Audit Report

Autonomy Osmosis

v1.1

March 6, 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 Autonomy Labs Ltd. to perform a security audit of the Autonomy Osmosis 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:

https://github.com/Autonomy-Network/autonomy-osmosis-contracts

Commit hash: `00f5679fe7a518c88d9a0e8b0e5c9d0b496b34d1`

Fixes have been verified on the commit with the following hash:
`02528b908b97c0a5548623972790df59122ddc5d`

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

Autonomy Network is a decentralized automation protocol. It allows users to have transactions automatically executed in the future under arbitrary conditions.
It is an infrastructure service and a tool that allows anyone to automate any on-chain action with any on-chain condition.

The audit scope comprehends the `registry-stake` and the `wrapper-osmosis` CosmWasm smart contracts.

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	**Low**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium**	-
Test coverage	**High**	`cargo tarpaulin` reports a code coverage of 92.50%

Summary of Findings

No	Description	Severity	Status
1	Delays in the execution of `UpdateExecutor` transactions could temporarily inhibit the capability of the protocol to execute requests	Major	Resolved
2	`stakes` vector could exceed the CosmWasm VM memory limit when loaded	Major	Resolved
3	`Unstake` transactions are likely to fail if more than one of them is processed in the same block	Major	Acknowledged
4	Protocol is prone to censorship	Minor	Acknowledged
5	Partially implemented tax deductions may lead to failures if tax rates are updated	Minor	Resolved
6	Contracts are not compliant with `CW2` Migration specification	Minor	Resolved
7	`AssetInfo` struct is not validated	Minor	Resolved
8	Incorrect sanity check could let the execution to panic	Minor	Resolved
9	Updating the `AUTO` token information cause staked funds to be locked in the contract	Minor	Resolved
10	Updating the stake_amount config parameter could cause an underflow during unstake	Minor	Resolved
11	Arbitrary request execution could lead to a privilege escalation in the scenario where the `registry` is the owner of another contract	Informational	Resolved
12	Additional funds sent to the contract are lost	Informational	Resolved
13	Contracts should implement a two step ownership transfer	Informational	Resolved
14	Custom access controls implementation	Informational	Resolved
15	`cosmwasm-storage` crate requires custom implementations	Informational	Resolved

Detailed Findings

1. Delays in the execution of `UpdateExecutor` transactions could temporarily inhibit the capability of the protocol to execute requests

Severity: Major

The guard in `contracts/registry-stake/src/contract.rs:338` checks if the current epoch's `executor` is set or not, ensuring that `update_executor` is called before `execute_request` for the new epoch.

Since transaction order from the mempool is not deterministic, the capability of the protocol to execute requests will be temporarily inhibited until an `UpdateExecutor` transaction is executed.

Recommendation

We recommend calling the `update_executor` function in line `contracts/registry-stake/src/contract.rs:335` in order to update the `executor` if it is still not set for the current `epoch`.

Status: Resolved

2. `stakes` vector could exceed the CosmWasm VM memory limit when loaded

Severity: Major

In `contracts/registry-stake/src/state.rs:68`, the `stakes` attribute is defined in the `State` struct as a vector of strings.

This implies that every time the `read_state` function is invoked, all the `State` data, including the `stakes` vector, is loaded into memory.

Since the `stakes` vector has no maximum length, the CosmWasm VM memory could not have enough space available to load it resulting in an execution panic.

A bad actor could intentionally use this issue to cause any interaction with the protocol to be very gas intensive, up to the point where any interaction will run out of gas.

With current parameters on Osmosis, the cost of this attack would be in the millions of US dollars. While this might still be economically viable, we classify this issue as major, since it can be resolved with a contract upgrade.

Recommendation

We recommend defining a hard cap for `stakes` vector size or implementing a different data structure that gives access to the required information without loading the entire vector data in memory.

Status: Resolved

3. `Unstake` transactions are likely to fail if more than one of them is processed in the same block

Severity: Major

The `Unstake` transaction handler, defined in `contracts/registry-stake/src/contract.rs:635-698`, takes as input parameter a vector of numeric indexes that represent the user's stake slots in the `stakes` vector.

In order to process the deletion of the vector data at the provided indexes within an `O(1)` asymptotic cost, the `swap_remove` function is used. This method performs a delete substituting data in the selected index slot with the data stored in the latest element of the vector.

Consequently, a removal of one element affects also the ordering of the elements stored in the latest positions of the vector. If we have more than one `Unstake` transaction in the same block, the indexes for any subsequent positions might be wrong, leading to an error, or, if the same sender controls the position, to the deletion of the wrong position.

This implies that the likelihood of an `Unstake` transaction to succeed depends on different unpredictable factors like:

- The order of transactions in a block
- The cardinality of `Unstake` transactions in a block
- The distance of the index to the end of the `stakes` vector

The likelihood of having a transaction error increases drastically on particular events, for example a market crash where users may want to exit funds from the protocol quickly.

Additionally, bad actors can grieve users by staking many small amounts, and then front-running others to cause unstake failures. This could be used together with a short of the tokens to attack the whole protocol and profit from a panic in users.

Recommendation

We recommend collecting all the indexes that have to be deleted from different `Unstake` transactions and then remove them when electing the next epoch `executor`.

Another solution could be using a tree, for example an `avl::AvlTreeMap` in order to store new positions.

Status: Acknowledged

4. Protocol is prone to censorship

Severity: Minor

The guard defined in `contracts/registry-stake/src/contract.rs:342-347` ensures that only the currently elected `executor` can submit the `ExecuteRequest` message.

This implies that if the currently elected `executor` is not responsive and actively participating, the protocol will be stuck for the entire `epoch` with no executed `requests`.

Since every `AUTO` token holder can stake its tokens and participate in the election, there is a possibility that the `executor` could be an inactive protocol participant.

Also, a malicious actor could stake a large amount of tokens in order to intentionally stop the `requests` execution, causing all or targeted executions to get delayed.

This is particularly impactful for users and attractive for attackers if `requests` contain trading transactions that require to be executed in a timely manner.

Usually, in proof of stake (PoS) systems, such actors would bear an economic cost such as infrastructure maintenance, cost of staked capital, and third-party delegation loss. Also, in this case, it is required to deposit the stake in the contract only for one block since there is no unbonding period, reducing the attacker's capital cost and making the attack cheaper with respect to other PoS systems.

In fact, in `contracts/registry-stake/src/contract.rs:635-698`, the `Unstake` message handler allows the currently elected executor to unstake all of its stake slots without losing its role.

Consequently the `executor` can be in charge of its role without having its stake deposited in the protocol. This would allow an attacker to get an `AUTO` loan, stake and be elected, unstake, repay the loan and act as an inactive `executor` in order to stop `requests` execution without having to hold or loan tokens for long.

Also, it is worth to note that in `contracts/registry-stake/src/contract.rs:645`, during the handling of the `Unstake` transaction, the `executor` election is done before removing the unstaking user's address from the `stakes` vector. This implies that an

unstaking user could be elected as the current epoch `executor` without having capital at risk stored in protocol.

We classify this issue as minor since the client intends to set the epoch length to 100 blocks, which corresponds to 10 minutes on Osmosis and is low enough to reduce the impact of an attack. Still, market manipulation even over a short time window may be exploited.

Recommendation

We recommend:

a) Increasing the cost for an executor by disallowing executors to unstake while they are elected.
b) Increasing the cost for an executor by adding an unbonding period.
c) Keeping the epoch duration short to reduce the amount of damage an inactive executor can cause.
d) Increasing the number of executors per epoch, such that there is no single point of failure. Multiple executors could be allowed to execute requests within alternating preassigned slots. This prevents inactive executors to block the protocol, since there are multiple ones in each epoch, so requests are still executed. While there are multiple approaches, executor selection could be done by randomly choosing from the set of stake-weighted executors, while removing already selected executors to prevent double-selection. In the next epoch, the same algorithm should run again, without filtering out previously selected executors. For an efficient implementation, an optimized data structure should be used, for example, a sortition sum tree.

Status: Acknowledged

The client states:

> tl;dr there is actually infact no additional cost to attack Autonomy compared to any other PoS system with a similar market cap. The recommendations don't actually solve the underlying issues, which are present in every other PoS system such as ETH. c) is valid in that the recommendation would marginally help (but not solve) the underlying issue, but the difference in practicality between other PoS systems and Autonomy in regard to c) are so small that it likely wouldn't actually make a difference to the decision to attack, and the 2nd order effects of implementing the recommendation could cause more harm than the underlying issue it's trying to prevent.
>
> It's true an executor has the ability to censor or ignore txs to the exact same degree as a validator/miner in any PoS/PoW etc system - this is a widely known and acknowledged issue with any consensus protocol/blockchain. As with other PoS/PoW etc systems, the attacker is losing out in fee revenue by censoring things and therefore pays an opportunity cost, and the requests will eventually be executed aslong as there is atleast 1 executor who is purely financially motivated / not malicious. There is no extra ability of an attacker to do this in Autonomy vs any other system like ETH's PoS etc.
>
> It's true that performing the censorship attack on Autonomy doesn't require running infrastructure (bot/node) like it would in the ETH example, since you would still have to produce blocks which requires running an ETH node, but this cost, especially in comparison to the

amount of capital required to perform a censorship attack on Autonomy, is so small that it's a negligible rounding error and can basically be ignored.
Presumably 'cost of staked capital' is referring to the loss in value of the staked AUTO tokens when performing the censorship attack - an attacker's optimal strategy is to borrow AUTO tokens and use them to stake, since they only need to repay the debt in AUTO and therefore don't suffer a loss from any price decrease in AUTO when censoring. This is equally possible in ETH - to borrow ETH with stablecoins as collateral, stake it, produce empty blocks in an attempt to drop the price of ETH, and not lose any money because the debt is denominated in ETH. An unbonding/unstaking period does not change the economics of this attack. It is therefore not cheaper to perform the censorship attack on Autonomy vs a PoS system like ETH if it had an equivalent market cap to Autonomy.

It's not a problem that executors can unstake while still being an executor because there is no slashing in the system (as it's impossible to prove malicious censorship, the same as it's not possible to slash for that in ETH or any other chain), and censoring is the only 'bad' thing that an executor can do.

Recommendation responses:
a) preventing unstaking while an executor/validator in any PoS system, including Autonomy, does not increase the cost of attack via devaluing of staked capital, if the attacker borrowed the tokens they staked, as shown above

b) adding an unbonding period to any PoS system, including Autonomy, does not increase the cost of attack via devaluing of staked capital, if the attacker borrowed the tokens they staked, as shown above

c) reducing the epoch length/time would not reduce an attacker's ability to censor in general because having 50% of the stake means they can only execute for 50% of the time, regardless of the epoch length. However it is true that, because of The Law of Large Numbers, a longer epoch means fewer epochs over a given (short in practicality) time period, which means there is a higher change that the same executor will be chosen for every epoch in that time period without interruption. However, e.g. on Osmosis the block time is ~6s, so 100 blocks is 600s i.e. 10 minutes. Assuming there is $100M of AUTO staked on Autonomy then in order for an executor to get 75% of the stake, they'd need to borrow $300M of AUTO and put up >> $300M as collateral. Then they have a 0.75^6 = 0.178% chance to be the executor for any given hour. The chance to censor for 1 full day is 0.75^(6*24) = 0.00000000000000001%. Therefore the chance of being able to fully censor for any amount of time that would cause the price of AUTO to dump is impractically low, especially given the capital requirements. It is true that even if an attacker isn't able to fully censor for an hour, simply causing delays in things like limit orders could impact the price of AUTO. There is an inherent tradeoff between the length of the epoch and the fees charged to users - the extreme of 1 epoch = 1 block would mean that executors would only be able to know whether they're executors for the next block, and would need to broadcast all their execution txs in the hope that they get added into the next block. The practical reality of blockchains is that this is rarely the case, and even using a 'fast' gas price results in the tx being mined >=2 block in the future. Therefore every executor would waste gas on most of their execution attempts that are reverted if they're mined after the next block when they're no longer the executor. This means that running an executor is now not guaranteed to be risk-free profitable, which it should be when not being malicious - the execution fee that goes to executors would therefore need to be increased in proportion to the risk of late-mined txs that comes with a smaller epoch length, increasing the cost and reducing the appeal of the system as a whole, leading to lower executor revenue overall. Even with the extreme of 1 epoch

> = 1 block, it's still possible to cause delays in executions of e.g. limit orders and potentially affect the price of AUTO. It's still an open question as to what the ideal epoch length is, but 100 blocks is enough to probabilistically guarantee that full censorship can't occur while minimizing the fees charged to users.
>
> d) Preventing the same executor from being selected >=2 times in the same epoch (i.e. selecting 100 executors proportional to stake in a 100 block epoch 'without replacement') by saving a list of the executors already selected (which are identified by their addresses) as a mechanism to prevent the same attacker from being the same executor for every block does not work and is no different from having an epoch length of 1 block. This is because an attacker can have arbitrarily many addresses that are all staking the same amount each. Say an executor has 50% of the total stake spread equally over 1000 staker slots, each with different addresses - if the attacker is chosen for the 1st slot in the epoch, then the chance of the attacker to be selected for the 2nd slot is 50% * 999/1000. As the attacker spreads out their stake over N staking slots, the chance for the attacker getting the 2nd staking slot is 50% * (N-1)/N, which is effectively 50% as N gets larger. Therefore while an attacker cant Sybil attack the overall PoS system, they can Sybil this specific mechanism of trying to reduce the ability of an attacker to cover X blocks/slots any more than having an epoch have a length of 1 block does. This is effectively the same recommendation as 3.c., but the extreme version, and is therefore also completely impractical for the reasons described above - very few executors would even be able to execute anything because on average most txs don't go in the next block, B, and you couldn't broadcast the execution until block B-1 had already been mined.

5. Partially implemented tax deductions may lead to failures if tax rates are updated

Severity: Minor

In `packages/autonomy/src/asset.rs:44-46`, the `compute_tax` function that is used to calculate taxes to be paid when transferring native coins, always returns zero.

Consequently it is not effective and future changes in the Osmosis (or another) chain could inhibit the execution of requests submitted in the `registry`.

Recommendation

We recommend implementing a tax rate query and deducting taxes from native assets to ensure full compatibility.

Status: Resolved

6. Contracts are not compliant with `CW2` Migration specification

Severity: Minor

The following contracts do not adhere to the `CW2` Migration specification standard:

a) `registry-stake`

`b) wrapper-osmosis`

This may lead to unexpected problems during contract migration and code version handling.

Recommendation

We recommend following the CW2 standard in all the contracts. For reference, see https://docs.cosmwasm.com/docs/1.0/smart-contracts/migration.

Status: Resolved

7. `AssetInfo` struct is not validated

Severity: Minor

In `contracts/registry-stake/src/contract.rs:61`, the `AssetInfo` struct representing the `AUTO` token provided during the instantiation is stored without validation.

Since this struct contains an address, it should be checked to ensure that it is a valid one.

Recommendation

We recommend validating the `AssetInfo` struct before storing it.

Status: Resolved

8. Incorrect sanity check could let the execution to panic

Severity: Minor

In line `contracts/registry-stake/src/contract.rs:655`, the `idx` value is validated to be greater than the `stakes` vector length.

However since the `idx` value is used previously in line `contracts/registry-stake/src/contract.rs:651` as an index for accessing the `stakes` vector, the mentioned validation will never be reached and the execution will panic.

Recommendation

We recommend moving the validation check before accessing the vector index in line `651`.

Status: Resolved

9. Updating the `AUTO` token information cause staked funds to be locked in the contract

Severity: Minor

In `contracts/registry-stake/src/contract.rs:152`, the `owner` is able to update the `AUTO` token `AssetInfo`.

Since both the `Stake` and `Unstake` transactions are dependent on this information, an update of it without migration would leave staked funds stuck in the contract.

We classify this issue as minor since only the owner can cause it.

Recommendation

We recommend removing the possibility of update the `AUTO` token `AssetInfo` and handle this scenario with a migration.

Status: Resolved

10. Updating the `stake_amount config` parameter could cause an underflow during unstake

Severity: Minor

In `contracts/registry-stake/src/contract.rs:153`, the `owner` is able to update the `stake_amount config` parameter.

If the new value is greater than the previous one, it causes an underflow panic during the handling of the `Unstake` messages in `contract/registry-stake/src/contract.rs:663`.

We classify this issue as minor since only the owner can cause it.

Recommendation

We recommend implementing validation of the new value or removing the possibility of updating the `stake_amount` value and handle this scenario with a migration.

Status: Resolved

11. Arbitrary request execution could lead to a privilege escalation in the scenario where the `registry` is the owner of another contract

Severity: Informational

In `contracts/registry-stake/src/contract.rs:175`, the user can set any arbitrary target address.

In a scenario where the `registry-stake` contract is the owner of another smart contract, this could allow an attacker to execute arbitrary privileged messages.

Recommendation

We recommend implementing a blacklist mechanism for target addresses.

Status: Resolved

12. Additional funds sent to the contract are lost

Severity: Informational

In `contracts/registry-stake/src/contract.rs:577` and `contracts/registry-stake/src/contract.rs:462`, a check is performed that ensures that coins with the expected `denom` field have been sent.

This validation does not ensure that no other native tokens are sent though, and any additional native tokens are not returned to the user, so they will be stuck in the contract.

Recommendation

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

Status: Resolved

13. Contracts should implement a two step ownership transfer

Severity: Informational

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.

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 lowercased.
2. The new owner account claims ownership, which applies the configuration changes.

Status: Resolved

14. Custom access controls implementation

Severity: Informational

The contracts within the scope of this audit implement custom access controls. Although no instances of broken controls or bypasses have been found, using a battle-tested implementation reduces potential risks and the complexity of the codebase.

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

Recommendation

We recommend using a well-known access control implementation such as `cw_controllers::Admin` (https://docs.rs/cw-controllers/0.14.0/cw_controllers/struct.Admin.html).

Status: Resolved

15. `cosmwasm-storage` crate requires custom implementations

Severity: Informational

Due to the usage of the `cosmwasm-storage` crate across the codebase, extra functions to support read and write operations on data types have been implemented. The newer `cw-storage-plus` crate offers most of those functionalities out of the box.

For example, the `singleton` type may be replaced with the `Item` type of `cw-storage-plus`, removing the need of having the `store_state` and `read_state` methods in `contracts/registry-stake/src/state.rs`.

Recommendation

We recommend using the `cw-storage-plus` crate for storage management. More details can be found in the official documentation https://crates.io/crates/cw-storage-plus

Status: Resolved