ackee blockchain

Karpatkey

zodiac-modifier-roles

by Ackee Blockchain

30.12.2022

Contents

1. Document Revisions

0.1	Draft report	December 21, 2022
1.0	Final report	December 21, 2022
1.1	Fix review	December 30, 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
Lukáš Böhm	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

Two reviewed contracts Roles and Premissions are part of the Zodiac protocol. The protocol is an expansion pack for DAOs, that allows more complex permission management with better flexibility and interaction. Zodiac architecture is divided into four main parts: Avatars, Modules, Modifiers, and Guards. The code review focuses on Modifiers' submodule `Roles`. Roles allow controlling the access right for specified enabled modules. Modules scope allowed addresses, allowed functions on allowed addresses, allowed parameters on allowed functions, type of the call, and whether or not they can send ETH to an address.

Revision 1.0

Karpatkey engaged Ackee Blockchain to perform a security review of the Roles and Permissions contracts with a total time donation of 10 engineering days between December 6 and December 21, 2022 and the lead auditor was Lukáš Böhm. The audit has been performed on the commit `7acf0ef`.

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:

- detecting possible reentrancies in the code,
- ensuring modules cannot call not allowed addresses,
- ensuring modules cannot call not allowed functions or pass not allowed parameters,
- looking for common issues such as data validation,
- adherence to solidity best practices,
- the correctness of access controls.

Our review resulted in 8 findings, ranging from Info to Medium severity. The overall quality of the code is good. The contract Roles is well in-code documented. On the other hand, the library Permissions lacks good in-code documentation.

Ackee Blockchain recommends Karpatkey:

- properly validate input data,
- carefully handle the owner role,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on the given commit: `f84b65c`.

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.

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: Dangerous transfer ownership	Medium	1.0	Acknowledged
M2: Renounce ownership risk	Medium	1.0	Acknowledged
M3: Lack of data validation in contract deployment	Medium	1.0	Acknowledged
M4: Lack of data validation in `setMultisend`	Medium	1.0	Acknowledged
W1: Usage of `solc` optimizer	Warning	1.0	Acknowledged
I1: Incorrect use of `assert` statement	Info	1.0	Acknowledged
I2: Unreachable condition	Info	1.0	Fixed

	Severity	Reported	Status
I3: Unnecessary usage of post-inc	Info	1.0	Acknowledged

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 for understandability purposes and does not replace project documentation.

Contracts

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

Permissions.sol

The library contains logic for the Roles assignment, removing, and updating.

Every `Role` structure has a:

- members (modules) - mapping of addresses that are allowed to perform actions for a given role
- targets - mapping of addresses that members can call
- functions - mapping of functions that members can call on targets
- compValues - mapping of values that limits the function arguments
- compValuesOneOf - mapping of lists of values that the function arguments must be one of

The library also provides functionality for checking if role members can call a specific function on a specific target with specific arguments.

Roles.sol

The contract inherits from gnosis `Modifier` and uses the logic of Permissions library. The first part of the contract implements the functions for roles

assigning and removing. The main logic is happening in the Permissions library. The contract owner is the only address with the privilege to update the parameters of the `Role` struct.

The second part of the contract contains four executing functions. These functions can be called by the appropriate role members or the module set with the default role. Two of these functions also return data resulting from the called function.

Actors

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

Owner

The Owner of Roles contract is set in `setUp` function. The Owner can assign the parameters for the roles, revoke and update them, and set `multisend` address. In an out-of-scoped contract `Modules` owner can also update addresses `avatar` and `target`. The Owner role can be transfered to a new address.

Modules

The addresses that are allowed to perform actions for a given role. The modules can be assigned to the role by the Owner of the contract.

5.2. Trust model

The protocol and users rely on the Owner's honesty. The role of the Owner has the ultimate power over the protocol behavior.

M1: Dangerous transfer ownership

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Roles.sol	Type:	Access Control

Description

The contract Roles inherits functionality to transfer the ownership from OZ contract `OwnableUpgradeable`.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}

function _setOwner(address newOwner) private {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

However, the transfer function has **not** have a robust verification mechanism for the new proposed owner. The error cannot be recovered if a wrong owner address is accidentally passed to it. Thus passing a wrong address can lead to irrecoverable mistakes.

