Playground Labs

Kapital DAO

by Ackee Blockchain

21.12.2022

Contents

1. Document Revisions

0.1	Draft report	Dec 6, 2022
1.0	Final report	Dec 21, 2022
1.1	Fix review	Dec 21, 2022

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, RockawayX.

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
Jan Kalivoda	Lead 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

The Kapital DAO builds SaaS tools used by the world's largest guilds and games to onboard players and improve asset management, all powered by the KAP token.

Revision 1.0

Playground Labs engaged Ackee Blockchain to conduct a security review of Kapital DAO with a total time donation of 10 engineering days. The review took place between September 14, 2022, and December 2, 2022.

The scope was full-repository and the security review was focused on the GovernanceV2 deployment/upgrade process and the reintroduction of staked UniswapV2 KAP/ETH liquidity provider token voting.

The commit for the given scope was: `a8fe3c9`.

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

- ensuring the interactions with the oracle are correct,
- checking voting weight calculation,
- analysis of locking mechanisms,
- analysis of the upgrade process,
- simulation of the upgrade process,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,

- looking for common issues such as data validation.

Our review resulted in 9 findings, ranging from Info to Medium severity. The more severe issues are connected to Trust model.

Ackee Blockchain recommends Playground Labs to:

- address all reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review has been performed on the same commit (from first revision) as there were not any changes. Only statements from the client were added to the findings.

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: The `VESTING_CREATOR` role can vote multiple times	Medium	1.0	Acknowledged
M2: Governance can lock funds forever	Medium	1.0	Acknowledged
M3: Dynamic changes of the lock period	Medium	1.0	Acknowledged
L1: Lack of project identifier for address validation	Low	1.0	Acknowledged
W1: Pitfalls of upgradeability	Warning	1.0	Acknowledged

	Severity	Reported	Status
W2: Execute could not be triggered if there are burned a lot of KAP tokens	Warning	1.0	Acknowledged
I1: Boost can only be turned off	Info	1.0	Acknowledged
I2: Missing code comments	Info	1.0	Not fixed
I3: Ambiguous error messages	Info	1.0	Not 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.

Governance

The Governance contract is for proposal creation, voting, and execution. Users can vote only once per active proposal during the designated voting window. When voting, the member's voting weights from trusted sources are directly added to their chosen position. Anyone can execute a proposal in a given execution window if it passes its quorum.

GovernanceV2

The new Governance contract. The main difference is the reintroduction of staked UniswapV2 KAP/ETH liquidity provider token voting support and changes were implemented to allow for the toggling of voting weight sources and confirmation of governance upgrade by a permissioned caller.

GovernanceFund

The GovernanceFund contract is a storage mechanism for funds owned by the DAO. Arbitrary actions can be performed with these funds via a DAO vote and subsequent DAO execution.

GovernanceRegistry

The GovernanceRegistry contract holds the latest address of the Kapital DAO governance contract. Also, it allows changing this address by the actual Governance.

MultisigFund

The MultisigFund contract is a storage mechanism for funds owned by the corresponding multisig. Arbitrary actions can be performed with these funds by the multisig address.

Staking

The Staking contract allows users to stake Uniswap v2 KAP-ETH LP tokens in return for KAP rewards. The rewards are distributed according to the current emission rules. Users stake their assets for a lock period during which they can not withdraw them. As time progresses, staking users accumulate rewards, which they can claim. The rewards are then made available to the users after 52 weeks from the RewardsLocker. Users also have the option to extend their staking period, for which they are rewarded with a multiplier on their claim amount.

Vesting

The Vesting contract is used by the Kapital DAO team members and private investors to lock and linearly vest KAP. The contract allows members to collect the available funds and delegate the weight of tokens to another member.

KAPVesting

The new Vesting contract changes allows to set GovernanceRegistry (it is not immutable anymore).

RewardsLocker

The RewardsLocker contract acts as a storage for KAP rewards. Rewards can be claimed from the staking pool and then locked in the RewardsLocker for 52 weeks before being made available for withdrawal.

Transactor

The Transactor contract provides functionality to perform an arbitrary `call` on provided addresses. It is extended by Governance, MultisigFund and GovernanceFund to perform arbitrary actions with their funds.

Token

The Token contract represents the KAP token. It is `ERC20Burnable` with 1 billion pre-mint to the contract deployer.

TimeLock

The TimeLock contract allows users to lock their KAP tokens. The locked tokens then provide the users the ability to vote on proposals because they are exchanged for voting weight. The TimeLock has also a delegate version, which allows users to delegate their voting weight to another user.

Delegator

The Delegator contract serves as a base contract for time locks that want to use the weight delegation functionality.

Actors

This part describes the system's actors, roles, and permissions.

KAP Token Owner

The initial owner of all KAP tokens.

