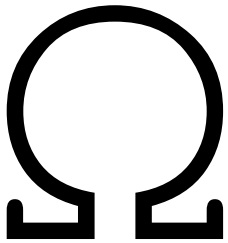

Team Omega

https://teamomega.eth.limo

Giza ARMA February Audit

Final Report

May 14, 2025

Summary

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

We wrote a preliminary report on February 26, 2025. Giza has subsequently addressed a significant number of issues.
The current document is an intermediate audit report - some issues of low severity remain unresolved, but are acknowledged by the team, and fixes will be applied in the next release.
We will write a final report once that release is ready.

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

In February 2025, Team Omega audited the software from the following two repositories and commits:

```
https://github.com/gizatechxyz/agent_mode/commit/6eda723cc50c4d7216120cba07de0a31963411b9

https://github.com/gizatechxyz/arma-contract-interactions/commit/e4d11cd3aaf5e9c677dd86270081a3f5885cc622
```

We mostly focused on the python code that defines the various services and interactions.

Resolution

The Giza team has subsequently addressed most of the issues in the following commits:

```
https://github.com/gizatechxyz/agent_mode/commit/188e4100c3e5198dc3cdc221ca957fd04e58aff5

https://github.com/gizatechxyz/arma-contract-interactions/commit/3b6081586049db825ceae7ecd2f40e9e479f861d
```

We have updated the issues below. A number of issues remain open, which Giza intends to address in future iterations of the code. These issues have been marked with the “in progress” label.

This iteration also contains some new code, code that was not part of the original review. Some of this code is not part of the production code base but is rather functionality for reporting or is intended for admins. This new code was not reviewed by us.

In the `arma-contract-interactions`, we found a number of issues pertaining to the logic for stablecoin swaps and to integrations relative to the Mode chain. The current release will be deployed on Base only, and will only use USDC-based vaults. These issues we found are not relevant anymore, and are removed from the report.

Specifically, this code that is out of scope for this report concerns the following directories in the `agent_mode` repository:

```
cli
reports
```

And furthermore a set of functions that are mostly related to calculating and storing transaction volume.

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.

Summary

We found **1 high severity issue** - 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. We did classify **13** issues as “low” - we believe the code would improve if these issues were addressed as well.

Severity	Number of issues	Fix in progress	Resolved
High	1		**1**
Medium	2	1	**1**
Low	13	3	**9**

Contract Interactions - arma-contract-interactions

C1. Decision processes may use outdated information on vault performance [high] **[resolved]**

The function `MorphoVaultsFactory.create_vaults` will query the Morpho API for information about the vaults, including volatile information on the state of the protocol such as APR or TVL. Then for each vault it calls the function `_get_or_create_vault`, which will check if `self._vaults` already includes a cached value for the vault, and will return the cached value if that value already exists.

This means that `create_vaults` may return outdated data on the state of the vault. This is a problem when these outdated values are used to make decisions, for example on where to allocate the money in `run_wallet_job`.
Recommendation: It is unclear to us what the intended usage is of the "cached" value in `self._vaults` is, as `_get_or_create_vault` takes the entire data structure as its argument (as opposed to an identifier). We'd recommend to either not cache the value at all, or, if a cache is needed, to use a proper cache-management library which will at least allow you to work with expiry dates.
Resolution: This issue was resolved as recommended.

C2. Query in create_vaults may miss some vaults, is indeterministic [medium] **[resolved]**

The GraphQL query in `MorphoVaultsFactory.create_vaults(target_vault_names)` will get the first 1000 vaults from the Morpho API, loop through them, and return the subset of vaults that have (1) their "normalized" name in `target_vault_names` and (2) use USDC as their asset.