Exploit scenario

The current owner Alice wants to transfer the ownership to Bob. Alice calls the `transferOwnership` function but supplies the wrong address by mistake. As a result, the ownership will be passed to a wrong and possibly dead address.

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

The current owner Alice wants to transfer the ownership to Bob. The two-step process would have the following steps: Alice proposes a new owner, namely Bob. This proposal is saved to a variable candidate. The candidate Bob calls the function for accepting the ownership, which verifies that the caller is the new proposed candidate. If the verification passes, the function sets the caller as the new owner. If Alice proposes the wrong candidate, she can change it. However, it can happen, with a very low probability that the wrong candidate is malicious (most often, it would be a dead address).

An authentication mechanism can also be employed to prevent the malicious candidate from accepting the ownership.

Fix 1.1

Acknowledged. Client's comment:

" Before the introduction of OZ twoStepOwnership contract we did not want to implement a non-standard version of the ownable contracts. We are considering implementing this across the Zodiac stack but as for the time being we consider it reasonable to keep the current version of ownable. This is largely due to the fact that the owner in Zodiac contracts will most likely be a DAO that will carefully propose and select any modifications to the owner state, reducing the likelihood of accidental updates to the wrong address "

Go back to Findings Summary

M2: Renounce ownership risk

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Roles.sol	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 or in a protocol where the owner's role is necessary can have harsh consequences. Therefore it has to be used only after careful consideration.

For example, when one of the modules became malicious, and the ownership is renounced, the malicious module can continue to operate under the assigned role, and no one can stop it.

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 if there is no other way to make the protocol more trustless.

Fix 1.1

Acknowledged. Client's comment:

" Zodiac is a library for DAO tooling and extensions to a multisig, the recommendation to use a multisig in the protocol is naturally the case for these contracts. "

Go back to Findings Summary

M3: Lack of data validation in contract deployment

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	Roles.sol	Type:	Data validation

Description

The contract's constructor and initialization `setUp` function set the addresses of `owner`, `avatar`, and `target`. However, there is almost no data validation implemented. The only check is performed inside the function `transferOwnership(_owner)`, which does not allow the zero address to be set as an owner.

If the function caller or the contract deployer passes the wrong address, the protocol will work in an unintended way. The parent contract `Modules.sol` contains two setter functions for `avatar` and `target` variables that help to fix such a mistake and set the correct address. Nevertheless, the wrong address may be passed as an input argument, stay undetected for some time, and make the protocol useless.

Recommendation

The two-step ownership transfer will solve the problem for the `owner` variable, as mentioned in a separate finding.

The same solution mentioned in multisend data validation finding can be applied to the `avatar` and `target` variables. If these two addresses are in the company's control, their implementation can be modified. Then the ID identifier should be used to check the addresses. Define an original identifier for a contract, such as keccak("contractName"), and then check the value

during the contract composition by calling `contractAddress.getId()` and comparing it with an input argument. It makes it almost impossible for a wrong address to be accepted.

If addresses are already deployed and cannot be changed, at the very least check against the zero address should be performed.

Fix 1.1

Acknowledged. Client's comment:

" We don't find checks for zero address useful given that the solidity compiler no longer allows for empty arguments to function calls that originally made that case more likely and the domain space for invalid addresses is too large to check onchain. The suggestion to add an identifier to `_target` and `_avatar` is noted, these will likely be a Safe contract but this is not necessarily the case. These addresses can always be tested for correctness and resubmitted as well. "

Go back to Findings Summary

M4: Lack of data validation in `setMultisend`

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	Roles.sol	Type:	Data validation

Description

The function `setMultisend` in the contract Roles lacks any data validation whatsoever. The input address should refer to a `multisend` library contract. However, there is no check for this.

```
/// @dev Set the address of the expected multisend library
/// @notice Only callable by Owner.
/// @param _multisend address of the multisend library contract
function setMultisend(address _multisend) external onlyOwner {
    multisend = _multisend;
    emit SetMultisendAddress(multisend);
}
```

Even though the Owner can call this function anytime, there is still a possible risk when the Owner does not notice the mistake. In such a scenario, the protocol will work in an unintended way until the new `multisend` address is set.

