Autonolas
Launch Assessment

Report Date: June 6th, 2022

Prepared for:

David Minarsch

CEO, Valory AG

Prepared by:

Blake Johnson

ApeWorX Ltd.

SCOPE

1. Autonolas *onchain-protocol* repository (supplied via GitHub @ 77f8ca2)
2. Autonolas *token* repository (since combined into *autonolas-governance*)

GOALS

1. Review codebase and identify pedigree of code
2. Review codebase organization and documentation
3. Summarize suggestions for improvements

GENERAL SUGGESTIONS

These are high-level processes that the ApeWorX team recommends to follow for all projects.

S1. Check for known security issues while developing smart contracts.

Review and understand latest hacks and exploits, both of similar protocols, as well as any dependencies you rely on

S2. Consider special features of your contracts.

a. Are the contracts upgradeable?
b. Do your contracts conform to ERCs?
c. Do you have unit tests?

S3. Visually inspect critical security features of your code often.

S4. Document critical security properties and use automated test generators to evaluate them.

a. Learn to document security properties for your code.
b. Define security properties in Solidity. Focus on your state machine, access controls, arithmetic operations, external interactions and conformance to standards.
c. Define security properties. Focus on inheritance, variable dependencies, access controls and other structural issues.

S5. Frequently review your Development Process.

This is one of the most crucial factors in having a secure codebase.

S6. Have an audit readiness checklist.

Since audits are expensive and time consuming, completing a checklist helps ensure your codebase is read for outside review. It also helps catch as much low-hanging fruit as possible. It removes this task from the auditors' responsibilities and allows them to focus their attention on identifying deeper vulnerabilities.

a. Documentation - The less time spent trying to understand your system, the faster a deep dive into your code can be accomplished. And the more time an audit team can spend finding bugs. Find documentation standards, and implement them throughout your code base.
b. Clean code - Well-formatted code is easier to read. It reduces the overhead needed to review. Run linters on your code regularly. Fix any errors or warnings unless there is a good reason for not doing so. (*for Solidity, Ethlint is a good option*). If the compiler outputs any warnings, address them. Remove any comments that indicate unfinished work (TODOs: if possible). Remove commented code, and any code that you do not need.
c. Testing - Do everything in your power to have 100% code coverage if possible. Review the list of test cases for gaps. Look into fuzz testing. You should have very clear instructions in a markdown file for running the test suite. List all required dependencies.
d. Automated Analysis - There are many CLI tools for this to be performed in JavaScript, Python. This is not an essential step, but it certainly helps.
e. Frozen code - You should not be in development of code that is being audited. The code development should be frozen at the time of the audit. Audit teams will ask for a specific git commit hash as the target. If you do make a change to the code during an audit, it will have wide-ranging consequences and impacts on things like the threat model and other code that interacts with the changes. If your team is not ready for an audit, push the audit back.
f. Use a checklist - Create an audit prep checklist. List out all of your steps that you need to hit before your audit begins.

S7. Have a pre-launch security checklist available.

Before deploying, your checklist should have taken the necessary precautions to enable reporting and responding to bugs and security issues.

S8. Create an incident response plan.

Having a plan documented in advance can help your team respond to potential security incidents swiftly and calmly.

S9. Be aware of your token's limitations and the security implications of your token.

S10. Upgradability can be problematic.

Immutable code is a core feature of smart contracts to achieve trustlessness. Upgradeability removes this feature and makes it harder for downstream users leveraging your system to be able to reliably trust the interactions they have with your system when upgrades are made.

SUMMARY

This is a report completed by the ApeWorX team. The goal of this assessment is to review the codebase structure of the Valory Protocol and associated smart contract repositories, clear up issues with the organization of code, documentation and processes, and suggest potential improvements of all facets of the codebase and development processes in preparation for launch.

The ApeWorX team developed a detailed list of suggestions provided later on in this assessment. To summarize, we recommend a refactoring of both supplied repositories to support deployment and audit goals. We highly suggest that any dependencies being used throughout the codebase be referenced, rather than copied from the original sources, to improve source code management.

It was also found that a release schedule was specified to be released in phases, but the codebase was in a mono-repository format making it difficult to audit each individual stage. The ApeWorX team suggests that Valory split the repository into separate stages, and reference earlier phases in each successive stage. This allows for each component to be audited before the launch of the next.

Lastly, it is highly recommended to move all pdf formatted documentation to markdown within each stage's repository, or build a separate documentation portal with appropriate links. The current documentation is confusing to navigate, and pdf files are difficult to reference as a specification.

On the positive side, with the three progressive launches of the project, Valory significantly reduced risk inherent in the release of their protocol. The tiered release allows for potential unknown risks in the later stages to be mitigated prior to launch while getting live exposure. With the initial launch using high pedigree code for critical components, many of the potential risks associated with this release have been reduced. Considering the code being leveraged is currently developed by leaders in the space, and perpetually reviewed, security risks are mitigated exceptionally by the Valory team.

Furthermore, the use of such high pedigree code mitigates risk such that the need for an audit is pushed off to a later stage when the Valory team is ready to launch their second phase. Considering the second phase is heavily infused with custom built software, an audit pre-release of the second phase would be a stern recommendation from the ApeWorX team.

CODE PEDIGREE ASSESSMENT

Group	Contract File	Dependencies	Pedigree Assessment
Governance	*ERC20VotesNonTransferable.sol*	*openzeppelin/contracts/governance/utils/IVotes.sol* *openzeppelin/contracts/token/ERC20/IERC20.sol*	*IVotes.sol* last updated in v4.5.0 *IERC20.sol* last updated in v4.6.0
Governance	*GovernorBravoOLA.sol*	*openzeppelin/contracts/governance/Governor.sol* *openzeppelin/contracts/governance/extensions/GovernorSettings.sol* *openzeppelin/contracts/governance/compatibility/GovernorCompatibilityBravo.sol* *openzeppelin/contracts/governance/extensions/GovernorVotes.sol* *openzeppelin/contracts/governance/extensions/GovernorVotesQuorumFraction.sol* *openzeppelin/contracts/governance/extensions/GovernorTimelockControl.sol*	*Governor.sol* last updated in v4.6.0 *GovernorSettings.sol* last updated in v4.4.1 *GovernorCompatibilityBravo.sol* last updated in v4.6.0 *GovernorVotes.sol* last updated in v4.6.0 *GovernorVotesQuorumFraction.sol* last updated in v4.5.0 *GovernorTimelockControl.sol* last updated in v4.6.0
Governance	*Timelock.sol*	*openzeppelin/contracts/governance/TimelockController.sol*	*TimelockController.sol* last updated in v4.6.0
Governance	*VotingEscrow.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/governance/utils/IVotes.sol* *openzeppelin/contracts/token/ERC20/IERC20.sol* *openzeppelin/contracts/token/ERC20/ERC20.sol* *openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol*	*Ownable.sol* last updated in v4.4.1 *IVotes.sol* last updated in v4.5.0 *IERC20.sol* last updated in v4.6.0 *ERC20.sol* last updated in v4.6.0 *SafeERC20.sol* last updated in v4.4.1 *ReentrancyGuard.sol* last updated in v4.4.1

		openzeppelin/contracts/security/ReentrancyGuard.sol	
Multisig	*GnosisSafeMultisig.sol*	*gnosis.pm/safe-contracts/contracts/proxies/GnosisSafeProxy.sol* *gnosis.pm/safe-contracts/contracts/proxies/GnosisSafeProxyFactory.sol*	*GnosisSafeProxy.sol* last updated in v1.3.0 *GnosisSafeProxyFactory.sol* last updated in v1.3.0 Could not determine pedigree of *GnosisSafeMultisig.sol*
Registries	*AgentRegistry.sol*	*openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol* *openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/ReentrancyGuard.sol*	*ERC721Enumerable.sol* last updated in v4.5.0 *Ownable.sol* last updated in v4.4.1 *ReentrancyGuard.sol* last updated in v4.4.1
Registries	*ComponentRegistry.sol*	*openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol* *openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/ReentrancyGuard.sol*	*ERC721Enumerable.sol* last updated in v4.5.0 *Ownable.sol* last updated in v4.4.1 *ReentrancyGuard.sol* last updated in v4.4.1
Registries	*RegistriesManager.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/Pausable.sol*	*Ownable.sol* last updated in v4.4.1 *Pausable.sol* last updated in v4.4.1
Registries	*ServiceManager.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/Pausable.sol*	*Ownable.sol* last updated in v4.4.1 *Pausable.sol* last updated in v4.4.1
Registries	*ServiceRegistry.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/ReentrancyGuard.sol*	*Ownable.sol* last updated in v4.4.1 *ReentrancyGuard.sol* last updated in v4.4.1
Tokenomics	*Depository.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/token/E*	*Ownable.sol* last updated in v4.4.1 *IERC20.sol* last updated in v4.6.0

		RC20/IERC20.sol *openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol*	*SafeERC20.sol* last updated v4.4.1
Tokenomics	*Dispenser.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/security/Pausable.sol* *openzeppelin/contracts/security/ReentrancyGuard.sol* *openzeppelin/contracts/token/ERC20/IERC20.sol* *openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol*	*Ownable.sol* last updated v4.4.1 *Pausable.sol* last updated v4.4.1 *ReentrancyGuard.sol* last updated v4.4.1 *IERC20.sol* last updated v4.6.0 SafeERC20.sol last updated v4.4.1
Tokenomics	*Tokenomics.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol* *openzeppelin/contracts/token/ERC721/extensions/IERC721Enumerable.sol* *uniswap/lib/contracts/libraries/FixedPoint.sol* *uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol*	*Ownable.sol* last updated v4.4.1 *SafeERC20.sol* last updated v4.4.1 *IERC721Enumerable.sol* last updated v4.5.0 *FixedPoint.sol* has been deprecated *IUniswapV2Pair.sol* last updated in v1.0.0
Tokenomics	*Treasury.sol*	*openzeppelin/contracts/access/Ownable.sol* *openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol* *openzeppelin/contracts/security/ReentrancyGuard.sol*	*Ownable.sol* last updated v4.4.1 *SafeERC20.sol* last updated v4.4.1 *ReentrancyGuard.sol* last updated v4.4.1

RECOMMENDATIONS

R1. Refactor OLA Token repository

It is recommended that the Token repository be refactored to build the OLA token contract referencing Solmate as a dependency rather than building the OLA token on top of the solmate repository. This is not best practice, and updates to Solmate will require a rebase every time Solmate makes changes in order to keep up to date, rather than just updating versions referenced. This will be confusing for auditors and you are also pulling in a lot of unused code which makes the audit more difficult to assess.

R2. Reference OLA Token implementation directly

During the review, we determined that the *onchain-protocol* repository was creating a separate implementation of the OLA token for testing purposes. To ensure accurate testing, the token contract should be referenced as a dependency from the token repository rather than a mock copy being created for testing purposes.

R3. Split *onchain-protocol* repository into stages

There are three planned stages of the release of the protocol by the Valory team. Each stage should be self-contained in a separate repository. Following the launch, there should be very few changes made to ensure that the code best mirrors what is deployed on-chain. The next stage to be released should reference previous stages and their explicit dependencies.

Typically, when there is a completion of work at one stage, the next stage will still be in development. If the code is kept all together in a single repository, it makes it very difficult to audit any changes. From our review, it appears the three suggested repositories should be *autonolas-governance*, *autonolas-protocol*, then *autonolas-tokenomics*.

It is also recommended by the ApeWorX team to get every stage audited eventually. Each stage should be audited before their respective launch date, or at worst very shortly thereafter.

As changes are made moving forward, you will want to have a history of the differences from the audit for small bug enhancements and fixes. After an audit, every re-deployment of a component from a stage should be tagged and tracked to make it easy to identify changes.

R4. Ensure each deployment can be performed separately

As stated previously, it is not ideal for every stage to be referenced in one script for the deployment. This launch will be done in stages. Each stage's repository should have a way to deploy every contract inside it individually. If there is a fault in one of the contracts in a stage which has been fixed and ready to deploy, this allows you to deploy that component separately. This also allows you to release your stages separately and make necessary changes before deployment. It is bad practice to have all of the deployments in one script, because in an emergency scenario where you just need to deploy one update, trying to determine what portion of a script needs to be triggered takes precious time.

R5. Understand Solmate pedigree

Make sure the differences of the latest solmate from the last audited solmate version are well understood. Solmate v6.4.0 is currently referenced, the last audit was v6. There have been more than 140 commits since Solmate's last audit.

R6. Improve *VotingEscrow.sol* pedigree

From what we can tell, *VotingEscrow.sol* was copied from the Solidly project. The Solidly implementation of *ve.sol* was a reimplementation of the *VotingEscrow.vy* contract from Curve. Valory appears to have modified the Solidly implementation using components from OpenZeppellin. It is recommended that this is split up into a series of commits so it easier to identify it's pedigree. To start, take what was copied, and make that the base commit. Then, make changes that include substitutions of OpenZeppelin, and make that their own commits. Use the commit history to document changes that were made from the original implementation. There should be a clear record of what was done from the start to the latest changes, as this will improve the auditability of your code.

R7. Improve Gnosis Safe contract pedigree

Currently, a part of the multisig implementation was copied over to the Valory Protocol repository from Gnosis Safe. It is recommended that multisig is referenced as a dependency rather than copying it over. Gnosis Safe maintains this repository already so it should be possible to obtain the exact code necessary for testing and deployment of the governance system.

R8. Tag releases prior to audit

There should be a process of tagging set up in order to track deployments. This will make a future audit much more simple to complete, and will reduce confusion. It is recommended to

use major versions to track audited releases, and to apply a *-preaudit.X* tag suffix to denote audits in progress and track each successive set of audits for the release that way as well.

R9. Explore exploit mitigation strategy

We suggest that Valory takes the time to have a full understanding of how difficult it would be to replace different parts of your architecture. If there is a bug following deployment, the last thing you want to run into is the need for a major refactor of the core architecture. For example, the voting escrow contract locks the OLA tokens. If there was a fault in that contract, it would be a serious fault, and would not be possible to recover from that fault without considering a redeploy of the entire project. This consideration should lead to reducing risk for contracts like the voting escrow to prevent unrecoverable scenarios. The code pedigree of this contract should therefore be high (meaning a reduction in the amount of changes), or a lot of due diligence should be applied to this contract in the form of audits and testing prior to launch.

R10. Review OpenZeppelin release notes

The assumptions behind the OpenZeppelin contracts should be understood. The last time OpenZeppelin was audited was in 2018 on v2.0.0, which was more than 1400 commits ago. Make sure every single change since the last major audited version is reviewed for security patches, and each change is well-understood for how to avoid pitfalls from using it incorrectly.

R11. Add security contact details

It is recommended best practice to define a *SECURITY.md* file in smart contract repositories, or a separate *security* repository. This should contain links to audits performed on the codebase, as well as a *security@valory.xyz* email address (with corresponding PGP key) to contact in case of emergency. Lastly, this should contain your bounty program details, such as links and scoping of any bounties you have for your codebase(s).