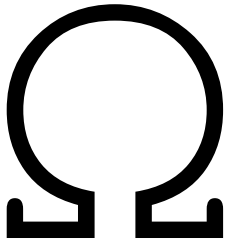

Team Omega

https://teamomega.eth.limo

Giza ARMA System Security Review

Final Report

December 9, 2024

Preliminaries

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 Review

Team Omega has done security analysis of the ARMA architecture and software as described in the following document:

https://docs.google.com/document/d/1lpQt39gCS-aSOnJoJJoKO_hq2f3wKkAM8f8YmnMaYBY

Specifically, Team Omega has performed a security review of:

1. The overall system setup as outlined in the above document
2. The frontend code, developed here: `https://github.com/gizatechxyz/pilot/tree/b4d9abda7cf83d10de9507a7d9246d92474cd9ab`
3. The backend code, developed here: `https://github.com/gizatechxyz/agent_mode/tree/1c2456002881a419cf7df21d0a4ed463f5547de1`
4. The agents API, developed here: `https://github.com/gizatechxyz/agents-api/tree/6afc9ab74b2d057042e027a6ac39250ed2882db5`

This review consists of two parts: the first part contains a high-level review of the various system components and their associated risks, and lists some general recommendations and best practices. The second part of the document lists a number of specific issues with the code under review.

Resolution

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

- `https://github.com/gizatechxyz/pilot/tree/2fe30221212f4f67a6fbbe0db46394135844ea37`
- `https://github.com/gizatechxyz/agent_mode/tree/ed0349b0d6e4c09835e48cc1e53c9f3b889fd360`
- `https://github.com/gizatechxyz/agents-api/tree/adf6c26ba28eef7836c1baa57a7d6968a41d870d`

Part of the code was refactored, and was moved to:

- `https://github.com/gizatechxyz/arma-contract-interactions/tree/e44dac31a9c7edf7c3af7577143f7094ec1bc0ee`

Apart from the fixes for the issues we raised in this report, there are also new, unrelated changes. We checked the changes in as far as they addressed the issues in this report, but did not audit the new code systematically.

History of this document

- We delivered a first version of this report on November 9, 2024
- Giza subsequently addressed a large part of the issues in either personal comments or fixes in the code
- We have updated the document with these fixes in the current, final, version of the report

Risks/Attack Vectors

The ARMA infrastructure depends on a number of components and services. Some of these are under direct control of Giza (the frontend, backend, agents API, the custom deployment of the Thirdweb AA infrastructure, and the configuration of services from Google, like access control and the firebase instance). Other components that could get compromised are not under control of GIZA, but do enlarge the attack surface of the system: the blockchain (Mode, Ethereum), the bridge (Across network), the Lending Protocols that ARMA interacts with (Ionic, Ironclad and Layerbank), the Thirdweb infrastructure as deployed by Giza, and finally the Google Cloud infrastructure, where all services run.

Below we list a number of general remarks on security best practices and procedures.

Components that are not under control of Giza

Component	**Type of risk**	**Risk Mitigation (what can you do before)**	**Postmortem (what can you do after)**
User (user wallet, browser, the person itself)	- User loses funds, unauthorized access to private keys	-	-
Across network (bridge)	- User loses funds on deposit or withdrawal - All (bridged) funds are lost	-	Pause/halt the system on failure. Implement monitoring of bridge functionality
Mode Network	- Funds are temporarily unavailable - Funds are irredeemably lost	-	Pause deposits immediately Withdraw all funds back to Ethereum when possible
Lending Protocols: - Ionic - Ironclad - LayerBank	- Protocol gets compromised and funds are frozen/lost - Providing wrong data (on APR) Note that if one protocol is compromised, the entire system is at risk	Do due diligence (see below)	Withdraw all funds from the affected protocol for all users Protocol should be disabled (no new deposits)
Swap router - KIM	- Swap gives bad prices - Swap is compromised	Add a third party oracle to check prices Simulate swap outcome before executing	Disable all swaps (e.g.. pause the agent)
Third party Data sources, such as Defillama (used for tvl of protocols) and Mode block explorer used in frontend only for view	These data are not used for any critical logic	-	-
Infrastructure (Google Cloud,	- One or more services are down	Have external (i.e. outside of the google cloud) backups of essential	

including GCP, Firebase, …)	- Data corruption or manipulation - Attacker gets full access	data that you need to restore access to the accounts	
DNS	- DNS server gets compromised - Domain is not renewed and taken over by hostile party	Use a reputable DNS provider Monitor expiry of DNS records	Pause the backend completely Inform all users immediately
Github	- A github accounts gets compromised.	Make sure all contributors have 2FA enabled Disable continuous deployment, or make at least 2 users sign off on deployments Require reviews before merging Limit contributors with write access as much as possible Limit contributors with admin access as much as possible	-
Various software dependencies	software dependencies that can introduce (new) vulnerabilities	Use tooling such as npm audit and pip-audit to identify known vulnerabilities Github has systems that alert you if a vulnerability is found in one of the dependencies	Disable any affected systems until the problem is fixed

Components that are under Control of Giza

Component	**Type of risk**	**Risk Mitigation (what can you do before)**	**Postmortem (what can you do after)**
Frontend	- user (gets tricked into) signing unintended transactions - Attacker steals user credentials - Frontend has a critical bug	- Prepare a way of disabling the frontend (e.g. redirecting users to a new landing page hosted on an independent system)	- Disable the frontend until the problem is solved.
ARMA Backend (auth and proxy for agent api, agent logic)	- Agent makes bad decisions - Unauthorized access - Access to sensitive data	- Monitor agent behavior and on-chain user positions closely,	- Pause or disable the entire system - Pause or disable the

	- Bug in Agent logic leads to loss of funds	especially in the initial stages - Add additional, separate, sanity checks - Store all passwords and keys encrypted	agent
Agent's API (proxy to thirdweb engine, data persistence, transaction validation) and ThirdWeb Engine	- Unauthorized access - Data gets lost - Data gets compromised - Transactions get blocked		- Pause or disable the system
Google Cloud Services and Configuration	Errors in GC configuration can lead to unauthorized access on all levels	- Have an explicit policy document mapping out who has access to what, and match that with implemented policies - Be prepared to withdraw credentials at any time	-

DeFi protocols

The security of the entire system depends on whether the protocols where the money is deposited are trustworthy. Here is some basic due diligence. Overall, our assessment is that Ironclad is the weakest link in the system.

Ionic

Ionic is a decentralized non-custodial money market protocol, supported by a comprehensive security monitoring and failsafe systems

- Website: https://www.ionic.money/
- TVL: 125.56M$ (https://defillama.com/protocol/ionic-protocol)
- Github: https://github.com/ionicprotocol/monorepo/tree/development/packages/contracts Last audit: by Zellic from May 19, 2022, https://reports.zellic.io/publications/ionic-protocol
- Last code changes: August 2024 - it is unclear which changes are deployed after the last audit

Overall security assessment: 👍 (high TVL, medium term history without incidents)

Ironclad

Ironclad Finance is a decentralized, user-driven borrowing and lending liquidity market

- Website: https://www.ironclad.finance/TVL: 1.56M$ (https://defillama.com/protocol/ironclad-finance?denomination=USD)
- Audits: https://docs.ironclad.finance/resources/audits Ironclad is an incarnation of a series of protocols forked from Granary. Their website lists a number of audits (the last one from february 2023 is from another fork, ethos.finance), while the last changes in the contracts are from June 2024. It does not explicitly link the current deployments to the audit reports.

Github: https://github.com/The-Granary/Ironclad_Finance/commits/main/contracts

Overall security assessment: 😐 (low TVL, unclear audit status)

Layerbank

Layerbank is a TVL aggregator

- Website https://layerbank.finance/
- TVL: 172M$ (https://defillama.com/protocol/layerbank#information)
- Audits: Last audit by Peckshield, from October 2023, https://github.com/peckshield/publications/blob/master/audit_reports/PeckShield-Audit-Report-LayerBank-v1.0.pdf
- Github: https://github.com/layerbank/contracts (last github changes: July 19, 2023) - The audit report seems up to date with the code

Overall security assessment: 👍 (high TVL, medium term history without incidents, recent audit)

Recommendations

We have given specific recommendations in the tables above, here we give a more general overview of policies and systems you may implement.

Roughly, recommendations can be distinguished in those having to do with *prevention, monitoring, and damage control.*

Prevention

- Limit access to vulnerable systems as much as possible and review your access policies regularly
- Enforce peer review on any code changes, audit all changes before new deployments
- Store passwords and private keys encrypted

Security Monitoring

If something happens could put funds at risk, it is best that you know as soon as possible. Here are some signals that you should be aware of:

- Monitor System Health (are the systems up and running, is the infrastructure compromised, is there anomalous activity)
- On-chain monitoring (e.g. is there activity in any of the on-chain wallets that is not in line with expectations, do the holdings of an agent or a wallet diminish unexpectedly, etc)
- User reports (e.g. twitter mentions, but users should also have a way of contacting you directly in case of problems or suspicions)

Ideally, monitoring happens on infrastructure that is independent of the running system.

Monitoring is only useful if signals are actually acted upon. This means that:

- Triage:: You should think of how to distinguish real signals from noise - if a lot of false flags are raised, it becomes tempting to ignore them.
- Ownership: Somebody needs to see the signals: make sure it is always clear who is responsible at any time
- Escalation: You should have an escalation policy: who should be contacted in the case of an emergency, what should be done by whom?
- You should have technical possibilities to limit the impact in case of problems, as described in the next section

Incident Response

If something bad or suspicious happens, you need some way to limit the (potential) damage. Here, the main goal is to safeguard the user investments and/or limit the loss as much as possible.

- Prepare a formal incident response plan
- Have a communication plan: What do you communicate to your users in case of a breach, and what channels do you use?
- Implement “emergency brakes” which halts all or parts of the system. This holds for specific sub-components (i.e. stop the agent from trading) or disabling the frontend, or have on-chain mechanisms that limit user actions and make these pausable. Make sure these “brakes” are accessible to the right people, so they can actually be used in case of an emergency.
- Implement methods to disable or limit on-chain activity - retract permissions or allowances, pause contracts where possible

- Implement a way of safeguarding user funds in case of an emergency (for example, wind down user positions and transfer their holdings to their EOA account or another holding account in case of emergency)

Resolution

Giza has acknowledged most of the observations above, and updated procedures and scripts where they thought this was useful and suitable.

Specific issues

General

G1. The agents decision logic will almost surely lead to loss of funds [high] [resolved]

The Giza agent decides, daily, whether to switch investment strategy, by looping over the protocols and finding the protocol/token combination that gives the highest APR. If the APR is higher than that of the current protocol,, the agent will withdraw the tokens from the current protocol, swap them if necessary for a new token, and deposit the token in the new protocol.

The decision to switch does not take into account the costs of changing protocols and tokens. These costs may include:

- Slippage and swap costs
- Transaction fees
- Protocol fees
- Losses (or gains!) from volatility/price discrepancies between stable coins (even if these are stablecoins, they is some tolerance to deviate from the peg)

Note that the gain from switching protocols can be very small: an increase of APR of 1% a year corresponds to only about 0,02% a week or 0,003% a day.

Due to the fact that the switching costs may easily outweigh the gain in return of switching, it may in some cases be desirable to remain with the current strategy, even if the APR is not the highest, or to choose a strategy which does not give the highest APR, but which uses the same token as the current one.

These decisions are hard to make, as the difference in daily profits between protocols will almost always be smaller than the costs for switching strategies, and so they must involve longer term planning and predictions about the differences in future APR.

Recommendation: Reconsider the agent logic to also count costs as part of the total expected APR in its decision making process

Severity: High

Resolution: The algorithm now takes into account the costs of swapping and gas, which addresses the main concerns.

G2. Consider implementing “emergency brakes” [info] [partially addressed]

As part of our recommendations for incident response, we recommend implementing “emergency brakes” in crucial parts of the system that allow you to disable or pause affected services, or even allow you to unwind positions in case of an emergency. The purpose here is to limit damage as much as possible - e.g.. limit the amount of money lost, or the number of users affected.
There are several places where this would be useful:

- The frontend. Here, the result would be that users get redirected to a new landing page that explains that the system is temporarily disabled
- Parts of the backend, especially those that run “autonomously”
- Retract permissions via the Thirdweb Engine
- Add an on-chain layer of protection

It is important that such emergency brakes are easily accessible, as emergencies can happen any time, also when . A permission model that distinguishes admins (which have wide-ranging power and so is limited to a very small set of people) and incidence responders (which can only halt systems but not do any specific damage, and so can be extended to a larger group of people) is helpful here.
Resolution: This issue gives a number of general recommendations, some of which do not require code changes. The team has acknowledged these recommendations.

Frontend

F1. Slippage of cross chain swaps is unnecessarily high [medium] [partially resolved]

For cross chain swaps, slippage is currently hardcoded at 2%, which seems quite arbitrary, and high as well. The system should instead use the Across API to query a more convenient price, as suggested here: https://docs.across.to/reference/selected-contract-functions
Recommendation: Fetch the slippage from the Across API, and only use the hardcoded option as a maximum restriction on the result of the calculation from the API (to ensure an upper limit on the slippage).

Severity: Medium
Resolution: .

F2. User can skip paying fees as it is taken on the front end side with no payment verification [medium] [not resolved]

The calculation of the amount to pay as fee is done on the backend, but the actual payment of the fee is done on the front end side, with no verification that the fee payment has been made. This means that users can skip or modify the payment of the fee.
Recommendation: Approve the fee receiver wallet in the wallet permissions and make the fee payment a part of the deactivation process on the backend server.
Severity: Medium
Resolution: The issue was acknowledged by the team - the behavior is intentional

F3. Logs should exclude any sensitive data [low] [resolved]

Logs meant for debugging should ideally be removed from the production app, but in any case they should not contain any sensitive data. See for example: https://github.com/gizatechxyz/pilot/blob/22ac07d321479dcfd91b9172c060884b59bc882a/lib/auth.ts#L27
Recommendation: Remove the logs or limit them to development only, and ensure sensitive data is not printed to the logs.
Severity: Low
Resolution: The issue was resolved as recommended.

F4. Timestamps which are expected to be identical may be slightly inconsistent [low] [resolved]

Every time the current timestamp is needed in the code, it is retrieved by calling JS's `new Date()`, which gets a date object with the current time in milliseconds. Since the code simply queries the current timestamp whenever it is needed, it can end up using slightly different timestamps in different parts of the execution of a single operation. This could lead to data saved in one place not accurately matching data saved in another place. For example, the `permissionStartTimestamp` here might be slightly different than the one used later in the execution here, when they should in fact be the same.
Recommendation: Whenever a timestamp is needed in various parts of the same operation, get the current timestamp first, then use and pass it as needed, instead of using the `new Date()` every time.
Severity: Low

Resolution: The issue was resolved as recommended.

F5. permissionEndTimestamp sent to the server may be wrong [info] [resolved]

The `permissionEndTimestamp` sent to the server is hardcoded at 1 year, while the `permissionEndTimestamp` registered in the contract is using the `expirationDate` set by the user. So in case the user sets a different expiration time the `permissionEndTimestamp` sent to the user will be wrong.
Recommendation: Set the `permissionEndTimestamp` to the user set `expirationDate` (if exists).
Severity: Info
Resolution: the issue was resolved as recommended

F6. Wallet labeling is not used properly [info] [not resolved]

All backend wallets created get the label “test”. It could probably use the user’s address for the label to have a unique label for each wallet, or alternatively it should pass no label if labels are deemed unnecessary. Ideally, the backend should set the label based on the user address, instead of relying on the front end to pass the label as part of the request.
Recommendation: Remove the label part from the front end and set a unique label when the wallet is created on the Agents API.
Severity: Info
Resolution: The issue was not resolved.

F7. Posthog debug mode should be disabled for production [info] [resolved]

The Posthog configurations set its debug mode to true regardless of the environment used, but it should not be used for production.
Recommendation: Make Posthog debug mode conditional on the environment the app runs in.
Severity: Info
Resolution: The issue was not resolved.

Backend

B1. Authentication flaw allows unrestricted execution of any command by any user [critical] [resolved]

The server uses the JWT token to verify whether a user is logged in or not when making a call. However, it does not verify the JWT belongs to the wallet passed to the endpoint to execute on, allowing any user to access any endpoint of any other user.
This means that any user which is logged in can: get all data of another user the system has, activate, run (see issue B4) and deactivate the agent of any other user, update a user's sessions permissions on the Agents API and so disrupt transactions from the backend by removing approved contracts and functions.
Recommendation: The authentication functionality (either on the backend or the `isLogged` on the Agents API) must verify the address the JWT belongs to matches the wallet to execute a command on, for example like shown here.
Severity: Critical
Resolution: The issue was resolved.

B2. Slippage protection relies on a quote which itself has no slippage protection [critical] [not resolved]

In `_approve_and_swap`, the slippage protection currently gets a quote for the swap, then takes 0.5% off of the quote as slippage. However, the quote received already includes a slippage with it, which is not checked at all. This renders the whole slippage protection essentially obsolete, and allows swaps with just about any degree of slippage, putting most of the user's deposit at risk.
Recommendation: Cap the slippage from the quote itself at the hardcoded limit, and don't allow any swaps below the hardcoded limit.
Severity: Critical
Resolution: The code now simulates a transaction to get a real value of the possible slippage.

B3. Any user can cause the new agent activation to halt for everyone by manipulating the TVL [high] [not resolved]

Due to issues B6 and B7, it is possible for any user to manipulate their deposit data by passing a fake deposit amount to the activation call. And since the TVL calculation depends on the amount saved in

user deposits, any user could pass a deposit amount which is higher than the TVL limit, and cause the server to believe the TVL cap has been reached and so to reject activation of new agents.
Recommendation: Fix the issues mentioned, and ensure TVL reflects the on-chain state of the system.
Severity: High
Resolution: the issue was not resolved, as B6 and B7 were not resolved

B4. Authentication allows a user to access a task meant for the Job Key and vice versa [medium] [not resolved]

The authentication function checks if either a valid Job Key was passed in the request header, or a valid JWT of a user, and if so allows access to the endpoint. This means that there is no distinction between endpoints meant to be accessed by the Job Key and endpoints which are meant for the user. This is overly permissive and risky, especially since it increases the risk to the system in case the Job Key is ever compromised.
Recommendation: Separate the authentication methods for endpoints meant to be accessed by the Job Key and those meant to be accessed by users.
Severity: Medium
Resolution: the issue was not resolved

B5. Slippage protection is hardcoded at 0.5% [medium] [not resolved]

The slippage protection for swaps between USDC and USDT is hardcoded at 0.5%, which seems arbitrary, and higher than necessary. The code should get expected slippage based on a quote, and only cap the slippage at 0.5%, instead of taking a constant 0.5% off the quote queried.
Recommendation: Use the slippage from the quote received, and only cap the total slippage at 0.5% instead of setting it at 0.5% below the quote.
Severity: Medium
Resolution:

B6. Deposit amount passed on activate is not verified [medium] [not resolved]

When calling `activate`, the user sends the amount they deposit as part of the request. However, this amount is not verified at any point, making it unsafe to rely on in future calculations, such as of the TVL (see issue B7)
Recommendation: When activating, the wallet manager logic doesot rely on the deposit amount passed, but instead it checks the balance of the user and uses it for its logic. Instead of asking the user to pass their deposit amount in the request, it will be safer and more accurate to use the balance as the deposit amount the same way it is used in the wallet manager.
Severity: Medium

Resolution: The developers acknowledged the issue but decided to not resolve it

B7. TVL calculation is flawed [medium] [partially resolved]

The system calculates its TVL by summing up all user deposits as saved in the deposits array of each wallet. However this calculation is inaccurate. First, due to issue B6, which makes the deposit amount saved in the deposits array unreliable. Second, the user may withdraw their deposits (with or without the agent's deactivation), which again would lead to the deposits array being a flawed source of information. Third, the system's TVL increases as the user generates interest which is also a part of the value locked in the system, but this will not be reflected by the deposits array. Due to these reasons, the current TVL calculation is flawed and unreliable, and needs to be refactored to rely on on-chain data.
Recommendation: The TVL should be relying on on-chain data of users deposits in the different protocols for better accuracy and reliability, this data could be cached and updated periodically to avoid performance issues.
Severity: Medium
Resolution: The calculation was updated and now excludes deactivated wallets.

B8. Checking TVL cap on agent activation does not account for the new deposit [low] [acknowledged]

When activating a new agent, the TVL capped is being checked and the activation is canceled if the TVL was reached. Yet the new deposit that is about to be added is not accounted for, making it possible for the TVL to cross its intended limit by an unknown amount (the whole user deposit).
Recommendation: Add the new user deposit to the current TVL and check if the cap was reached for the combined amount.
Severity: Low
Resolution: The developers have acknowledged the issue and state that this is compatible with the intended behavior.

B9. User deposit data is not being reset on agent deactivation [low] [resolved]

When the user deactivates their agent, their deposits and remaining_deposit data is not being reset. This leads to reactivation of the agent having older data in these variables, which affects the fee calculation of reactivated agents. It also affects the TVL calculation, as withdrawn funds are still being counted in the TVL. And it may affect other parts of the server in future development. Additionally, a user's new deposit is not saved on reactivation of an agent.

Recommendation: When deactivating the agent, calculate and save the fee which will be required to pay, then delete the user's deposits and remaining_deposit data. You could delete saved fee data once it's paid.
Severity: Low
Resolution: The issue was resolved, the reactivation now overrides previous deposits.

B10. Whitelist based deposit limit is not enforced on the backend [low] [resolved]

The deposit limit of a user which is based on a whitelist stored in the server is verified on the front end, but not on the backend, making it trivial for a technical user to circumvent that limit.
Recommendation: Move verification of deposit limit to the backend agent activation process.
Severity: Low
Resolution: The issue was resolved as recommended.

B11. Unnecessary approval before withdrawing from lending protocols [low] [resolved]

Before making a withdrawal, there is a check of whether an approval of the token withdrawn exists for the lending protocol, and if not an approval transaction is being executed. However, there is no need to approve any funds before making a withdrawal, since the user receives tokens and does not send them.
Recommendation: Remove the unnecessary approval logic from the withdrawal process.
Severity: Low
Resolution: The issue was resolved as recommended.

B12. Extra rewards are not being transferred to the user on agent deactivation [low] [resolved]

The deactivation code claims the extra rewards from the lending pools (which offer such rewards), but it does not transfer them to the user in any way, leaving them in the smart wallet. It also does not account for them in the fee calculation, nor in the APR of the lending protocol.
Recommendation: Add the process needed to withdraw such extra rewards from the smart wallet on deactivation, and (if desired) also include them in the fee calculation.
Severity: Low
Resolution: The issue was resolved; the withdrawal happens in the frontend

B13. Missing check if extra rewards exist before claiming [info] [not resolved]

There is no check that extra rewards from a lending pool exist before calling to claim them. It would be better to check and avoid making the transaction if there are no rewards.
Recommendation: Check if any extra rewards are available to claim before making a transaction to claim them.
Severity: Info
Resolution: The issue was not addressed

B14. remaining_deposit does not take into account whitelisted addresses [info] [resolved]

The `remaining_deposit` calculated in `get_wallet_information` assumes the limit for all addresses is the same, and does not account for the whitelisted addresses having no limit. However, the variable is only used for logs and could also just be removed.
Recommendation: Remove `remaining_deposit` or ensure it is ignored for whitelisted addresses.
Severity: Info
Resolution: The issue was resolved as recommended by removing the variable.

Agents API

A1. Access restriction to API endpoints is inconsistent [low] [resolved]

In `thirdweb-sessions.controller` some functions are guarded using `@UseGuards(AdminGuard)`, which checks that the API key passed is known.

However, not all functions are protected by authentication; not only the GETters have no access control (these functions only read data, and so access is probably harmless), but also the , `createWallet` is not guarded by any authentication, and there's no check the caller is authenticated with same user ID that has been passed.

In general, all calls on the agents API can be restricted to the backend server, since the user is not meant to ever call it directly. This will allow effective access management in a single place (i.e. either in the backend server, or in the Google Cloud configuration).

We also note that the Agents-api should *already* be isolated to be callable from the backend only, as per the technical document, so it is not completely clear to us whether this extra authentication layer is really needed:

> Agents-API: A REST API responsible for interactions with the ThirdWeb Engine. This service is fully isolated and only accessible within the Google Cloud Platform (GCP) infrastructure, restricting exposure to external threats. Authorization is managed through Google Metadata Server's IAM-based authentication, granting access solely to Arma-Backend with predefined permissions.

Recommendation: Either apply the authentication checks in the code consistently and apply them to all endpoints of the Agents-API, or remove them altogether and implement access restrictions on the server level.
Severity: Low
Resolution: The issue was resolved: all sensitive endpoints now have a `@UseGuards(AdminGuard)` decoration.