Vulnerable scenario

The user wants to execute a transaction where the address `to` == `multisend` library address, but the library address is not set correctly. For that reason following condition will end up in the `else` branch, and the transaction will probably fail.

```
if (multisend == to) {
    checkMultisendTransaction(role, data);
} else {
```

```
    checkTransaction(role, to, value, data, operation);
}
```

Recommendation

If the company controls a `multisend` library, then the ID identifier should be used to check the address. The identifier is a robust technique for avoiding mistakes during the address initialization and assignment. Define an original identifier for a contract, such as keccak("contractName"), and then check the value during the contract composition by calling `contractAddress.getId()` and comparing it with an input argument. It makes it almost impossible for a wrong address to be accepted.

If `multisend` library is not in the control of the same company, then the address should, at the very least, be checked against the zero address. Another option to make the protocol more transparent is to use a `getter` function, which allows users (modules) to check the library's address and avoid unintended behavior.

Fix 1.1

Acknowledged. Client's comment:

" Similar response to M3. MultiSend is a Safe library contract that is deployed once and reused and is not frequently updated. We do not feel it is necessary to add an ID. "

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

Acknowledged. Client's comment:

" We consider the optimizer stable at this point and its use will allow for new features to be added in a V2 of the roles modifier. We have reviewed the solc audit and find their responses reasonable for usage of the optimizer "All our semantics tests are run with optimizer enabled and optimizer disabled. Also all of them are run in a combination with multiple EVM versions." "

Go back to Findings Summary

I1: Incorrect use of `assert` statement

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

Description

The function `checkTransaction` of the contract Permissions violates the assert in line `#281`. The `assert()` is used in Solidity to assert invariants. The well-built code should never ends in a failing assert.

There are two possible ways to reach this point:

- incorrect `assert` usage,
- bug in a code that leads to a false invariant.

In this scenario, the `assert` is used to revert if none of the following conditions end up in a `true` value:

```
if (data.length != 0 && data.length < 4)

if (target.clearance == Clearance.None)

if (target.clearance == Clearance.Target)

if (target.clearance == Clearance.Function)
```

Before the conditions are checked, the new variable of the type `TargetAddress` is created. A value of the same type from a role mapping is assigned to the variable.

```
TargetAddress storage target = role.targets[targetAddress];
```

Even if the mapping is empty, the new variable is created, and it holds the default value of the type `TargetAddress` which is `Clearance.None`. For this reason, the `assert` function will never be triggered because one of the previous conditions will be reached and return the function.

Recommendation

Remove the `assert()` function from the code because it is a dead piece of code that can never be reached.

Fix 1.1

Acknowledged. Client's comment:

" While obviously unreachable, this assert is symbolic, and we will for now keep it. "

Go back to Findings Summary

I2: Unreachable condition

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

Description

In the contract Roles is the internal function `setupModules`. This function is called only in the `setUp` function, which is called in the constructor. The function `setUp` also works as a initialize function that has to be called while initializing a new contract instance. During the initialization, the function `__Ownable_init()` with `initialized` modifier is called and guarantees that the function `setUp` will be called only once.

Because the call is atomic, the function `setupModules` will always be called during the contract deployment when `modules[SENTINEL_MODULES]` will be zero. For this reason, the function's condition can never end up in a **True** state. Thus it can be called a dead code.

```
function setupModules() internal {
    if (modules[SENTINEL_MODULES] != address(0)) {
        revert SetUpModulesAlreadyCalled(); // can never be called
    }
    modules[SENTINEL_MODULES] = SENTINEL_MODULES;
}
```

Recommendation

Remove the dead code.

Fix 1.1

Fixed. The condition was removed, and the more appropriate `assert` function

was used instead.

```
assert(modules[SENTINEL_MODULES] == address(0));
```

Go back to Findings Summary

I3: Unnecessary usage of post-inc

Impact:	Info	Likelihood:	N/A
Target:	Roles.sol, Permissions.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

- Roles.sol #298,
- Permissions.sol #321, #366, #475, #491, #507, #612, #630, #811

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

Acknowledged. Client's comment:

" No action will be taken for this point. "

Go back to Findings Summary

6. Report revision 1.1

No significant changes were performed in the contracts. All the changes are responding to reported issues.

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain, Karpatkey: zodiac-modifier-roles, 30.12.2022.

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.

ackee

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq