Audit Report

Atlo

v1.0

May 9, 2022

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 Liftoff Labs Inc. to perform a security audit of Atlo prefund 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/ATLO-Labs/atlo-prefund

Commit hash: `d7c417a39cac3a5d0cb01c92509a166455a824e2`

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

The submitted code implements the Atlo account and prefund smart contracts which are both used to support token launches. The account contract acts as the entry point for users while the prefund contract is responsible to hold the launcher's funds.

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 improve 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 a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	User's existing balance will be inaccessible if `UpdateAccounts` is called with a new denom	Critical	Resolved
2	Fee denom deposited by user will be inaccessible if excluded from configuration	Major	Resolved
3	Migration might fail due to out of gas error	Minor	Resolved
4	Incorrect deposit history will be logged when users deposit into a prefund	Minor	Resolved
5	Guaranteed allocation size might be incorrectly overwritten when updating configurations	Minor	Resolved
6	Missing logical validations might cause unexpected outcomes	Minor	Resolved
7	Missing address validations might lead to failures of execution handlers	Minor	Resolved
8	Missing tax deductions	Minor	Acknowledged
9	`try_update_accounts` allows arbitrary balance updates	Minor	Partially Resolved
10	Lowercasing denoms will cause issues with IBC tokens	Minor	Acknowledged
11	Prefund deposit amount needs to be higher than intended	Informational	Resolved
12	Possibility of duplicate denoms in configuration is inefficient	Informational	Resolved
13	Users can deposit into inactive/ended launchers	Informational	Resolved

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium-High	-
Test coverage	Medium-High	-

Detailed Findings

1. User's existing balance will be inaccessible if `UpdateAccounts` is called with a new denom

Severity: Critical

In `contracts/account/src/commands.rs:694-703`, a user's existing balance will be overwritten when the `denom` in the `BatchUpdateRequest` is not found in previously stored balances. This is caused by not including the existing balance when adding a new balance in line `721`. Consequently, balances with denoms that were sent previously will be stuck in the contract.

A test case demonstrating the above scenario can be found in appendix 2.

Recommendation

We recommend appending a user's existing balance when `update.denom` is not found in balances.

Status: Resolved

2. Fee denom deposited by user will be inaccessible if excluded from configuration

Severity: Major

In `contracts/account/src/commands.rs:332-347`, platform funds sent by the user will not be included as deposited funds. This is problematic since there is no validation that platform funds must be included in the allowed funds vector. As a result, users that deposit platform funds would not have their account balance updated, leading to a loss of funds.

A test case demonstrating the above scenario can be found in appendix 1.

Recommendation

We recommend verifying `fee_denom` is included in `denoms` during contract instantiation and update config phase.

Status: Resolved

3. Migration might fail due to out of gas error

Severity: Minor

In `contracts/account/src/contract.rs:183-187`, the `for` loop tries to process all launchers stored inside the contract. Since launchers are unbounded and cannot be removed, sooner or later the launchers' storage might grow too big to process. Consequently, this might cause the migration transaction to fail due to an out of gas error.

We classify this issue as minor since it can only be caused by the owner.

Recommendation

We recommend changing the migration logic into a state machine to guarantee consistency of migrations. For example, this could be achieved by automatically setting a `migration_pending` boolean when the migration process starts, and also automatically removing the `migration_pending` boolean once the old unbonded launchers have no more entries, which indicates that the migration is finished.

Status: Resolved

4. Incorrect deposit history will be logged when users deposit into a prefund

Severity: Minor

In `contracts/account/src/commands.rs:364-373`, the deposit history of users that deposited into a prefund will be incorrect.

Firstly, the `launcher_id` value is hardcoded to `None` in the `FundingHistory` although the user deposits to a prefund in line `246`.

Secondly, if `config.fee_denom` is not included in `config.denoms` vector, the deposit history will be empty because it is ignored, see lines `332` to `347`.

Thirdly, the deposit history will record the user's deposit to be lower than intended due to double platform fee deduction if the `config.fee_denom` is included in `config.denoms` vector (see lines `295-300` and `332-336`).

In result, this would cause the `Investor` query message to return an incorrect `deposit_history` value as seen in `contracts/account/src/queries.rs:72`.

The above scenarios can be demonstrated with test cases which can be found in appendix 3.

Recommendation

We recommend several preventive measurements such as modifying the implementation to dynamically set `launcher_id` to user's choice of prefund in line `246`, verifying

`config.fee_denom` is included in `config.denoms` during contract instantiation and config updates, and lastly refactoring the code to only deduct platform fee once when users deposits into a prefund.

Status: Resolved

5. Guaranteed allocation size might be incorrectly overwritten when updating configurations

Severity: Minor

In `contracts/prefund/src/commands.rs:80-94`, the value of `state.guaranteed_allocation_size` is forcefully updated even if there's no valid update to the corresponding `max_participants` value. If `guaranteed_allocation_size` is instantiated to contain a valid `Some` value and one of the admins decides to update configurations that aren't related to it (e.g. the name or description), the value of `guaranteed_allocation_size` will be overwritten to `None` in line `93`. As a result, the `State` query message will return an incorrect value in `contracts/prefund/src/queries.rs:46`.

A test case demonstrating the above scenario can be found in appendix 4.

Another possible scenario that leads to an incorrect value of the `guaranteed_allocation_size` is when `updated_config.total_tokens_for_sale` has a `Some` value while the `updated_config.max_particpants` has a `None` value. In that case, `state.guranteed_allocation_size` is set to `None` whilst its value should get updated to the new value of `total_tokens_for_sale`.

Recommendation

We recommend preserving the value of `guarantee_allocation_size` when `max_participants` and `total_tokens_for_sale` is provided as `None` while calculating the new value when either `max_participants` or `total_tokens_for_sale` is a `Some` value.

Status: Resolved

6. Missing logical validations might cause unexpected outcomes

Severity: Minor

In the prefund contract, there are several config values that are currently not validated. For example, the value of `msg.end_date` should be greater than `msg.start_date` and the value of `msg.max_prefund` should be greater than `msg.min_prefund`. If any of these values are configured incorrectly, it would cause the contract to be unusable. For example,

the `try_allocate` function would fail in `contracts/prefund/src/commands.rs:123-129` if `msg.start_date` is greater than `msg.end_date`.

Recommendation

We recommend verifying `msg.end_date` to be greater than `msg.start_date` and `msg.max_prefund` to be greater than `msg.min_prefund` in `contracts/prefund/src/contract.rs:61-62`, `66-67`, `contracts/prefund/src/commands.rs:63-64` and `76-77`.

Status: Resolved

7. Missing address validations might lead to failures of execution handlers

Severity: Minor

In the prefund contract, there are addresses in config values that are not validated before storing. Incorrect addresses would lead to issues when executing functions such as `whitelist_contract`. This issue exists during both the instantiation of the prefund contract in `contracts/prefund/src/contract.rs:46-69` and updates to the config in `contracts/prefund/src/commands.rs:43-78`.

Recommendation

We recommend validating the `admins`, `account_contract`, `mainfund_contract`, `kyc_contract`, and `whitelist_contract` addresses using `deps.api.addr_validate` before storing them.

Status: Resolved

8. Missing tax deductions

Severity: Minor

While Terra's tax rate has been set to zero, the tax mechanism is still implemented and the rate might be increased again in the future. It is still best practice to include functionality to deduct taxes.

A non-zero tax rate could be reinstated via a governance proposal due to circumstances where the expected income from the tax rewards increases significantly. In this situation, stablecoin transactions on Terra would expand to a state where a meaningful portion of the staking rewards income is derived from tax rewards rather than the vast majority coming from swap fees.

We consider this to only be a minor issue since the contract owner can recover from tax mismatches by simply sending funds back to the contract. Additionally, the likelihood of the Terra team to increase taxes again is low.

See this discussion for more details about the tax rate changes on Terra.

Recommendation

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

Status: Acknowledged

9. `try_update_accounts` allows arbitrary balance updates

Severity: Minor

In `contracts/account/src/commands.rs:666`, the admins are able to increase any existing investor's balance without any restriction limit. The team states that the amount of allocation is determined by their own Atlo rating which is calculated via an off-chain script. Since there's no validation in place (e.g. verifying the investor did invest into the prefund or making sure the refund amount is equal to or lower than the deposited amount), a miscalculation in the script might cause the investor to get more/less tokens than intended.

Recommendation

We recommend adding several logic validations to `try_update_accounts` functionality such as:

- Add a check to validate whether the contract has sufficient balance to successfully execute a refund without shortfalling the contract balance related to pending investments.
- Validate the refund amount should be equal to or lower than the investor's deposited amount via analyzing their prefund history for the specific launcher.

Status: Partially Resolved

The Atlo team states that they implemented an off-chain validation ensuring the correct funds are in place, hence only the second recommendation point has been implemented.

10. Lowercasing denoms will cause issues with IBC tokens

Severity: Minor

In `contracts/account/src/commands.rs`, there are several instances where token denoms are converted to lowercase via `to_lowercase`. Since token denoms are case sensitive, the contract will be unable to support IBC tokens.

Recommendation

We recommend not lowercasing denoms.

Status: Acknowledged

The Atlo team states that they have no current plans to use this contract with IBC tokens, and will implement this recommendation if needed in the future.

11. Prefund deposit amount needs to be greater than intended

Severity: Informational

In `contracts/account/src/commands.rs:333-316`, user's deposited funds are deducted twice even though they have already been deducted in lines `295` to `300`. This causes an unnecessary requirement that the prefund deposit amount must be greater by twice the value of platform fees. There might be a possibility that the user's deposit is rejected due to insufficient funds.

Recommendation

We recommend only deducting the platform fee once for pre-fund deposits.

Status: Resolved

12. Possibility of duplicate denoms in configuration is inefficient

Severity: Informational

In `contracts/account/src/contract.rs:43`, denoms are added into the config during contract instantiation phase. If the admin decides to add custom denoms via passing a valid `msg.denoms` vector, the corresponding `denoms` function will not remove duplicate denominations from the vector. Consequently, there is a possibility that the same denom is added twice to the `config.denoms` vector, which is inefficient.

Recommendation

We recommend removing duplicate denoms in `contracts/account/src/helpers.rs:8-13`.

Status: Resolved

13. Users can deposit into inactive/ended launchers

Severity: Informational

In the accounts contract, a value is stored to keep track in which state the launcher is. However, the launcher's state does not get automatically updated nor does it get used to validate whether the given launcher id should receive investments or not. Hence, an investor may invest in a launcher that has ended, which would lead to a failure when allocating the funds, because the prefund contract checks for the `end_date`. To resolve this issue, the admin would need to withdraw funds from the prefund contracts individually.

Recommendation

We recommend using the launcher state to restrict users to only invest in launchers that have not ended.

Status: Resolved

Appendix

Test case for issue 1

```
#[test]
fn test_deposit_none_not_in_denom() {
    // file: contracts/account/src/tests.rs
    let mut deps = mock_dependencies(&[]);
    let info = mock_info("creator", &coins(1000, "earth"));

    let fee = Uint128::from(100 as u32);
    let msg = custom_instantiation(
        None,
        Some(vec!["uusd".to_string()]),
        Some(true),
        Some(fee),
        Some("uluna".to_string()),
    );

    let _res = instantiate(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    let funds_uluna = coin(1_000, "uluna");

    let user1 = mock_info(USER1, &vec![funds_uluna.clone()]);
    let msg = ExecuteMsg::Deposit {
        launcher_id: None,
        contract_id: None,
    };
    let _res = execute(deps.as_mut(), mock_env(), user1, msg).unwrap();

    let result = query(
        deps.as_ref(),
        mock_env(),
        QueryMsg::Investor {
            wallet: Addr::unchecked(USER1),
        },
    )
    .unwrap();
    let value: InvestorResponse = from_binary(&result).unwrap();

    // no balance recorded
    assert_eq!(value.balances, vec![]);
}
```

Test case for issue 2

```
#[test]
fn test_update_accounts_new_denom() {
    // file: contracts/account/src/tests.rs
    let mut deps = mock_dependencies(&[]);
    let admin1 = mock_info(ADMIN1, &coins(1000, "uusd"));
    let info = mock_info("creator", &coins(1000, "earth"));
```

```
// set config.denoms to include uusd and uluna
let msg = custom_instantiation(
    None,
    Some(vec!["uusd".to_string(), "uluna".to_string()]),
    Some(true),
    None,
    None,
);

let _res = instantiate(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();
let msg = ExecuteMsg::SetLauncher {
    id: "1-launcher".to_string(),
    name: Some("LauncherOne".to_string()),
    state: Some(LauncherState::Active.value()),
    contracts: Some(vec![LauncherContract {
        id: "1-contract".to_string(),
        name: "Contract".to_string(),
        address: Addr::unchecked("CONTRACT"),
    }]),
    vesting: Some(Addr::unchecked("vesting_contract")),
    proposal_id: Some(1),
};
let _res = execute(deps.as_mut(), mock_env(), admin1.clone(), msg).unwrap();

// user deposit uluna
let user1 = mock_info(USER1, &coins(1_000, "uluna"));
let msg = ExecuteMsg::Deposit {
    launcher_id: None,
    contract_id: None,
};
let _res = execute(deps.as_mut(), mock_env(), user1.clone(), msg).unwrap();

// verify investor balance to include uluna
let result = query(
    deps.as_ref(),
    mock_env(),
    QueryMsg::Investor {
        wallet: Addr::unchecked(USER1),
    },
)
.unwrap();
let value: InvestorResponse = from_binary(&result).unwrap();
assert_eq!(
    value.balances,
    vec![InvestorBalance {
        amount: Uint128::from(1_000_u64),
        denom: "uluna".to_string()
    }] as Vec<InvestorBalance>
);

// admin updates user account to increase uusd balance
let msg = ExecuteMsg::UpdateAccounts {
    withdrawal: vec![BatchUpdateRequest {
```

```
            wallet: Addr::unchecked(USER1),
            amount: Uint128::from(500 as u32),
            denom: "uusd".to_string(),
            launcher_id: "1-launcher".to_string(),
        }],
    };
    let _res = execute(deps.as_mut(), mock_env(), admin1.clone(), msg).unwrap();

    let result = query(
        deps.as_ref(),
        mock_env(),
        QueryMsg::Investor {
            wallet: Addr::unchecked(USER1),
        },
    )
    .unwrap();
    let value: InvestorResponse = from_binary(&result).unwrap();

    // user uusd balance increased but uluna balance is missing
    assert_eq!(
        vec![InvestorBalance {
            amount: Uint128::from(500 as u32),
            denom: "uusd".to_string()
        }],
        value.balances
    );
}
```

Test case for [issue 3]

```
#[test]
fn test_send_fee_with_not_in_denom() {
    // please paste in contracts/account/src/tests.rs

    // setup
    let mut deps = mock_dependencies(&[]);
    let admin1 = mock_info(ADMIN1, &coins(1000, "uusd"));

    let info = mock_info("creator", &coins(1000, "earth"));
    let fee = Uint128::from(100 as u32);

    let msg = custom_instantiation(
        None,
        Some(vec!["uusd".to_string()]),
        Some(true),
        Some(fee),
        Some("uluna".to_string()),
    );

    let _res = instantiate(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    let res = query(deps.as_ref(), mock_env(), QueryMsg::Config {}).unwrap();

    // verify fee is set to uluna which isn't included in value.denoms
```

```
let value: ConfigResponse = from_binary(&res).unwrap();
assert_eq!(vec!["uusd".to_string()], value.denoms);
assert_eq!(true, value.can_deposit);
assert_eq!(fee, value.fee_amount.unwrap());
assert_eq!("uluna".to_string(), value.fee_denom.unwrap());

// create launcher id and contract id
let msg = ExecuteMsg::SetLauncher {
    id: "1-launcher".to_string(),
    name: Some("LauncherOne".to_string()),
    state: Some(LauncherState::Active.value()),
    contracts: Some(vec![LauncherContract {
        id: "1-contract".to_string(),
        name: "Contract".to_string(),
        address: Addr::unchecked("CONTRACT_ADDR"),
    }]),
    vesting: Some(Addr::unchecked("vesting_contract")),
    proposal_id: Some(1),
};
let _res = execute(deps.as_mut(), mock_env(), admin1, msg).unwrap();

// user deposit funds with Some(lancher_id) and Some(contract_id)
let funds_uusd = coin(1_000_000, "uluna");
let user1 = mock_info(USER1, &vec![funds_uusd.clone()]);
let msg = ExecuteMsg::Deposit {
    launcher_id: Some("1-launcher".to_string()),
    contract_id: Some("1-contract".to_string()),
};
let res = execute(deps.as_mut(), mock_env(), user1, msg).unwrap();

// when executing Allocate only single fee deduction is deducted
let funds_uluna_after_fee = coin((funds_uusd.amount - fee).u128(), "uluna");
assert_eq!(
    res.messages,
    vec![SubMsg::new(Wasm(WasmMsg::Execute {
        contract_addr: "CONTRACT_ADDR".to_string(),
        msg: to_binary(&Allocate {
            investor: Addr::unchecked("USER1"),
        })
        .unwrap(),
        funds: vec![funds_uluna_after_fee]
    })),]
);

// query investor information
let result = query(
    deps.as_ref(),
    mock_env(),
    QueryMsg::Investor {
        wallet: Addr::unchecked(USER1),
    },
)
.unwrap();
let value: InvestorResponse = from_binary(&result).unwrap();
```

```
    // empty deposit history
    assert_eq!(value.deposit_history, vec![]);
}


#[test]
fn test_send_fee_with_some() {
    // please paste in contracts/account/src/tests.rs

    // setup
    let mut deps = mock_dependencies(&[]);
    let admin1 = mock_info(ADMIN1, &coins(1000, "uusd"));

    let info = mock_info("creator", &coins(1000, "earth"));
    let fee = Uint128::from(100 as u32);

    let msg = custom_instantiation(
        None,
        Some(vec!["uusd".to_string()]),
        Some(true),
        Some(fee),
        Some("uusd".to_string()),
    );

    let _res = instantiate(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    let res = query(deps.as_ref(), mock_env(), QueryMsg::Config {}).unwrap();

    // verify fee is set
    let value: ConfigResponse = from_binary(&res).unwrap();
    assert_eq!(vec!["uusd".to_string()], value.denoms);
    assert_eq!(true, value.can_deposit);
    assert_eq!(fee, value.fee_amount.unwrap());
    assert_eq!("uusd".to_string(), value.fee_denom.unwrap());

    // create launcher id and contract id
    let msg = ExecuteMsg::SetLauncher {
        id: "1-launcher".to_string(),
        name: Some("LauncherOne".to_string()),
        state: Some(LauncherState::Active.value()),
        contracts: Some(vec![LauncherContract {
            id: "1-contract".to_string(),
            name: "Contract".to_string(),
            address: Addr::unchecked("CONTRACT_ADDR"),
        }]),
        vesting: Some(Addr::unchecked("vesting_contract")),
        proposal_id: Some(1),
    };
    let _res = execute(deps.as_mut(), mock_env(), admin1, msg).unwrap();

    // user deposit funds with Some(lancher_id) and Some(contract_id)
    let funds_uusd = coin(1_000_000, "uusd");
    let user1 = mock_info(USER1, &vec![funds_uusd.clone()]);
```

```
    let msg = ExecuteMsg::Deposit {
        launcher_id: Some("1-launcher".to_string()),
        contract_id: Some("1-contract".to_string()),
    };
    let res = execute(deps.as_mut(), mock_env(), user1, msg).unwrap();

    // when executing Allocate only single fee deduction is deducted
    let funds_uusd_after_fee = coin((funds_uusd.amount - fee).u128(), "uusd");
    assert_eq!(
        res.messages,
        vec![SubMsg::new(Wasm(WasmMsg::Execute {
            contract_addr: "CONTRACT_ADDR".to_string(),
            msg: to_binary(&Allocate {
                investor: Addr::unchecked("USER1"),
            })
            .unwrap(),
            funds: vec![funds_uusd_after_fee]
        })),]
    );

    // query investor information
    let result = query(
        deps.as_ref(),
        mock_env(),
        QueryMsg::Investor {
            wallet: Addr::unchecked(USER1),
        },
    )
    .unwrap();
    let value: InvestorResponse = from_binary(&result).unwrap();

    // however in deposit_history, double tax deduction happened
    let double_tax_deducted = coin((funds_uusd.amount - fee - fee).u128(), "uusd");
                                assert_eq!(Uint128::from(double_tax_deducted.amount),
value.deposit_history.first().unwrap().amount);
}
```

Test case for [issue 4]

```
#[test]
fn test_update_config() {
    // file: contracts/prefund/src/tests/tests.rs
    let mut deps = mock_dependencies(&[]);
    let info = mock_info("creator", &coins(1000, "earth"));
    let _res = instantiate(
        deps.as_mut(),
        mock_env(),
        info.clone(),
        default_instantiation(),
    )
    .unwrap();

    let res = query(deps.as_ref(), mock_env(), QueryMsg::State {}).unwrap();
    let state: StateResponse = from_binary(&res).unwrap();
```

```
    assert_eq!(
        Some("16000.0000493822".to_string()),
        state.guaranteed_allocation_size
    );

    let admin1 = mock_info(ADMIN1, &coins(1000, "earth"));

    let msg = ExecuteMsg::UpdateConfig {
        config: UpdateConfigRequest {
            admins: None,
            id: None,
            name: None,
            description: None,
            logo: None,
            token: None,
            enabled: None,
            account_contract: None,
            mainfund_contract: None,
            kyc_contract: None,
            whitelist_contract: None,
            must_check_whitelist: None,
            must_check_kyc: None,
            max_participants: None,
            total_tokens_for_sale: None,
            start_date: None,
            end_date: None,
            suggested_amount: None,
            price: None,
            denom: None,
            min_prefund: None,
            max_prefund: None,
            can_withdraw: None,
            can_deposit: None,
        },
    };
    let _res = execute(deps.as_mut(), mock_env(), admin1.clone(), msg);

    let res = query(deps.as_ref(), mock_env(), QueryMsg::State {}).unwrap();
    let state: StateResponse = from_binary(&res).unwrap();

    // autoset as None despite no new config provided
    assert_eq!(state.guaranteed_allocation_size, None);
}
```