Governance

Governance represents the DAO. It can change the governance addresses in GovernanceRegistry. It also can manipulate the funds from GovernanceFund. In the Staking contract, the Governance can add and adjust the emissions and turn off the boosting.

Multisig

Multisig holders can retrieve funds from Multisig Fund. Multisig is also assigned the role KAP Saver in the RewardsLocker and the role Vesting Creator in the Vesting. Additionally, the multisig can adjust emissions and turn off the boost in the Staking contract.

Vesting Creator

A role created in the constructor of Vesting. It is granted to the creator of the Vesting contract and the Multisig. It allows the holder to create new vesting agreements.

KAP Saver

A role created in the constructor of RewardsLocker. The KAP Saver address can transfer (anytime after deployment) arbitrary amounts of KAP tokens from it.

MultisigVetoer

In the Governance contract, the team multisig is initially assigned the role MultisigVetoer. It allows the multisig to veto any proposals.

Delegator

A delegator can be any user who either decides to delegate their voting weight inside a TimeLock contract or inside Vesting. The delegator can revoke delegation or switch to a new delegate. As such it is an important role because it is directly involved in the voting processes in the DAO.

5.2. Trust model

Users have to trust the KAP Saver that he/she will not drain their rewards. Additionally, users have to trust the DAO members with a high voting weight so that they will not inconveniently manipulate the DAO or the parameters of the staking contracts. Also, the users have to trust the team multisig that it will pass the control to users and will not sabotage community-driven proposals through a veto. The KAP Token Owner has to be trusted to distribute the KAP tokens according to initial distribution rules. Lastly, the users have to trust the Vesting Creator that he/she will not secretly create new vesting agreements, which would allow him to manipulate the voting process.

M1: The VESTING_CREATOR role can vote multiple times

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	GovernanceV2.sol	Type:	Trust Model

Description

Vesting Creator can create vesting for a short period with a beneficiary address that he/she owns. Then, Vesting Creator can vote for the proposal from the beneficiary address and wait until the vesting ends to claim it and transfer it back to his address with the role. After that, Vesting Creator can create another vesting for other beneficiary addresses and vote again. This way, Vesting Creator can vote multiple times for the same proposal.

Exploit Scenario

Bob is the Vesting Creator.

- Bob creates vesting for his address A with a short period (shorter than the voting period).
- Bob votes for the proposal from his address A.
- Bob waits until the vesting ends.
- Bob claims the vesting and transfers it back to his initial address.
- Bob creates another vesting for his address B and repeats the steps above.

Recommendation

Do not allow to claim vesting during the voting period.

Solution (Revision 1.1)

Acknowledged by the client.

> The Vesting Creator role is assigned to a Gnosis multisig with shared control by five wallets. In the unlikely event that undesired behavior occurs, Vesting as a Voting Weight Source can be turned off in Governance by the Voting Manager. Any malicious proposals created can then be vetoed by the Vetoer.
>
> — Playground Labs

Go back to Findings Summary

M2: Governance can lock funds forever

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	TimeLock.sol	Type:	DoS

Listing 1. Excerpt from TimeLock.updateLockPeriod

```
    function updateLockPeriod(uint256 newLockPeriod) external {
        require(newLockPeriod > 0, "TimeLock: invalid lock period");
        require(
            msg.sender == governanceRegistry.governance(),
             "TimeLock: Only governance"
          );
          lockPeriod = newLockPeriod;
      }
  }
```

Description

Governance can decide to update the lock period to an arbitrary value. If a user is slow to react and doesn't withdraw his funds in time, the funds can end up being locked to an arbitrary amount of time.

The `TimeLock` contract contains the function `updateLockPeriod` (see Listing 1) which allows the governance to update the lock period. There is no restriction on the value that can be set.

Exploit scenario

Alice locks her funds in the TimeLock contract. The governance proposes a new proposal to update the lock period to some high value. The current lock period is longer than the voting period and Alice is thus unable to withdraw the funds in time.

Such a scenario is rather unlikely to happen. However, some powerful governors may get hacked, and the attacker can decide to update the lock period to a high value.

Recommendation

Decide upon a reasonable maximum lock period and restrict the `updateLockPeriod` function to only allow values below that maximum.

Solution (Revision 1.1)

Acknowledged by the client.

> The Timelock lock period should be at least equal to the Voting Period. This is the minimum to prevent double spending. To prevent malicious proposals from being created and executed within the lock period, the Timelock lock period should not be set above the WaitToExecute period in Governance. Malicious proposals can also be vetoed by the Vetoer.
>
> — Playground Labs

Go back to Findings Summary

M3: Dynamic changes of the lock period

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	TimeLock.sol	Type:	DoS

Listing 2. Excerpt from [TimeLock.updateLockPeriod](#)

```
    function updateLockPeriod(uint256 newLockPeriod) external {
        require(newLockPeriod > 0, "TimeLock: invalid lock period");
        require(
            msg.sender == governanceRegistry.governance(),
             "TimeLock: Only governance"
         );
         lockPeriod = newLockPeriod;
     }
 }
```

Listing 3. Excerpt from [TimeLock.unlock](#)

```
        require(
            block.timestamp > lockAgreement.start + lockPeriod,
            "TimeLock: early collect"
        );
```

Description

The `lockPeriod` parameter, which dictates the minimal amount of time that must pass before a user can withdraw their funds, can be changed dynamically (see [Listing 2](#)). When a user locks his funds a `LockAgreement` is created. The agreement is a struct that contains important parameters for the given lock.

However, the `LockAgreement` lacks a `lockPeriod` field. This means that the `lockPeriod` to which a user agreed can be changed dynamically to a new value

- that is because the `unlock` function uses the current value of the `lockPeriod`, see Listing 3. That can result in the user's funds being locked longer than he initially agreed to.

Exploit scenario

Alice locks her funds for 2 weeks. However, the `lockPeriod` is changed by the governance to 1 month. Now, Alice can't withdraw her funds for 1 month.

Recommendation

Implement logic that ensures that the `lockPeriod` remains constant for a given `LockAgreemnt`.

Solution (Revision 1.1)

Acknowledged by the client.

> If the Timelock lock period is set below the WaitToExecute period, users will have the opportunity to withdraw deposits before the change is implemented.
>
> — Playground Labs

Go back to Findings Summary

L1: Lack of project identifier for address validation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	`**/*`	Type:	DoS

Listing 4. Excerpt from [Governance.constructor](#)

```solidity
    constructor(
        address _vesting,
        address _teamMultisig
    ) {
        require(_vesting != address(0), "Governance: Zero address");
        require(_teamMultisig != address(0), "Governance: Zero
address");

        vesting = IVotingWeightSource(_vesting);
        _grantRole(VETOER, _teamMultisig);
    }
```

Listing 5. Project Identifier

```solidity
    bytes32 public constant CONTRACT_TYPE = keccak256("Playground
Governance");
```

Listing 6. Require statement for Data validation

```solidity
    require(
        Governance(governance).CONTRACT_TYPE() == keccak256("Playground
Governance"),
        "Not a Playground Governance"
    );
```

Listing 7. Excerpt from GovernanceRegistry.changeGovernance

```
        IGovernance _newGovernance = IGovernance(newGovernance);
        require(_newGovernance.votingPeriod() > 0, "Registry: Invalid
voting period");
```

Description

Currently, the contracts in constructors are only checked against the zero address (see Listing 4).

This approach can filter out the most basic mistakes, but it is not sufficient to ensure more deep address validation. Further validation can be done by using contract/project identifiers.

Such an identifier can be a constant string or a hash of a string (see Listing 5). Upon construction of a new contract that requires a governance address a check of the identifier would be done (see Listing 6). The same check can also be done in the upgrade function.

In the upgrade function `changeGovernance` in `GovernanceRegistry` a check similar to the identifier check is done (see Listing 7). The check requires the contract at the given address to implement the `votingPeriod` function. It is almost impossible that an incorrectly passed address would pass this check because the probability of a random contract implementing such a function is very low.

Exploit scenario

A contract deployer passes a wrong address to a constructor of one of the Playground contracts. The address is not the zero address, but it is not a valid address of a Playground contract either. As a result, a contract is deployed with the wrong parameters.

Recommendation

It is recommended to use more stringent input data validation using the project-wide identifier - not only in the upgrade function but also in the constructors.

Such an approach might be not possible to implement when the contracts are circularly dependent on each other. Yet, this approach should be implemented where possible.

Solution (Revision 1.1)

Acknowledged by the client.

> Due to the staggered nature of the contract deployments and the interdependency of the contracts themselves, address validation has been moved off-chain to simplify the deployment/upgrade processes. The contract deployments were thoroughly verified by the team following mainnet deployment.
>
> — Playground Labs

Go back to Findings Summary

W1: Pitfalls of upgradeability

Impact:	Warning	Likelihood:	N/A
Target:	Governance.sol, GovernanceRegistry.sol	Type:	Upgradeability

Description

The employed upgradeability system uses a GovernanceRegistry contract which holds the current active governance contract. When a new upgrade is needed, a new separate Governance contract is deployed, and only the address in the registry is changed. The only contract that can change the address in the registry is the current active governance contract.

Such an approach to upgradeability can have some downsides:

1. Upgrading to new governance can lead to double spending on the old governance

If the current governance contract uses a `TimeLock` contract as a source for voting weight and the new governance contract has a shorter voting period than the current one and an upgrade to the new version is employed, then a double-spending of tokens can happen on the old governance.