There are two unlikely but potential issues with this logic. The first is that you may miss out on vaults once Morpho has more than 1000 vaults. The second problem is that that the name of the vault is not necessarily unique in the Morpho database (although it can be expected to be, this is not guaranteed) and this holds even more for the "normalization" of the name) - and because the result set is not ordered, if there is are two vaults with the same "normalized" name, you may get an arbitrary one of these vaults in your returned results (e.g. if there is a vault named "A B C" and another named "a-b-c", either one could be returned).
Recommendation: Add a where-clause to the query to filter by USDC addresses server side. This will optimize your query and make it extremely unlikely you will hit the 1000 limit. Do not assume that the name field is unique *modulo* normalization, but instead filter by addresses.
Resolution: This issue was resolved. The query itself now filters for the expected token, chain, and only whitelisted. In the next release, the team has also committed to use vault address instead of name to avoid potentially conflating duplicated names, and to add paginated queries in case there are ever too many vaults returned for the query.

C3. Gas limit logic could lead to failed transactions [low] [in progress]

The current logic for setting the gas limit of transactions uses a constant of `GAS_UNITS`, along with a multiplier (usually starting at 1), which is increased in case a transaction fails. This approach is highly wasteful, since if the limit is set too low, the gas from such attempts will be lost. Also, if the transaction fails for any other reason other than being out of gas, submitting again with a higher gas limit will only waste more fees unnecessarily.
Recommendation: Instead of using a constant and retries, it would be better to estimate the gas of the transaction first, and then send it once.

Severity: Low
Resolution: This issue was not resolved yet, but the team intends to include a fix in the next release.

C4. Big approval logic is inefficient [low] **[resolved]**

On wallet activation, if the system flag of `EXECUTE_BIG_APPROVAL` is `True` (currently seems `False` on both dev and production) the system will do an approval transaction for various contracts the user is expected to interact with frequently, with an approved amount of 25x the amount deposited. However, this logic for approval seems to offer worse trade offs than the alternatives of either giving an unlimited approval, or approving each interaction separately. Since if an approved spender is compromised, having an approval of many times over the amount managed in the wallet puts the user's funds at risk as in the same way as with unlimited approval, and since the approvals are for a limited amount, eventually they will have to be renewed, which will require the logic and (less, but still) some extra transactions.
Recommendation: Either change the option to do unlimited approvals, or remove the big approval logic and only approve the exact amount needed for each operation.
Severity: Low
Resolution: This issue was resolved, the big approval logic was removed.

C5. Morpho extra rewards are not accounted for in adjusted APR [low] [in progress]

The Morpho vaults have extra rewards, but these are not being accounted for in the adjusted APR calculation. Instead the helper function `_get_extra_rewards_apr` says Morpho doesn't have extra rewards, but the `get_extra_rewards_info` function clearly allows for claiming extra rewards for Morpho vault.
Recommendation: If possible, include the APR for the Morpho extra rewards in the adjusted APR calculations, or if not possible, at least update the comment in `_get_extra_rewards_apr` to mention that Morpho does have extra rewards.
Severity: Low
Resolution: This issue was not resolved yet, but the team intends to include a fix in the next release.

ARMA Backend - agent_mode

M1. Jobs API key can access any endpoint of any wallet [medium] [in progress]

The role of the Jobs API key is to periodically execute the "run" action of the system. However, the authentication system does not limit its access to just calling the `run` action. Instead, if a valid Jobs API key is passed, the system will skip any other authentication checks and grant full access to all system endpoints. This gives it much more power over the system than necessary, and it makes a potential leak of the key risky to the whole system.
Recommendation: Limit the use of the Jobs API key to only access the run action, and block it from accessing the other endpoints.
Severity: Medium
Resolution: The Giza team acknowledged the issue, but chose to not address this in the current version of the code as it is useful to address emergency situations. It will be addressed in the next release.

M2. Run recovery may use wrong amount [low] [resolved]

In `recovery_service.py`, on lines 123 and 188 in `_handle_run_failed_wallet` and `_handle_activation_failed_wallet`, when calling `activate_wallet`, the value for the token argument is correctly `wallet_info.current_token`, but the value for the `amount` that is passed is the balance of the original deposit amount.
Recommendation: Pass the current amount of the `current_token` (or the original balance of the the original token)
Severity: Medium
Resolution: The issue was resolved - the code was completely removed.

M3. Statistics don't handle deactivated wallets consistently [low] **[resolved]**

The `_get_wallet_stats` function in `total_statistics.py` counts `initial_deposits` of all wallets, including deactivated ones, but it filters them out of other statistics, such as the total transactions and the total deposits by token, which seems inconsistent.
Recommendation: Handle deactivated wallets consistently when calculating statistics.
Severity: Low
Resolution: This issue was not resolved yet. The team has shared a fix which we reviewed, which is to be included in the next release.

M4. get_last_transaction_hashes does not correctly filter out transactions without a hash [low] **[resolved]**

The `get_last_transaction_hashes` checks to filter out transaction hashes which are equal to `None`, but when getting the transaction hash, it gives it a default value of empty string (“”). This will mean it can never be `None`, and so the filtering will not work.
Recommendation: Fix the filtering or remove the default value assigned to the hash.
Severity: Low
Resolution: This issue was resolved. The related code was removed.

M5. get_last_transaction_hashes may return less transactions than expected [low] **[resolved]**

The `get_last_transaction_hashes` gets the last 10 transactions of a wallet, then filters out dry run transactions and those without a hash, which may leave the final list with less than the intended 10 transactions.
Recommendation: Filter the transactions when getting them from the database, or try to get more if post filtering you are left with less than 10.
Severity: Low
Resolution: This issue was resolved. The related code was removed.

M6. get_last_transaction_hashes does not return last transactions [low] **[resolved]**

The `get_last_transaction_hashes` returns up to 10 transactions from each action, and does so by going over the list of all wallets and getting the latest transactions from each. However, it doesn’t take into account the time of transactions as compared between wallets. So if it checks wallet A first, then wallet B, transactions from B may not be included in favour of transactions from A in the final result, even if B's transactions were later than A's
Recommendation: Get transactions based on time, and not just going by the order of the list of wallets.
Resolution: This issue was resolved. The related code was removed.

M7. Jobs API key validation is vulnerable to time analysis [low] **[resolved]**

The current authentication method for the Jobs API key checks equality of the string passed and the key saved in the system settings. This however introduces a vulnerability to time analysis, where an attacker may try to analyze response time of the check to get to the key used. It is safer to compare digests to prevent such potential analysis.
Recommendation: Use the `compare_digest` function instead of comparing raw strings.
Resolution: The issue was resolved as recommended

M8. Data endpoints don't require authentication [low] **[resolved]**

Data reading endpoints, such as the `get_wallet_information`, and `get_performance_chart_data` don't require any authentication, meaning anyone can read data about any address, which can be a privacy concern for users of the system.
Recommendation: Add authentication requirement for all data endpoints of the system.
Severity: Low
Resolution: The team acknowledged the issue, and state that this is by design: most of this data is on-chain, and so is already readable by all, and there are good UX reasons for keeping these end-points publicly accessible.

M9. Wallet filtering by state does not cover full range of states [low] **[resolved]**

In multiple points, such as the `_get_non_deactivated_wallets` and the `check_current_tvl`, the code filters out deactivated wallets. However, the `DEACTIVATED` status is not the only status for a deactivated wallet, you may also want to filter out `DEACTIVATING`, `ACTIVATION_FAILED`, and `DEACTIVATION_FAILED`.
This also applies when filtering other states, it is important to always cover the full range of states to filter. Note that this could allow some manipulation of the TVL, if a user purposely makes the deposit transaction to fail on activation, the wallet will be in an `ACTIVATION_FAILED` state, but the deposit will still be counted.
Recommendation: Filter the full range of status options instead of just a single state.
Severity: Low
Resolution: This issue was resolved as recommended.

M10. get_wallet_stats does not count vaults with APR >= 30 [low] [in progress]

In the function `get_wallet_stats` calculates, among other things, the `total_weighted_deposits` - which is the total amount deposited including interest. However, if the APR of a wallet is 30 or larger, it will not be counted:

```
if wallet_apr < 30:
      total_generated += profit
      weighted_apr_sum += initial_deposit * wallet_apr
      total_weighted_deposits += initial_deposit
```

This will systematically underestimate the total weighted APR.
Recommendation: Remove the check that the APR must be less than 30 to be counted
Resolution: This issue was not resolved yet, but the team intends to include a fix in the next release

M11. Agent authentication may pass even without Thirdweb approval [low] **[resolved]**

In the `authenticate` function, the JWT web token is checked on the Thirdweb API, and if the API does not return an error, the authentication passes successfully. This is the relevant code:

```
        try:
            response.raise_for_status()
        except HTTPStatusError as e:
            logger.warning(f"Error authenticating with wallet: {e}")
            raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="Not authenticated") from e

        logger.info("Authenticated with wallet")
```

The intention here seems to be that if the Thirdweb API throws an error, then authentication fails. However, `raise_for_status` raises an error only on HTTP status codes that are error codes (i.e. those >= 400). It will not raise an error on redirect codes or, say, a 204 response - but such responses definitely do not indicate that the token is Ok.
Recommendation: Instead of using `raise_for_status`, just check directly that the status is as expected (i.e. it should be 200), or check the response content for explicit successful validation. You could also consider checking the JSON body to be sure..
Resolution: The team acknowledged the issue but chose to leave the code unchanged.

M12. Ambiguity in parsing of token header [low] **[resolved]**

The `verify_wallet` function contains the following code:

```
      if "bearer" in auth.lower():
            auth = auth.split(" ")[1]
```

The result is a certain flexibility in the header format which is probably not harmful, but also not necessary. There is also the (very improbable) case where the header consists of a web token that contains a string like "`bEaRER`" somewhere inside the JWT string, which will fail the verification even if it is a perfectly fine header.
Recommendation: As you exactly know that the authorization header starts with the string "Bearer ", you should just check for this and then strip the prefix, i.e. have code that is something like:

```
      prefix = "Bearer "
      If not auth.startswith(prefix): raise BadHeaderError()
      auth = auth[len(prefix):]
```

Resolution: The issue was resolved as recommended

M13. APR calculation in get_wallet_apr is wrong both in very short as in very long time frames [low] **[resolved]**

In `business_adapter.py`, the APR is calculated on the basis of the active days and the last recorded performance data:

```
days_active = (datetime.now(timezone.utc) - last_reactivation_date).days
[...]
apr = ((performance_data[0].value * 1e6 - initial_deposit) /
initial_deposit) * (365 / days_active) * 100
```

In very short timeframes, this calculation may underestimate the APR. For example, suppose you run the calculation 73 hours after `last_reactivation_date`, and the last update of the performance data (which happens daily) happened 23 hours ago, and resulted in a gain of 1%. This corresponds to an APR of 175%, but your calculation will return 121%.

In contrast, on long timeframes, the calculation becomes kind of meaningless, as you are basically averaging out the APR of "all time", and you compare it with the initial deposit. For example, you got 1% on the first day but 0% in the 100 days after that - the current calculation will return an APY of 3.65%, but in reality, the wallet has not seen any gains for 3 months and so is likely to remain in that state.
Recommendation: You can easily correct for the underestimation on the short timeframe by using the last time the performance data was updated (or update the performance data before running the query). Instead, with respect to the long-term skew: it depends on the kind of use case that was intended here and whether the current calculation is as expected, or if it should be corrected to approximate the “current APR”.
Resolution: The issue was resolved as recommended for a short timeframe, and for a long timeframe, the team has confirmed that is the intended behaviour for now.