of the contracts. During the review, we paid particular attention to:

- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation,
- ensuring the token handling logic is correct.

After the manual review of the core codebase, we moved our attention to the mathematical libraries, specifically ABDK library for quadruple precision. For this part of the audit, we implemented differential fuzz tests to observe the behavior of the mathematical functions under randomized conditions. More

information about the tests can be seen in appendix B. During the tests, we spot inconsistencies in the results of logarithmic functions. For some more edge case inputs, the outputs of ABDK `log_2()` function differ in some less significant bits from the results of the Python math Bigfloat library. We had not enough time to debug the function to find the root cause of the issue as this was not the main objective of the audit. Even though the function usually behaves as expected, and we cannot describe the issue with a hundred percent confidence, we consider it necessary to discuss the issue with the IPOR team. We also plan to continue the investigation of the issue because the ABDK library is well-known and used across the ecosystem.

The protocol architecture is well-designed, and the code quality is above average. The code lacks in-code NatSpec documentation for some functions. Nevertheless, IPOR team provides high-quality documentation for the protocol and its components. This is very appreciated, especially for contracts with a mathematical reward logic. The team also provides a helpful diagram to understand liquidity mining mechanics better. We encourage the team to continue with a professional approach to make the project transparent, understandable, and easy to use. The project contains thousands of lines of TypeScript tests with excellent code coverage.

Our review resulted in 21 findings, ranging from Info to High severity. In the protocol, no actual thread has been found, and most of the issues are about the code performance and quality. The most severe one is a trust model and handling the ownership role (see M1: Reclaiming renounced ownership or M2: Renounce ownership risk).

Ackee Blockchain recommends IPOR:

- carefully handle the owner role,
- improve the code quality by adding NatSpec documentation,

- pay more attention to the code performance and gas usage,
- investigate further the ABDK library inconsistencies,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The fix review was done on November 21 on the given commit: `9b963ee`.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

The status of all reported issues has been updated and can be seen in the findings table. The acknowledged issue contains the client's comments.

Revision 1.2

Based on the twitter post, The Ipor team finds that the same problematic behavior can appear in the protocol. Ackee Blockchain was asked to cooperate with the investigation and fix the vulnerability. For more details see the issue detail.

The codebase was moved to the new repository `IPOR-Labs/ipor-power-tokens` and fix review 1.2 was performed on the commit `c4eeca4` on December 22, 2022.

See Revision 1.2 for the review of the updated codebase and additional information we consider essential for the current scope.

The status of all reported issues has been updated and can be seen in the findings table. The acknowledged issue contains the client's comments.

Revision 1.3

The Ipor liquidity mining protocol was changed from the standpoint of syntax; some contracts and variables were renamed and other slight cosmetical changes were introduced. The Ipor team engaged Ackee Blockchain with the request to update the report to reflect those changes. The time allocation for the review was 4 hours.

The goal of this revision was to check the changes and confirm that they introduced no semantical changes relative to the Revision 1.2 and that the previous audit is relevant even for the newer version of the protocol.

The protocol review was done on the main branch and the commit: `64e303a`.

It is important to note that no functional testing of the contracts was done, the review was performed only on the diff against the last reviewed version.

Conclusion

The changed files were examined using a diff tool and no semantical changes were discovered, i.e. the protocol should function the same as in the previous iteration.

See Revision 1.3 for the list of changes.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Solution*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: Reclaiming renounced ownership	Medium	1.0	Fixed
M2: Renounce ownership risk	Medium	1.0	Acknowledged
M3: Non-programatic approach for setting constants	Medium	1.0	Fixed
W1: Usage of `solc` optimizer	Warning	1.0	Acknowledged
I1: Unnecessary usage of post-inc	Info	1.0	Fixed

	Severity	Reported	Status
I2: Inconsistent definition of iterator variables in for loops	Info	1.0	Fixed
I3: Variables should be declared as constants	Info	1.0	Acknowledged
I4: Lack of zero-amount check	Info	1.0	Fixed
I5: Unnecessary use _msgSedner()	Info	1.0	Acknowledged
I6: Confusing function name	Info	1.0	Fixed
I7: Unnecessary variables creation	Info	1.0	Fixed
I8: Incorrect initialization pattern	Info	1.0	Fixed
I9: Usage of memory instead of calldata	Info	1.0	Fixed
I10: Reading length of an array in for loop	Info	1.0	Partly fixed
I11: Redundant use of SafeERC20 library	Info	1.0	Fixed
I12: Lack of robust contract composition	Info	1.0	Fixed
I13: Require should be assert	Info	1.0	Acknowledged

	Severity	Reported	Status
I14: The owner can prevent unstaking from LiquidityMining	Info	1.0	Fixed
I15: Code duplication	Info	1.0	Fixed
I16: Comment quality	Info	1.0	Fixed
H1: Inability to unstake when the contract runs out of rewards	High	1.2	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

LiquidityMining

LiquidityMining allows for staking `ipTokens` and for delegating `Power Ipor` tokens. Users are incentivized to staking and delegating via rewards denominated in `Ipor` tokens. The rewards are calculated from cumulative checkpoints created as users interact with the contract by staking, unstaking, and claiming the rewards.

Part of the logic is in `LiquidityMiningInternal` contract, which it inherits from. LiquidityMining is `upgradeable` and uses the UUPS upgradeability pattern. It also uses the `Ownable` pattern and allows the owner to set essential parameters like adding new tokens and setting rewards per block.

PowerIpor

PowerIpor contract is responsible for handling and managing `Power Ipor` token. The contract interacts with LiquidityMining contract and allows users to stake and delegating the token. `Power Ipor` token can be obtained by staking `Ipor` token.

MiningCalculation

The library contains the core mathematical logic for calculating rewards in the liquidity mining model.

ABDKMathQuad

The library is used inside MiningCalculation for performing mathematical operations with quadruple floating point numbers.

Actors

This part describes actors of the system, their roles, and permissions.

Owner

The system owner has a special privilege to set essential parameters and influence the system's behavior. This role can: - set the unstake fee, - set LiquidityMining contract address, - set pause manager address, - set rewards per block, - add and remove lp token assets, - upgrade the contract

Pause manager

The role has the ability to pause PowerIpor and LiquidityMining contracts.

User

Users can interact with the system by staking and delegating `Ipor`/`PowerIpor` tokens.

5.2. Trust model

In the current scope, the owners of the contracts have significant power over the system. Users have to trust the owner to not abuse the power and to not change the system in a way that would be detrimental to the users.

M1: Reclaiming renounced ownership

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	IporOwnable.sol, IporOwnableUpgradeable.sol	Type:	Access control

Description

The contracts that implement the ownable pattern implement a 2-step process to transfer ownership. In this process, the owner proposes a new owner, and the new owner accepts the proposal.

The ownership can also be renounced. In this case, the owner transfers the ownership to the zero address. When renouncing the ownership, it is essential to clear the `_appointedOwner`. This is not done in the current implementation.

Vulnerability scenario

The owner appoints a new owner. The appointed owner does not immediately accept the ownership; as time progresses, this action is eventually forgotten. After some time, the owner renounces the ownership, and the users gain the impression that the contract cannot have an owner anymore. However, the appointed owner can still accept the ownership and become the contract owner. If he does so, he can start to execute any function that the owner can execute and break the users' assumptions.

Recommendation

Override the `renounceOwnership` function and clear the `_appointedOwner` variable.

Fix 1.1

Fixed.

Go back to Findings Summary

M2: Renounce ownership risk

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	`**/*`	Type:	Trust model

Description

The contracts use `Ownable` pattern. This pattern allows for renouncing ownership to increase decentralization and lower the attack vector. However, renouncing ownership at the wrong moment can have harsh consequences. For example, it can block the upgradeability process. Therefore it has to be used only after careful consideration.

Recommendation

Handling the ownership of the contracts should be done with special care. If a malicious actor somehow gets access to the role, it can have fatal consequences over the protocol. Using multi-sig wallets is a good practice to mitigate the risk of losing contract ownership.

Fix 1.1

Client's comment:

"We are using Gnosi Multisig 4 / 6 with Timelock, so this action also will be restricted in this way. We would like to also stay with this option in case when this version of IPOR Protocol will not be maintained by DAO or IPOR Labs. In case of any Compliance and future regulation in DeFi and blockchain itself."

Go back to Findings Summary

M3: Non-programatic approach for setting constants

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Constants.sol	Type:	Code quality

Description

The library Constants sets some constants manually as literals. A programmatic approach should be preferred.

```
library Constants {
    uint256 public constant MAX_VALUE =

115792089237316195423570985008687907853269984665640564039457584007913129639
935;

    uint256 public constant MAX = type(uint256).max;
}

contract C {
  function test() public {
    assert(Constants.MAX_VALUE == Constants.MAX);
  }

}
```

Recommendation

Use the `type(uint256).max` expression instead of the literal. This approach makes the code more readable and maintainable.

Fix 1.1

Fixed.

Go back to Findings Summary

W1: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Fix 1.1

Client's comment:

"Currently we are using optimizer in already deployed IPOR Protocol smart contracts. Liquidity Mining is a part of IPOR Protocol so will be part of public repo ipor-protocol where we are using optimizer. We will monitor future issues related with Optimizer."

Go back to Findings Summary

I1: Unnecessary usage of post-inc

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMining.sol, LiquidityMiningInternal.sol, PowerIpor.sol	Type:	Gas optimization

Description

The project uses post-incrementation inside for loop headers. This is unnecessary and semantically identical to pre-incrementation. It is recommended to use pre-incrementation instead because it is more gas efficient and semantically equivalent

Locations

- LiquidityMining.sol/27,
- LiquidityMiningInternal.sol/79,132,183,248,
- PowerIpor.sol/125, 154, 186.

Recommendation

Replace the post-incrementation with pre-incrementation. Bare in mind that this approach cannot be carelessly used in all cases. Sometimes this could lead to a program's semantics change (but this is not the case for the for loops).

Fix 1.1

Fixed.

Go back to Findings Summary

I2: Inconsistent definition of iterator variables in for loops

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol, PowerIpor.sol	Type:	Code quality

Description

There are inconsistencies in definitions of the for loop iterator variables. Some are defined as `uint256 i;` and some as `uint256 i = 0;`. This impairs readability.

Locations

- LiquidityMiningInternal.sol/248,
- PowerIpor.sol/186.

Recommendation

Pick a unique style and follow it consistently. Because the style of setting the value to 0 is more common in the code, it is recommended to use it everywhere.

Fix 1.1

Fixed.

Go back to Findings Summary

I3: Variables should be declared as constants

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol	Type:	Gas optimization

Description

The shift functions in LiquidityMiningInternal return a constant-literal value.

```
function _getHorizontalShift() internal pure returns (bytes16) {
    return 0x3fff0000000000000000000000000000;
}

function _getVerticalShift() internal pure returns (bytes16) {
    return 0x3ffd99999999999999e36310e0e2a848;
}
```

Therefore from the semantical standpoint, they behave like constants.

Recommendation

Declare the shift variables as constants.

Fix 1.1

Client's comment:

"This is by design. Here is missing "virtual". We wanted to use that approach to have possibility to override that function in ITF Smart Contracts tailored for tests which are deployed in private testnets where IPOR Labs use ITF (ITF - IPOR Test Framerowk - separate IPOR Labs product for backtesting IPOR Protocol models)."

Go back to Findings Summary

I4: Lack of zero-amount check

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMining.sol	Type:	Data validation

Description

In LiquidityMining `unstake` function, there is no check for zero amount input. Unstaking zero amount does not cause harm, but it rebalances indicators and wastes unnecessary gas.

Recommendation

Add a simple requirement, and revert the transaction if zero amount is passed.

Fix 1.1

Fixed.

Go back to Findings Summary

I5: Unnecessary use _msgSedner()

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Gas optimization

Description

Across the project, the abstract contract `ContextUpgradable` is used. It provides view `_msgSender()` function that returns `msg.sender`. This approach is functional when the function is overridden for some exceptional cases where a different address than `msg.sender` should be returned. However, this function is not overridden in the project and is used in the same way as `msg.sender` may be used.

Recommendation

Classic `msg.sender` is realized via one instruction, `_msgSender()` invokes the whole machinery of calling an internal function. If there is no plan to override the function, it is recommended to use `msg.sender` instead of `_msgSender()` for gas efficiency.

Fix 1.1

Client's comment:

"This is by design. We wanted to use that approach to have possibility to override that function in ITF Smart Contracts tailored for tests which are deployed in private testnets where IPOR Labs use ITF (ITF - IPOR Test Framerowk - separate IPOR Labs product for backtesting IPOR Protocol models)."

Go back to Findings Summary

I6: Confusing function name

Impact:	Info	Likelihood:	N/A
Target:	PowerIpor.sol	Type:	Code quality

Description

While unstaking from PowerIpor, fifty percent of the staked amount is returned to the user. The amount is calculated by the private function `_calculateAmountWithoutFee`. The function's name may sound confusing and indicates that no fee is charged.

Recommendation

Rename the function to make it more clear for developers and users. E.g. `_calculateAmountWithFeeSubtracted`.

Fix 1.1

Fixed.

Go back to Findings Summary

I7: Unnecessary variables creation

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol	Type:	Gas optimization

Description

The contract LiquidityMiningInternal contains functions `delegatePwIpor`, `delegatePwIporAndStakeIpToken` and `undelegatePwIpor`. Inside these functions is a for loop over ipTokens array. In each iteration of the loop, new local variables `rewardsIteraion` and `accruedCompMultiplierCumulativePrevBlock` are created. Creating new variables inside the loops costs additional gas and should be avoided.

Recommendation

Move the variable creation before the loop cycle and leave only the assignment inside the loop.

Fix 1.1

Fixed.

Go back to Findings Summary

I8: Incorrect initialization pattern

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol	Type:	upgradeability

Description

The upgradeable contracts use the `_init` and `_init_unchained` functions as known from OpenZeppelin upgradeability Those functions are meant to initialize the contract state and avoid double initialization. The `_init` function should perform the logic that would typically be done in the constructor header, and the `_init_unchained` should perform the logic that would be done in the constructor body.

Vulnerability scenario

If the parents `init` functions are called, and two or more parent contracts have a same parent (diamond problem), it can lead to double initialization, because the `init` function of the shared parent would be called multiple times. Even though this is not a problem because there is no diamond pattern in the inheritance tree, we still consider it necessary to point this out to avoid future problems.

Recommendation

The `_init_unchained` function should not perform chaining. The `_init` function should contain linearized calls to the `_init_unchained` functions of all the contracts it derives from. Such an approach assures that double initialization will not happen and that all variables will get initialized.

For an inspiration contracts from OpenZeppelin can be used.

Fix 1.1

Fixed.

Go back to Findings Summary

I9: Usage of memory instead of calldata

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol, PowerIpor.sol	Type:	Gas optimization

Description

Several functions receive arguments as `memory`. However, they are only used as `calldata` and can be declared as such. Variable with `memory` type is stored temporarily in memory and can be modified, while `calldata` is stored in read-only memory and cannot be modified. `calldata` is saved in the cheapest memory location.

The following functions are affected:

LiquidityMiningInternal

- delegatePwIpor
- undelegatePwIpor
- delegatePwIporAndStakeIpToken

PowerIpor

- delegateToLiquidityMining
- delegateAndStakeToLiquidityMining
- undelegateFromLiquidityMining

Recommendation

Use `calldata` instead of `memory` for function arguments for read-only purposes because `calldata` is cheaper to use.

Fix 1.1

Fixed.

Go back to Findings Summary

I10: Reading length of an array in for loop

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMining.sol, LiquidityMiningInternal.sol, PowerIpor.sol	Type:	Gas optimization

Description

Certain functions in the project loop over an input array. In the loop header, there is the classical comparison `i < array.length`. This approach is more gas expensive than storing the length in a dedicated local variable and then comparing it against this variable: `i < variableStoringLen`.

The following functions are affected:

LiquidityMining - balanceOfDelegatedPwIpor

LiquidityMiningInternal - delegatePwIpor - delegatePwIporAndStakeIpToken - undelegatePwIpor

PowerIpor - delegateToLiquidityMining - delegateAndStakeToLiquidityMining - undelegateFromLiquidityMining

Recommendation

Create one dedicated variable and assign the array length to it. Then use this variable in the loop header. Array length will be read only once in a function, and some gas will be saved.

Fix 1.1

Partly fixed with a client's comment:

"In one place - LiquidityMiningInternal.delegatePwIporAndStakeIpToken - we

stay with array length inside the for statement because of calldata which cannot be used together with local variable - because of error "Stack too deep". Present changes makes, that gas cost now is lower than before changes." Go back to Findings Summary

I11: Redundant use of SafeERC20 library

Impact:	Info	Likelihood:	N/A
Target:	PowerIpor.sol, PowerIporInternal.sol	Type:	Coding practice

Description

Some contracts in the codebase use the SafeERC20 library. However, the contract interacts only with the project's Ipor token Therefore, the SafeERC20 library is redundant and can be removed because the Ipor token is a trusted contract.

The library is mainly meant for safer interaction with external tokens. Such a library is helpful because many tokens deviate from the standard in multiple ways (see list of non-standard tokens). The purpose of the SafeERC20 library is further discussed at the OpenZeppelin blog.

Recommendation

Short term, consider removing the library and measure the gas usage after the removal. If the gas usage is significantly different, consider removing the library. Long term, be aware that some tokens deviate from the standard and may not be fully compatible with the standard. In such cases, the SafeERC20 library should be used to interact with such tokens.

Fix 1.1

Fixed.

Go back to Findings Summary

I12: Lack of robust contract composition

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol	Type:	Data validation

Description

The project lacks a robust mechanism that could be used for secure contract composition. For validation of the contract, only a zero-address check is performed. The issue can be seen in LiquidityMiningInternal in the `initialize` function. There is no protection for initializing a random wrong address.

Recommendation

The identifier is a robust technique for avoiding mistakes during project deployment. Define an original identifier for a contract, such as keccak("contractName"), and then check the value during the contract composition, making it almost impossible for a wrong address to be accepted.

Fix 1.1

Client's comment:

"For IpTokens in LiquidityMining we will double check after deployment on Mainnet and before start mining rewards if there are correct IP Token addresses."

Go back to Findings Summary

I13: Require should be assert

Impact:	Info	Likelihood:	N/A
Target:	MiningCalculation.sol	Type:	Code quality

Description

Function `calculateAccruedRewards` In MiningCalculation contract contains require statement that always should be true:

```
require(
    blockNumber >= lastRebalanceBlockNumber,
    MiningErrors.BLOCK_NUMBER_LOWER_THAN_PREVIOUS_BLOCK_NUMBER
);
```

For function `calculateAccountRewards`, it works the same for the following requirement statement:

```
require(
    accruedCompMultiplierCumulativePrevBlock >=
accountCompMultiplierCumulativePrevBlock,
    MiningErrors.ACCOUNT_COMPOSITE_MULTIPLIER_GT_COMPOSITE_MULTIPLIER
);
```

Recommendation

The asserts provide much more information for reviewers and auditors because they convey that the given condition should always be true. Using requires is confusing because it implies that the condition could, in some cases, revert.

Fix 1.1

Client's comment:

”We would like to stay with ”required” instead ”assert” because is more clear and fast and easy do debug when error appeared (documented IPOR error code will be visible in Etherscan or in frontend console).”

Go back to Findings Summary

I14: The owner can prevent unstaking from LiquidityMining

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMiningInternal.sol	Type:	Trust model

Description

The owner of the contract LiquidityMiningInternal can prevent users from unstaking. The first way to do so is to pause the contract. The second way the owner can affect the unstaking is to remove a token by calling the function `removeIpToken`. In `unstake` function, there is a require that the token exists:

```
require(_ipTokens[ipToken], MiningErrors.IP_TOKEN_NOT_SUPPORTED);
```

The require will always return an error if the owner removes the token. Thus the user cannot unstake.

Recommendation

This issue can be resolved in multiple ways. Users' staked amount can be automatically returned to the user when the token is removed. The owner can be prevented from removing tokens when there are users staked. Alternatively, add some logic that allows users to unstake removed tokens but does not allow for staking them.

Fix 1.1

Fixed. User can unstake even if `IpToken` is no longer supported.

Go back to Findings Summary

I15: Code duplication

Impact:	Info	Likelihood:	N/A
Target:	LiquidityMining.sol, LiquidityMiningInternal.sol	Type:	Code quality

Description

In LiquidityMining contract in function `claim`, the rewards to be claimed are calculated. To calculate them, the following formula is used:

```
uint256 accruedCompMultiplierCumulativePrevBlock = MiningCalculation
            .calculateAccruedCompMultiplierCumulativePrevBlock(
                block.number,
                globalIndicators.blockNumber,
                globalIndicators.compositeMultiplierInTheBlock,
                globalIndicators.compositeMultiplierCumulativePrevBlock
            );

        uint256 iporTokenAmount =
MiningCalculation.calculateAccountRewards(
            accountIndicators.ipTokenBalance,
            accountIndicators.powerUp,
            accountIndicators.compositeMultiplierCumulativePrevBlock,
            accruedCompMultiplierCumulativePrevBlock
        );
```

However, this exact formula is also in LiquidityMiningInternal in `_calculateAccountRewards`.

Recommendation

A call to the internal function `_calculateAccountRewards` should be preferred to avoid code duplication and increase code readability.

Fix 1.1

Fixed.

Go back to Findings Summary

I16: Comment quality

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Code quality

Description

Across the project code base, there are some comments with typos or bad grammar, which can make the code harder to understand.

`LiquidityMiningTypes, #44`

```
/// @notive PowerUp is a result of logarythmic equastion defined in
documentation.
```

- grammar - logarithmic

`PowerIpor, #10`

```
/// Power Ipor smart contract allow you to stake, unstake Ipor Token,
deletage, undelegate to LiquidityMining Power Ipor Token.
```

- grammar - allows

`PowerIpor, #258`

```
///@dev We can transfer pwIporAmount because is in relation 1:1 to Ipor
Token
```

- grammar - it is

`MiningCalculation, #143`

```
/// @dev Composit Multiplier Cumulative for Prev Block stored in Account
structure cannot be greater than the newest accrued global
```

```
/// Composite Multiplier Cumulative for Prev Block
```

- a completly redundant comment without no added information about the code

`MiningCalculation, #28`

```
/// @dev Account's staked IP Tokens have to be >= 1
```

- redundant

`MiningCalculation, #47`

```
/// @notice Calculates aggreagated power up based on predefined in
specification equation.
```

- not possible to understand without some context

Recommendation

Correct typos, grammatical errors and improve the explanatory value of some comments.

Fix 1.1

Fixed.

Go back to Findings Summary

6. Report revision 1.1

No significant changes were performed in the contracts, and no new vulnerabilities were found. All the changes are responding to reported issues.

7. Report revision 1.2

Two contracts were renamed:

- `John.sol` --> `LiquidityMining.sol`
- `JohnInternal.sol` --> `LiquidityMiningInternal.sol`

All instances of string `John` were changed to `LiquidityMining` in the code and the report.

The protocol contains small architecture changes, and the new features that fix the latest issue. No main logic was changed.

The Solidity version of the protocol's contracts was upgraded from `0.8.16` to `0.8.17`.

H1: Inability to unstake when the contract runs out of rewards

High severity issue

Impact:	High	Likelihood:	Medium
Target:	LiquidityMining	Type:	Contract logic

Description

If the contract LiquidityMining run out of rewards, users will be unable to withdraw staked tokens. The transaction will fail and revert when the contract tries to send the tokens to the PowerIpor contract.

```
if (rewards > 0) {
  _transferRewardsToPowerIpor(msgSender, rewards); // REVERT
}
```

This revert leads to the lock of the user token at a moment when there is not enough balance for distributing the rewards.

Fix 1.2

A new function to unstake the tokens without the rewards was added. The remaining rewards balance is saved in `_allocatedPwTokens` mapping, from which it is possible to claim the rewards later.

```
if (rewards > 0) {
  if (claimRewards) {
    _transferRewardsToPowerIpor(msgSender, rewards);
  } else {
    _allocatedPwTokens[msgSender] += rewards;
  }
}
```

After this change, users can unstake their tokens without the rewards at any time. Moreover, at the same time, they retain the accumulated rewards that can be claimed after the rewards are resupplied to the contract.

Go back to Findings Summary

8. Report revision 1.3

The Ipor liquidity mining protocol was changed from the syntactical standpoint relative to the previous report revisions. This section lists the major changes that were made. The changes were checked using a diff between the files from the commit `c4eeca4` and the commit `64e303a`.

LiquidityMining

- update of the naming conventions,
- update of references to the renamed symbols,
- update of the in-line documentation,
- update of events

LiquidityMiningInternal

- update of the naming conventions,
- update of references to the renamed symbols,
- update of the in-line documentation,
- update of events

PowerIpor → PowerToken, PowerIporInternal → PowerTokenInternal

- update of the name of PowerIpor changed to PowerToken,
- update of the name of PowerIporInternal to PowerTokenInternal,
- update of the naming conventions,
- update of references to the renamed symbols,
- update of the in-line documentation,
- update of events

IporOwnableUpgradeable → MiningOwnableUpgradeable

- update of the name of IporOwnableUpgradeable to MiningOwnableUpgradeable,
- update of references to the renamed symbols

MiningErrors → Errors

- update of the name of MiningErrors to Errors,
- update of error strings from IPOR_ to PT_,
- update of the in-line documentation

MiningCalculation

- update of the naming conventions,
- update of references to the renamed symbols,
- update of the in-line documentation

IporMath → Math

- update of the name of IporMath to Math,
- update of the in-line documentation

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, IPOR: PowerIpor, LiquidityMining, January 28, 2023.

Appendix B: Differential fuzz tests for ABDK library

The protocol uses the mathematical library for mathematical operations over quadruple precision floating point data type. The library has been created by the company ABDK Consulting, and its code is publicly available. For testing this library, we decided to implement customized differential fuzz tests to inspect the behavior of the mathematical functions. All the library functions' results have been compared with the Python library: bigfloat. This specific python package is a wrapper for the well-known C library for multiple-precision floating-point computation MPFR.

