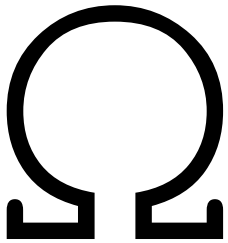

Team Omega

https://teamomega.eth.limo

Giza ARMA Audit

Final Report

June 20, 2025

Summary

Giza has asked Team Omega to audit their Backend code of the ARMA system.

We found **no high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **3** issues as “medium” - these are issues we believe you should definitely address. In addition, **14** issues were classified as “low”, and **5** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Severity	**Number of issues**	**Number of resolved issues**
High	0	-
Medium	2	1
Low	15	10
Info	5	2

The current document is a preliminary audit report. Giza will read this and fix any issues they choose to fix, and Omega will subsequently review the fixes and write a final report.

Team Omega
Team Omega (https://teamomega.eth.limo/) specializes in Smart Contract security audits on the Ethereum ecosystem.

Giza
Giza.tech (https://www.gizatech.xyz/) is developing a platform for AI solutions in web3

Scope of the Audit
Team Omega audited the software from the following two repositories and commits:

```
https://github.com/gizatechxyz/agent_mode/commit/9e7e0bc30fc1fdc17682d72fd848a65ccbde2313
```

```
https://github.com/gizatechxyz/arma-contract-interactions/commit/731405cf
ecee5a89b3c33e2eec0c6ddae94cff5a
```

We have audited earlier iterations of this code on other occasions. In the current audit, we focused on the changes since our last audit, which we did in February 2025. That audit report reviewed the state of the repository at the following commits:

```
https://github.com/gizatechxyz/agent_mode/commit/188e4100c3e5198dc3cdc221
ca957fd04e58aff5
```

```
https://github.com/gizatechxyz/arma-contract-interactions/commit/3b608158
6049db825ceae7ecd2f40e9e479f861d
```

Methodology
The audit report has been compiled on the basis of the findings from different auditors (Jelle and Ben). The auditors work independently. Each of the auditors has several years of experience in developing smart contracts and web3 applications.

Liability
The audit is on a best-effort basis. In particular, the audit does not represent any guarantee that the software is free of bugs or other issues, nor is it an endorsement by Team Omega of any of the functionality implemented by the software.

Resolution

After we delivered a preliminary report, most of the issues were addressed by the team at GIZA. The relevant commits are:

```
https://github.com/gizatechxyz/agent_mode/commit/fbbb36a12302173f97e97362
dda2e1022729c3cd
```

```
https://github.com/gizatechxyz/arma-contract-interactions/commit/1732ea0c
6624eed98146d49cfd6ad964b3990f18
```

We checked for each of the issues in our report if the issue was addressed (or not) and marked our observations below. We did not audit the other changes that were made.

Contract Interactions - arma-contract-interactions

C1. Euler rewards price estimate may be overestimated [low] [resolved]

A large part of the rewards in the Euler ecosystem currency come from rewards paid out in rEUL. This is a reward token that can be exchanged for EUL at maturity after an amount of time, counting from the moment the rewards are claimed. Before that, the tokens are not transferable.
In the functions calculating the rewards and the performance, the price of rEUL is assumed to be the same as EUL - cf. for example the definition of `TOKEN_PRICE_OVERRIDES`.
This seems overly optimistic. Surely, rewards in rEUL, not being liquid, are not as attractive as rewards paid out in liquid tokens.
Recommendation: You could count these rewards as being worth less in the calculations of APR and in the decision logic for choosing the protocols to invest in. Alternatively (or in addition) make sure your users know the type of reward that is being paid out.
Severity: Low
Resolution: The issue was resolved. The rEUL value is now discounted by a fixed 10%.

C2. Approval logic may fail when using USDT [low] [not resolved]

The approval logic in `tx_manager.py` works as follows: it checks whether enough tokens are approved, and if that is not the case, it will call the `approve` method with the desired amount of tokens to be approved.
For certain token implementations, such as USDT, a call to `approve` with a non-zero amount will revert if the current approval is higher than zero. The idea here is that a user must first reset the approval to 0 before setting it to a higher value.
Currently, all the integrated vaults use USDC, and so the problem does not occur. But if ARMA plans to integrate USDT vaults in the future - as is the case - it is important to make the logic work for these cases as well.
Recommendation: Check if current approval is 0, and if that is not the case, set the approval to 0 before setting it to a desired value.
Severity: Low
Resolution: The issue was not resolved.

C3. Stablecoin swaps may fail even if available for a fair price [low] [resolved]

In `tx_manager.py`, the function `_calculate_swap_min_out` takes a price quote and returns a value for “the minimum amount out for the swap transaction”. Essentially, to calculate that value, it uses the best price between the price quote given, and a hardcoded price given by `1 - MAX_SLIPPAGE_RATIO`.

This logic is problematic for several reasons.
First of all, the constant `MAX_SLIPPAGE_RATIO` is used for two different purposes (which may require different values). In the ETH conversion logic, constant is used for the commonly used sense of the term "slippage" (the difference between the price quote and the actual price that you get), while here, it represents a hardcoded limit on the price itself. There is no real relationship between the maximum acceptable amount of slippage on a trade of ETH and the acceptable deviation of prices between two stable coins.
Secondly, the effect of using a lower price than the one given by the price quote will almost surely make your swap transaction fail, and so it is unclear what the point is of trying to execute the swap in any case.
Thirdly, the current logic does not allow for any slippage of the price with respect to the price quote at all, which does not seem to be the intention here.
Recommendation: Rethink the "slippage protection" implementation for stablecoin swaps.
Severity: Low
Resolution: The issue was resolved. In fact, all the stablecoin swap logic was removed.

C4. APY is overestimated in Euler vault [low] [resolved]

In `euler_vault.py`, the function `get_adjusted_apr(amount)` is expected to return the APR obtained on an investment of `amount` tokens in the Euler vault. The calculation ultimately depends on a call to the function `computeAPYs` in a helper contract provided by Euler, and providing the function with the current interest rates and borrows, and adding `amount` to the cash argument.
However, this calculation does not take into account the fact that borrow rates will go down if the LTV of the vault diminishes, and so it will overestimate the actual APR returned on depositing `amount` tokens in the vault.
For reference, the function is defined here
https://github.com/euler-xyz/evk-periphery/blob/9be812e79ce74fa2f852d9afb678d483d3bbb31e/src/Lens/Utils.sol#L79
Clearly, the calculations of APRs on vault deposits in Euler are estimates - the actual APR returned depends on information that is not currently available (such as future deposits and borrows), and so some drift may be acceptable here. However, on relatively large deposits, the difference here may be significant.
Recommendation: To correctly implement this function, you could use `getVaultInterestRateModelInfo` from the Euler Lens, which will take the `InterestRateModel` into account when calculating the APY on a cash deposit.
Severity: Low
Resolution: The issue was resolved as recommended.

C5. APR on extra rewards is underestimated in Euler vault [low] [resolved]

The function `_get_extra_rewards_apr` calls the `EULER_REWARDS_API` endpoint, which returns the APR on extra rewards (and not the APY, which makes sense, as these rewards do not compound). However, this value is treated as if it were an APY, and is run through the APY-to-APR conversion, and so the actual number returned is lower than it should be.
Recommendation: Omit the APY-to-APR calculation, and just return the value that the API gives you
Severity: Low
Resolution: The issue was resolved as recommended.

C6. Euler's get_extra_rewards does not take into account that rewards may end soon [low] [resolved]

In `get_extra_rewards` in `euler_vault.py`, it is checked that the reward program is currently active:

```
      reward.get("startTimestamp", 0) <= current_timestamp <=
reward.get("endTimestamp", 0):
```

This does not take into account the case that rewards may end (very) soon.
Recommendation: We would recommend checking that the rewards program is not about to end just now - perhaps a 24 hour or 7 day timeframe would be reasonable to consider here.
Severity: Low
Resolution: The issue was resolved as recommended. The function now omits rewards which are about to end within 24 hours.

C7. Approve and swap ETH logs wrong max USDC amount [info] [resolved]

In `approve_and_swap_eth`, the `max_usdc` used for logging the max amount of USDC that can be swapped is divided by `USDC_DECIMALS`, which is 6, when it should be dividing by 10 ** `USDC_DECIMALS` (ie. 1e6).
Recommendation: Fix the calculation to do correct division.
Severity: Info
Resolution: The issue was resolved. The function was removed.

C8. Check ETH balance return in stablecoin option returns normal USD [info] [resolved]

The `check_eth_balance` function has a `return_in_stablecoin` option, which returns the balance in USD terms. This however uses normal decimals, and not stablecoin decimals, which makes the name of the option misleading and could lead to errors.
Recommendation: The function does not seem to be in use, but if you intend to keep it, we recommend that you rename the option to `return_in_usd`, or return in proper USDC decimals.
Severity: Info
Resolution: The issue was not resolved. The function was removed.

C9. Lending protocols should verify token passed in get token is supported [info] [resolved]

Multiple lending protocols implementing the `get_token` function do not have a check to verify the token passed is supported, which could lead to potential issues in the future.
Recommendation: Euler for example does check that the token passed to `get_token` is supported, we recommend checking it in the same way for the other protocols.
Severity: Info
Resolution: The issue was resolved as recommended.

C10. Wrong error in approve_and_swap_stablecoin [info] [resolved]

In `approve_and_swap_stablecoin`, if a `NotImplementedError` is raised during the approval, this error is intercepted, and instead an `ArmaContractsTxSimulatorError` is raised. This seems wrong.
Recommendation: Just raise the original error.
Severity: Info
Resolution: The issue was resolved. The `approve_and_swap_stablecoin` function was removed.

Giza Agent Mode - giza-agent_mode

A1. Rewards are only claimed when deactivating a wallet [low] [not resolved]

The `ClaimRewards` actions are only executed as part of `deactivate` logic, which means that users will not get these rewards until the wallet is deactivated.
Recommendation: Claim rewards any time a protocol is exited - i.e. also when re-allocating funds. Or allow for a user to claim rewards manually.

Severity: Low
Resolution: The issue was not resolved. However, the team confirmed this is the expected behaviour for this version, and will try to improve it in the future.

A2. Top-up logic is confusing and can be abused [medium] [resolved]

In `wallet_manager.py`, in the `_get_portfolio` function, if the `is_topup` argument is `false` and the wallet has a positive balance of non-deposited tokens, the following happens: the total amount of transfers of tokens from EOA accounts from the past 24 hours is collected from the blockchain, and that amount is then added as a new deposit to the database.
This logic seems quite arbitrary and fragile. First of all, it is unclear why the 24-hour period, and the fact that the money comes from an EOA account (instead of, say, from a Safe) is relevant. But, more importantly, by creating Deposits in the database each time this function is called – which is on each `run` action it can easily happen that there are more deposits than actual contributions to the portfolio. This could happen, for example, if the `run` action is called twice in a 24-hour period, or if the `run` action fails during re-allocation, and so the non-deposited are counted twice.
This is important, as the deposits are used in the APR calculations and other statistics, but are also used to calculate the fees that a user pays (which are based on the portfolio gains).
Recommendation: Review this logic.
Severity: Medium
Resolution: The issue was resolved. The logic was refactored to look for all transfers in the last 48 hours. However, the `top_up` function unnecessarily calls the `register_new_deposit` function (even though run will call it later as well), but it calls it without first verifying the transaction was not already registered as a deposit, which means the function may register the same deposit multiple times.

A3. The top_up endpoint can be used to avoid paying fees [medium] [partially resolved]

The `top_up` endpoint as defined in `agents.py` simply writes a `Deposit` object with the provided amount to the database if the amount is lower or equal to the wallet's balance. The end-point can be called by the owner of the wallet. This means that a user can manipulate the `Deposit` entry by sending a small amount of tokens to the wallet and then calling the `top_up` endpoint several times.
This is important, as the deposits are used in the APR calculations and other statistics, but are also used to calculate the (optional) fees that a user pays (which are based on the portfolio gains).
Recommendation: Review this logic.
Severity: Medium
Resolution: The issue was partially resolved. The `amount` now comes directly from a provided transaction hash, but proper protection from duplicated calls seems missing (see issue above).

A4. Protocols monitor does not monitor Euler [low] [resolved]

The monitor client defines the protocols to monitor in a list of `PROTOCOLS`. However, Euler seems to be missing from the list, and so won't be monitored.
Recommendation: Add the relevant Euler vault to the protocols list of monitor client.
Severity: Low
Resolution: The issue was resolved as recommended.

A5. Missing protocols in TVL monitoring [low] [resolved]

The `TVLMonitor` class uses a list of `WHITELISTED_PROTOCOLS` to filter the protocols to monitor. However there are 2 issues with the list. First, the name for the Seamless vault seems wrong. The name for it in the API is Seamless Vaults, while the name in the list is Seamless Protocol, a name which doesn't exist in the API. Second, Euler is missing from the list, and so won't be monitored.
Recommendation: Update the name for Seamless Vaults, and add the relevant Euler vault to the whitelisted protocols list of the `TVLMonitor`.
Severity: Low
Resolution: The issue was resolved as recommended.

A6. Wallet status updated to running before checking for failed deactivation [low] [resolved]

When calling the `run_wallet_job`, the code first updates the wallet status to `RUNNING`, and then checks if the status is `DEACTIVATION_FAILED`, which it can no longer be. This means that when called for a wallet with `DEACTIVATION_FAILED` status, instead of deactivating, the wallet will be run normally and essentially be reactivated.
Recommendation: Check for the `DEACTIVATION_FAILED` status before updating the wallet status to `RUNNING` in `run_wallet_job`.
Severity: Low
Resolution: The issue was resolved as recommended.

A7. Gas costs for deposit and withdraw actions depend on the protocol [low] [not resolved]

In `optimizer.py`, gas amounts that are used for calculating the costs for approval, depositing and withdrawing from a protocol are read from the database. The amounts returned are, for each type of

action, the sum of the “L1 costs” and the “L2 costs” as stored in the database. This way of storing gas costs as the sum of these two values is confusing (it is unclear to us why these two values are summed), but, more importantly, gas costs for deposit and withdrawal can differ significantly between protocols, and this is not taken into account by the optimizer.
Recommendation: It is easy to simulate the gas used by these transactions, or to find historical values on-chain. We recommend using those values.
Severity: Low
Resolution: The issue was not resolved.

A8. APR is overestimated if the total amount is below MAX_AMOUNT_PER_WALLET_FOR_OPTIMIZER [low] [resolved]

In `optimizer.py`, in `_build_piecewise`, if `T_max < MAX_AMOUNT_PER_WALLET_FOR_OPTIMIZER`, the value for the APR used is the base APR instead of the adjusted APR. The base APR is typically higher than the adjusted APR, but the adjusted APR includes extra rewards which the base APR does not. The code block does not check for the given `tolerance`, so the difference could be significant.
Recommendation: The calculations would be more precise if the code block is simply removed.
Severity: Low
Resolution: The issue was resolved. The base APR now returns with the extra rewards, so the difference should no longer be significant.

A9. The piecewise approximation systematically underestimates the actual APR [low] [not resolved]

In `optimizer.py`, in `_build_piecewise`, if the difference between the base APR and the APR when depositing the entire available sum is higher than a given `tolerance`, the representation is split into 3 equal tranches (one for the first 33%, one for the first 66%, and one for 100%), for which the APR is calculated.
The difference in APR between the three tranches can be significant, as the `tolerance` is not checked for the difference between the tranches. If the curve is steep, this may mean that the optimization algorithm takes into account a lower APR than can be actually obtained (for example, suppose the first tranche consists of 1000 tokens, of which the first 100 give a 50% APR, and the last 900 give 0% APR. The algorithm will see this as potential to invest 1000 tokens for a 5% APR, and may not allocate any tokens at all, missing out on the 50%.
Recommendation: Use more fine-grained pieces (i.e. build more pieces) if the difference between the pieces exceeds the `tolerance`. We realize that this makes the optimization problem harder, but it can certainly be optimized in other ways (for example, the allocation into *k* tranches within a protocol is currently represented by *k+2* variables (k +1 “lambda vars” and an “x_var”; it can be more concisely

represented with 2 variables - the x_var representing the amount to be allocated to the protocol, and a new variable < k that denotes the tranche on the basis of which the APR is calculated).
Severity: Low
Resolution: The issue was not resolved. The team decided to not fix the issue as the improvement was not deemed enough to justify the extra computation and complexity.

A10. Gas costs for claiming rewards are not taken into account [low] [not resolved]

Claiming rewards may incur significant gas costs. These costs are relevant in two different places:

1. Gas costs for claiming rewards are not taken into account when calculating the APR, and so are not counted in the optimization problem.
2. Gas costs for claiming rewards are not taken into account when claiming the rewards - in particular, if the gas costs exceed the value of the reward, it is better not to claim these at all.

Recommendation: Take the gas costs for claiming rewards into account both when calculating APR, as well as when deciding to claim rewards.
Severity: Low
Resolution: The issue was not resolved, but the team committed to resolve it in a future version, and further mentioned that currently gas is sponsored, which means for now it will not be an issue.

A11. Excluded protocols constraint is ignored on activation [low] [resolved]

The `run` function of `WalletManager` has a check before adding the exclude protocol constraint that skips adding the constraint in case there are no current protocols. This means that upon activation, the first run may not have the constraint, and so may deposit into protocols which are supposed to be excluded.
Recommendation: Remove the condition and always add the constraint.
Severity: Low
Resolution: The issue was resolved. The team has clarified this constraint is not meant to be used to exclude protocols on instantiation, but rather only protocols which are in existing use.

A12. Continuous Integration fails on github [info] [resolved]

In the version of the code we reviewed, the Github continuous integration action fails
Recommendation: Make sure your build process and tests run in the Github CI.
Severity: Info

Resolution: The issue was resolved as recommended.