This issue would manifest itself if the following constraints and steps were followed:

1. The `TimeLock` contract queries the current governance using the registry.
2. The old governance uses the lock contract.
3. The `TimeLock` contract uses the length of the voting period of the new governance.
4. Because the new governance has a shorter voting period it can lead to double spending because the `TimeLock` contract will return non-zero voting

weight in a shorter time window.

- This will be an issue only if the old governance can still make some relevant calls to external addresses. Because the governance is mainly used to manipulate funds and the `GovernanceFund` queries the current governance registry, this most likely will not be an issue. It is mainly a warning if the behavior changes in future upgrades.
- More generally, this could be an issue for every outdated component that queries some external contract which again queries the `GovernanceRegistry`. As a result, the information retrieved using the current governance contract might not always be correct as it is only relevant to a different version of the governance contract.

2. Independence of the state of the governance contracts

The new governance contracts will not share the state as they are independent contracts. That could eventually lead to compatibility issues.

3. Interaction with the old governance contracts

Users can still interact with the old governance contracts even after the address in `GovernanceRegistry` was updated to a new version. This might become a problem in the future as the governance contracts will evolve and their logic will change. Mainly if some important interactions with external addresses could still be made.

4. Monitoring of old governance contracts

As the number of governance contracts will grow it might become harder and harder to monitor them all. But again, this might only be a problem if the old governance contracts could still make important external calls.

Solution ([Revision 1.1](#))

Acknowledged by the client.

> Deprecated Governance contracts can be disabled by turning off all Voting Weight Sources.
>
> — Playground Labs

Go back to Findings Summary

W2: Execute could not be triggered if there are burned a lot of KAP tokens

Impact:	Warning	Likelihood:	N/A
Target:	GovernanceV2.sol	Type:	Logic error

Listing 8. Excerpt from [GovernanceV2.execute](#)

```
        require(yaysTotal + naysTotal >= QUORUM, "Governance: Quorum");
```

Description

The QUORUM constant is set to 4% of the total supply of KAP tokens. Each proposal execution has to have bigger attendance (yays + nays) than the QUORUM constant. The problem is that the QUORUM constant is not updated when the KAP token is burned. So if the KAP token is burned a lot, the QUORUM constant will be too big and it could cause the execute function could not be triggered in a basic scenario.

Recommendation

Consider making the QUORUM constant dynamic.

Solution (Revision 1.1)

Acknowledged by the client.

> The team is carefully monitoring the smart contracts on-chain for unusual behavior. If the total supply of KAP is anticipated to change dramatically, the quorum parameter can be updated via a Governance upgrade.
>
> — Playground Labs

Go back to Findings Summary

I1: Boost can only be turned off

Impact:	Info	Likelihood:	N/A
Target:	Staking.sol	Type:	Contract logic

Description

The `Staking` contract allows users to accumulate additional rewards through a boost. Whether the boost is applied during claiming rewards is decided by the `boostOn` variable. This variable is initialized to `true` upon the construction of the contract and can be set to `false` by admin addresses. However, after turning it off, it is not possible to set it back on.

Recommendation

Ensure that such behavior is aligned with the expected behavior of the staking contract.

Solution (Revision 1.1)

Acknowledged by the client.

> Boosting is a core feature of Staking and will only be disabled in the event of undesired behavior or in the interest of the DAO.
>
> — Playground Labs

Go back to Findings Summary

I2: Missing code comments

Impact:	Info	Likelihood:	N/A
Target:	Delegator.sol, Timelock.sol	Type:	Contract logic

Description

The contracts `Delegator.sol` and `Timelock.sol` are missing code comments. Code comments are important to easily understand the code, thus making it easier to audit and maintain.

Recommendation

Add code comments (including NatSpec) to the contracts.

Go back to Findings Summary

I3: Ambiguous error messages

Impact:	Info	Likelihood:	N/A
Target:	**/*	Type:	Logging

Listing 9. Excerpt from Staking._boost

```
        require(block.timestamp < end, "Staking: Remaining");
```

Listing 10. Excerpt from Staking._boost

```
        require(
            MIN_LOCK <= newLock && newLock <= MAX_LOCK,
            "Staking: New lock"
        );
```

Description

Many places in the code base contain ambiguous error messages (eg. see Listing 9, Listing 9). Such errors do not contain enough information to easily parse why the given transaction failed.

Additionally, the errors contain the name of the contract they originated in. This information, however, can be parsed from the failed transaction.

Recommendation

Use precise error messages which would be more descriptive about the causes of the erroneous call. Such messages will allow for a more straightforward analysis of failed transactions.

Go back to Findings Summary

6. Report revision 1.1

Issues from previous revision were only acknowledged. No changes were made.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Playground Labs: Kapital DAO, 21.12.2022.

Appendix B: 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.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq