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. If the payload field affects a service in any way, an eventual service ranking implemented in tokenomics, is going to signal about the intended service misbehavior.

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 by adding the zero-value check on the service manager level. Specifically, serviceManagerToken serving as a new service manager 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#L297-L303 for a test proving that the issue is resolved.

In absence of redeploying a new manager for the ServiceRegistryL2 contract, 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 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*. This behavior is easily spotted off-chain, and an eventual service ranking is going to signal about the intended service misbehavior for the rest of the service livespan.

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 tokens, other than ETH, from the contract.

By design, the current setup of the Treasury contract, there is currently no mechanism in place to facilitate the removal of the chain native tokens. Therefore, we strongly advise against assigning ServiceRegistryTokenUtility drainer role to the Treasury contract.

Since on goerli testnet the drainer is currently the Treasury contract, we strongly advise against initiating a vote to call the drain method until a governance vote is conducted to update the drainer address.

6. `GnosisSafeSameAddressMultisig` `create` function

Severity: Informative

The following function is implemented in the GnosisSafeSameAddressMultisig contract:

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

This function is used when a service is being redeployed with an existing multisig address.
The primary objective of this function is to retrieve multisig-related data from the provided `data`. Subsequently, the function checks whether the `owners` and `threshold` values provided as arguments match those associated with the provided multisig-related data. This verification process is essential because it ensures that the multisig configuration remains consistent.

It's important to note that this function does not validate whether the provided address is genuinely a Gnosis Safe multisig contract. This opens the possibility for a service owner to create a potentially malicious multisig contract. They can achieve this by implementing the `getOwner` and `getThreshold` methods and assigning them to the values of their rogue multisig, thereby bypassing proper Gnosis Safe multisig validation.

In conclusion, this function should be used with care, and in absence of redeployment of a new contract, it is strongly recommended to refrain from exploiting the service redeployment mechanics to create potentially insecure multisig contracts.