Contracts vulnerabilities

Vulnerabilities list:

Involved contracts and level of the bugs

The present document aims to point out some vulnerabilities in the autonolas-registry contracts.

Vulnerabilities

1. `tokenURI` function

Severity: Low

The following function is implemented in the GenericRegistry contract:

```
function tokenURI(uint256 unitId) public view virtual override
returns (string memory)
```

This function is defined by the EIP-721 standard. The standard states that the function is supposed to throw if ***unitId*** is not a valid NFT. However, in our contract, the function does not revert if the ***unitId*** is out of bounds, but just returns the value of a string with the defined prefix and 64 zeros derived from a zero bytes32 value.

Therefore, we recommend checking the return value of this view function, and if the last 64 symbols are zero, consider it to be an invalid NFT. Also one might use the ***exists()*** function to preliminary check if the requested NFT Id exists.

2. `create` function

Severity: Low

The following function is implemented in the GnosisSafeMultisig contract:

```
function create(address[] memory owners, uint256 threshold, bytes
memory data) external returns (address multisig)
```

This function creates a Safe service multisig when the service is deployed. Since Autonolas protocol follows an optimistic design, none of the fields for the Safe multisig creation are restricted. This way, the service owner might pass the *payload* field as they feel fit for the purposes of the service multisig. That said, any possible malicious behavior can also be embedded in the *payload* value.

In the event of the intended malicious multisig creation, the Autonolas protocol is not affected, however, accounts interacting with the corresponding service might bear eventual consequences of such a setup.

We strongly recommend not abusing the *payload* field of the service multisig when deploying the service to perform any malicious actions.

3. `update` function (zero bonds)

Severity: Low

The following function is implemented in the ServiceRegistry and ServiceRegistryL2 contracts:

```
function update(address serviceOwner, bytes32 configHash, uint32[]
memory agentIds, uint32 threshold, uint256  serviceId) external
returns (bool success)
```

This function allows updating a service in a *pre-registration* state in a CRUD way. E.g. if there is a need to remove `agentIds[i]` from the canonical agents making up the service, then it is sufficient to call this function and update it in such a way that a corresponding slots field is set to zero, i.e., `agentParam[i].slots=0,` also adjusting the `threshold`.

When an agent slot is non-zero, and an operator can register an agent instance for that slot, it is necessary that the corresponding agent bond is non-zero. In the current implementation, there is no check for agent bonds to be different from zero if the corresponding agent slot is non-zero. This vulnerability would enable an operator to register an agent instance without the corresponding security bond. Hence, the operator would not be affected by any possible slashing condition if the total operator bond is equal to zero.

This vulnerability is addressed for the ServiceRegistry contract and ServiceRegistryL2 by adding the zero-value check on the service manager level. Specifically, serviceManager contract handles the check before calling the original serviceRegistry's update() method. See https://github.com/valory-xyz/autonolas-registries/blob/main/test/ServiceManagerToken.js#L326-L333C25 for a test proving that the issue is resolved.

In absence of redeploying a new manager for the ServiceRegistryL2 contract on other chains, we recommend that service owners assign a zero-value to agent bonds only if the corresponding agent slot is zero.

4. `update` function (replacing agent Ids)

Severity: Low

The following function is implemented in the ServiceRegistry and ServiceRegistryL2 contract:

```
function update(address serviceOwner, bytes32 configHash, uint32[]
memory agentIds, uint32 threshold, uint256  serviceId) external
returns (bool success)
```

As described earlier, this function allows updating a service in a *pre-registration* state in a CRUD way. However, considering that there is no possible direct damage to the protocol and to save on transaction gas costs, the function is implemented via an optimistic approach.

Specifically, the service owner might not specify that some of the *agent Ids* of the previous setup must be taken out of the system (by setting corresponding *slots* variable to zero). This means that operators are able to register agent instances specifying non-declared service agent Ids (as those were deliberately left in the corresponding map from the previous setup). This might lead to deploying the service on *agent Ids* from the previous setup, declaring that they actually run on current ones (as retrieved via the *getService()* view function).

We strongly recommend not abusing the *update()* function in order to deploy the service to perform any malicious actions by using undeclared *agent Ids*, since this behavior is easily spotted off-chain.

5. `drain` function

Severity: Informative

The following function is implemented in the ServiceRegistryTokenUtility contract:

```
function drain(address token) external returns (uint256 amount)
```

The primary purpose of this function is to allow the removal of slashed tokens, other than chain-native tokens, from the contract.

By design, in the current setup of the Treasury contract, there is currently no mechanism in place to facilitate the removal of tokens other than ETH that have not been added to the Treasury through the treasury `depositTokenForOLAS()` method. Therefore, we strongly advise against assigning the drainer role to the Treasury contract for ServiceRegistryTokenUtility contract deployed on Ethereum.

6. `_checkTokenStakingDeposit` function

Severity: Informative

The following function is implemented in the ServiceRegistryTokenUtility contract:

```
function _checkTokenStakingDeposit(uint256 serviceId, uint256
stakingDeposit, uint32[] memory) internal view virtual
```

The primary purpose of this function is to ensure that the service owner's security deposit and the operator bonds are correctly configured. Specifically, it checks that the service owner's security deposit ($securityDeposit$) and the $bond$ for each operator are greater than or equal to $minStakingDeposit$. Given that $securityDeposit$ is defined as the maximum among the operator bonds ($max_{bond}\{bond\}$), when $minStakingDeposit$ equals $securityDeposit$, the following relationship holds:

$$minStakingDeposit = securityDeposit >= bond >= minStakingDeposit$$

This ensures that $securityDeposit = minStakingDeposit = bond$ for each operator bond. It's important to note that the service registry and service registry utility tokens do not enforce this requirement at the service level.

If one attempts to stake a service with a $securityDeposit$ equal to $minStakingDeposit$ and operator bonds that differ (e.g., $bond[i] > bond[i]$), it is recommended to terminate and update the service configuration to ensure compatibility with the staking logic.

7. `_isRatio` function

Severity: Informative

The following function is implemented in the StakingActivityChecker contract:

```
function isRatioPass(uint256[] memory curNonces, uint256[] memory
lastNonces, uint256 ts)
```

This function checks if the service multisig liveness ratio meets the defined threshold. The provided implementation serves as an illustrative example, and we highlight that multisig nonces are not tamper-resistant (cf. InternalAudit4 for more details on this). It is therefore recommended to extend the basic **isRatioPass()** functionality in the StakingActivityChecker to verify whether specific on-chain actions occur within designated time frames. For a tamper-resistant check on on-chain activity, you can consider the one implemented in MechActivityChecker.sol in this repository.

Additionally, the protocol optimistically assumes that the StakingActivityChecker contract used for deploying staking instances is implemented with a correct logic. Therefore, unless unexpected behavior such as reverts or non-boolean returns occur, the contract's results will be considered accurate. However, this optimistic assumption can be exploited by malicious users. For instance, malicious users could deploy multiple contracts with flawed activity checks that always return true. They could then vote for these contracts,

causing the OLAS amount to be distributed to all stakers, including those without activity. Conversely, malicious users could deploy contracts with incorrect liveness checks that always return false, leading to a situation where the OLAS amount is sent, but funds remain stuck in the staking contracts and cannot be recovered.
The following measures can be considered to mitigate eventual abuses:

1. Set a Sensible Threshold: The DAO needs to establish a sensible threshold to enable staking emissions.
2. On-Chain Blacklist: Implement an on-chain blacklist that can be updated through governance votes, allowing the community to monitor and exclude malicious contracts.
3. Off-Chain Reputation System: Consider using an off-chain reputation system, possibly leveraging oracles, to assess the trustworthiness of contracts.

8. `stake` function

Severity: Informative

The following function is implemented in the StakingFactory contract:

```
function stake(uint256 serviceId) external
```

The function stakes a specified service. However, if the service was evicted, it cannot be staked again until it is explicitly unstaked.

9. `unstake` function

Severity: Informative

The following function is implemented in the StakingBase contract:

```
function unstake(uint256 serviceId) external returns (uint256 reward)
```

The function unstakes a previously staked service. If there are no available rewards left on a staking contract, the service can be unstaked immediately at any time. However, if there are even small funds deposited on the staking contract, the service will not be unstaked. It is not considered to be a griefing attack, since the unstake time is pre-defined via a `minStakingDuration` parameter. After that time, the service is unstaked without any concern of zero or non-zero available rewards. When the service is staked, it implicitly agrees to be staked for at least the `minStakingDuration`.

10. `checkpoint` function

Severity: High

The following function is implemented in the StakingBase contract:

```
function checkpoint() external returns (uint256[] memory, uint256[]
memory, uint256[] memory, uint256[] memory)
```

The function goes through all currently staked services and checks for their activity KPIs. As designed, the function is `O(n)` in its complexity, where `n` is the number of staked services. The actual staking limit is 500 services per staking contract. With the recent Fusaka EVM upgrade being adopted, the hard cap on transaction gas limit is imposed. Governance action to limit staking number of slots takes roughly a week. During this time, if a staking contract is misconfigured and has more than 100 slots, there is a risk that having more number of services staked results in a failure of a checkpoint transaction. This scenario ultimately leads to all services being staked indefinitely. In time of waiting for the proposal properly limiting the number of staking contract services, launchers of staking contracts are urged to configure them with no more than 100 of staking slots.

11. `deploy` function

Severity: Informational

The following function is implemented in the ServiceRegistry and ServiceRegistyL2 contracts:

```
function deploy(address serviceOwner, uint256 serviceId, address
multisigImplementation, bytes memory data) external returns (address
multisig)
```

This function is responsible for deploying a service by creating a multisig instance controlled by the set of service agent instances. When the deployment uses a `RecoveryModule`-enabled multisig implementation, an additional module is installed to allow the service multisig to recover ownership in the event that agent instance keys are accidentally lost.

While this recovery mechanism allows re-obtaining control over the service multisig at the protocol level, it does not guarantee recoverability for all downstream integrations. In particular, certain interactions may still rely on the original multisig owner keys rather than the recovered ownership state.

A notable example is the PolySafe (Gnosis Safe L2) integration used by Polymarket. In this context, if the original multisig owner key is lost, there is no supported mechanism to regain effective control over the multisig from Polymarket's perspective, even if ownership is recovered on-chain via the recovery module. As a consequence, critical operations such as:

- buying or selling conditional tokens
- future earning Polymarket liquidity rewards

may become permanently inaccessible.

Therefore, although the recovery module provides resilience against agent key loss at the service level, it does not fully mitigate the risk of loss of functionality for external systems that bind permissions to the original multisig owner keys.