The test is implemented in brownie testing framework. For running brownie it is necessary to create `brownie-cofnig.yaml` file. To execute test, run command: `$ brownie test`

Fail test output:

```
- LOG -
  input:
    num --> 1.00000001566297685667080454276213494
    hex --> 0x3fff0000004345a00bc3712888bb1d66
  res:
    num --> 2.25968988597102224284446534371027237e-08
    hex --> 0x3fe584363652324b703238215b2cff47

Results:
ABDK =
1.10000100001101100011011001010010001100100100101101110000001100100011100000100001010110110010110011111111101000111p-26
BIG  =
1.10000100001101100011011001010010001100100100101101110000001100100011100000100001010110110010110100000011100011 1p-26
eq: False
```

Test source code:

```
from brownie import accounts as a
from bitstring import BitArray
from random import randint
from brownie import ABDK
from bigfloat import *
import brownie
import pytest
import sys


@pytest.fixture
def abdk_lib():
    return a[0].deploy(ABDK)

def test_randomize(abdk_lib):
    # NUMBER OF ITERATION
    iter = 10000
    # ROUNDING CONTEXT
    c = Context(rounding=ROUND_TOWARD_ZERO)
    with c:
        for i in range(iter):
            if i % 10 == 0:
            # RANDOM NUMBERS WITH RANDOM RANGES
            ranges = [2**2, 2**8, 2**16, 2**32]
            random_range = randint(0, len(ranges)-1)
            num_a = randint(0, ranges[random_range])
            num_b = randint(0, ranges[random_range])
            hex_a = abdk_lib.fromInt(num_a)
            hex_b = abdk_lib.fromInt(num_b)
            # NUMBER OF OPEARTIONS
            num_operations = randint(1,5)
            for _ in range(num_operations):
                # RANDOM OPERATION
                operation = randint(0,3)
                # ABDK      OPERATION
                # BIGFLOAT  OPERATION
                if operation == 0:
                    hex_res = abdk_lib.add(hex_a, hex_b)
                    num_res = BigFloat(num_a) + BigFloat(num_b)
```

```
                elif operation == 1:
                    hex_res = abdk_lib.mul(hex_a, hex_b)
                    num_res = BigFloat(num_a) * BigFloat(num_b)
                elif operation == 2:
                    hex_res = abdk_lib.div(hex_a, hex_b)
                    num_res = BigFloat(num_a) / BigFloat(num_b)
                else:
                    hex_res = abdk_lib.log_2(hex_a)
                    num_res = BigFloat(log2(BigFloat(num_a)))
                # RANDOMIZE ASSIGMENT
                if (operation * i) % 3 == 0:
                    hex_a = hex_res
                    num_a = num_res
                else:
                    hex_b = hex_res
                    num_b = num_res

            abdk_float  = to_bigfloat(hex_res)
            big_float   = binary_bigfloat(num_res)

            # CATCH INCONSISTENCIES IN OUTPUT FORMAT FOR +- 0 AND +- INF
            if abdk_float == '1.1p+16384':
                abdk_float = 'nan'
            elif abdk_float == '1.p-16383':
                abdk_float = '0p+0'
            elif abdk_float == '-1.p-16383':
                abdk_float = '-0p+0'
            elif abdk_float == '1.p+16384':
                abdk_float = 'inf'
            elif abdk_float == '-1.p+16384':
                abdk_float = '-inf'
            elif abdk_float == '-1.p+0':
                abdk_float = '-1p+0'
            elif len(abdk_float) == 5:
                abdk_float = abdk_float.replace(".", "")

            #print(f'Results:\nABDK = {abdk_float}\nBIGG = {big_float}\neq:
{abdk_float == big_float}\n.')
            assert abdk_float == big_float

#Sign bit: 1 bit
#Exponent width: 15 bits
```

```
#Significand precision: 113 bits (112 explicitly stored)
def to_bigfloat(abdk):
    res = "0x"
    sign = 0
    exp = -16383
    sig = "1."
    b = BitArray(hex=str(abdk)[2:]).bin
    sign = "-" if b[0] == "1" else ""
    for i in range(1, 16):
        exp += int(b[i])*2**(15-i)
    exp_sign = "+" if exp >= 0 else ""
    sig += b[16:].rstrip("0")
    return f"{sign}{sig}p{exp_sign}{exp}"

def binary_bigfloat(b):
    return "{0:b}".format(b)
```

Appendix C: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

ackee

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq