Introduction

Consider a DAO is going to own a deployed service. Let's assume that the DAO governance executing contract is the *timelock*, and this is the contract address that would become a service owner.

Make sure that the timelock contract has the *onERC721Received()* function implemented. This way, the *safeTransferFrom()* function can be used to transfer the service. The old service owner can either *approve()* the *timelock* contract for the service Id being transferred, or they are able to just transfer the service by calling the *safeTransferFrom()* function on their address: *oldServiceOwner* to call *serviceRegistry.safeTransferFrom(oldServiceOwner, timelock, serviceId)*.

The means of transferring the service are beyond the scope of this document. Let us consider that a specific service with a unique service Id has been acquired by the *timelock* contract, and now the DAO is in charge of the service.

Pre-requisites

Service state and proposal sequence.
The service is currently deployed and the DAO has to vote on a proposal that does the following sequence: terminate the service, unbond operators, update the service, activate service registration, register agent instances, re-deploy the service with the same multisig it currently has. Once the proposal is executed, the service is deployed and runs again with DAO specified agent instances registered by provided operators. Note that before the proposal is submitted with the steps described in the next section, one additional action is required: current agent instances (multisig owners) have to transfer the ownership to the *timelock* address.

Example for compute security deposit and bonds.
Service registration security deposit and the sum of overall bonds computation is described in the following example. Consider that the service has the following agent parameters:

- *agentIds=[2, 3, 4]*,
- *agentParams=[[2, 100], [4, 500], [3, 200]]*,

meaning that *agentId* 2 is going to have 2 agent instances with a bond of 100 (wei), *agentId* 3 is going to have 4 agent instances with the bond of 500, and *agentId* 4 is going to have 3 agent instances with the bond of 200. The registration security deposit is defined as a maximum possible bond from all the provided bonds. Thus, for the described setup the registration security deposit is equal to:

- *securityDeposit = max(agentParams[:][2])* = 500.

The sum of all the bonds is equal to:

- *totalBond = sum(agentParams[i][0] * agentParams[i][1], i)* = 2800.

Note that there could be from one to several operators registering agent instances, and thus the maximum number of operator-registered agent instances is equal to the overall number of agent instances in the service:

- *maxNumAgentInstances* = *sum(agentParams[i][:])* = 9.

Independently from the number of operators and corresponding number of agent instances each of them register, the sum of all operator bonds must be equal to the *totalBond*.

One-shot proposal for the DAO-owned service redeployment

Initial proposal arrays:

```
targets[]
values[]
payloads[]
```

Note that there is a possibility to secure the service with a custom ERC20 token. If that is the case, then step 0 must be completed, otherwise the DAO can proceed to step 1. In this example we assume that step 0 is going to be executed by default.

0. Approve for the custom ERC20 token.

If the service is going to be secured with the custom ERC20 token, then the *timelock* address must approve that token for the *ServiceRegistryTokenUtility* contract address and the minimum amount of a sum of service-computed *securityDeposit* plus *totalBond*:

- *approveAmount* = *securityDeposit* + *totalBond*.

Proposal arrays after step 0:

```
targets[ServiceRegistryTokenUtilityAddress]
values[0]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount])]
```

1. Terminate the service.

The service needs to be terminated before it can be updated.

Proposal arrays after step 1:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress]
values[0, 0]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId])]
```

2. Unbond agent instances.

The complete agent instance unbonding is necessary for the service to be able to update. In order to unbond all the agent instances from all the operators currently bonding, the DAO has to acquire each operator signature corresponding to the hash of the unbond message.

The unbond message hash is composed with the following function:
getUnbondHash(operatorAddress, serviceOwnerAddress, serviceId, nonce),
where *operatorAddress* is the address of each operator, *serviceOwnerAddress* is the current service owner (that in this example coincides with the *timelock* address), *serviceId* is the Id of the service, and *nonce* is the value corresponding to the unique (*operatorAddress* | *serviceId*) pair. Each pair's *nonce* is increased by 1 each time the signature is utilized by the service owner during the successful unbond transaction.

In order to get the up-to-date *nonce* specifically for the unbond message, the following function is conveniently provided:
getOperatorUnbondNonce(operatorAddress, serviceId),
that corresponds to the (*operatorAddress* | *serviceId*) pair. Once the signed unbond transaction is executed, the *nonce* value for the pair is incremented, and the same *nonce* used for the current message hash cannot be further utilized. In other words, the *nonce* guarantees that the signed unbond transaction can be executed only once.

After the unbond message hash is constructed by a specific operator, it has to be signed by the operator themselves, or pre-approved by the operator, or added to the verified set of hashes if the operator is a contract (see *_verifySignedHash()* function for more detail). The result of that action is the signature bytes of the operator corresponding to the unbond message hash.

The amount of unbond-related payloads is equal to the number of unbonding operators.

Proposal arrays after step 2:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress]
values[0, 0, 0]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId]),
serviceManager.interface.encodeFunctionData("unbondWithSignature",
[operator.address, serviceId, unbondSignatureBytes])]
```

3. Update the service.

Optionally, the service configuration is updated (refer to the *update()* function of the *ServiceRegistry* contract).

Proposal arrays after step 3:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress]
values[0, 0, 0, 0]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId]),
serviceManager.interface.encodeFunctionData("unbondWithSignature",
[operator.address, serviceId, unbondSignatureBytes]),
serviceManager.interface.encodeFunctionData("update", [token.address,
newConfigHash, newAgentIds, newAgentParams, newMaxThreshold, serviceId])]
```

4. Activate registration.

Before operators are able to register agent instances, the service owner activates the registration. In this part of the proposal, the value field is not going to be zero. If the service is ETH-secured, then the value is equal to the *securityDeposit* (in wei). Otherwise, in case of the custom ERC20 token-secured service, the value is equal to 1 (wei). Here we continue with the custom ERC20 token path. Thus, the value for this payload is equal to 1 (wei), and the actual *securityDeposit* in custom ERC20 tokens will be transferred from the service owner during the activation registration transaction.

Proposal arrays after step 4:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress]
values[0, 0, 0, 0, securityDeposit(1)]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId]),
serviceManager.interface.encodeFunctionData("unbondWithSignature",
[operator.address, serviceId, unbondSignatureBytes]),
serviceManager.interface.encodeFunctionData("update", [token.address,
newConfigHash, newAgentIds, newAgentParams, newMaxThreshold, serviceId]),
serviceManager.interface.encodeFunctionData("activateRegistration",
[serviceId])]
```

5. Register agent instances.

During this step, operators register agent instances corresponding to service's agent Ids. As in the case of unbond, register agents' action depends on operators.

The register agents message hash is composed via the following function: *getRegisterAgentsHash(operatorAddress, serviceOwnerAddress, serviceId, agentInstances, agentIds, nonce)*,

where *operatorAddress* is the address of each operator, *serviceOwnerAddress* is the current service owner (that in this example coincides with the *timelock* address), *serviceId* is the Id of the service, *agentIstances* are registered agent instance addresses, *agentIds* are agent Ids corresponding to *agentIstances*, and *nonce* is the value corresponding to the unique (*operatorAddress* | *serviceId*) pair. Same as for the unbond, each pair's *nonce* is increased by 1 each time the signature is utilized by the service owner during the successful register agents' transaction.

In order to get the up-to-date *nonce* specifically for the register agents' message, the following function is conveniently provided:
getOperatorRegisterAgentsNonce(operatorAddress, serviceId),
that corresponds to the (*operatorAddress* | *serviceId*) pair. Once the signed register agents' transaction is executed, the *nonce* value for the pair is incremented, and the same *nonce* used for the current message hash cannot be further utilized.

After the register agents' message hash is constructed by a specific operator, it has to be signed by the operator themselves, or pre-approved by the operator, or added to the verified set of hashes if the operator is a contract (see *_verifySignedHash()* function for more detail). The result of that action is the signature bytes of the operator corresponding to the register agents' message hash.

Note that all the registering operator signatures must be collected at this stage. The sum of all the bonds across all operators must correspond to the *totalBond* value calculated for the service setup.

Also note that if the same operator is going to register agents two times in a row or more (very unlikely, but could still be the case), each consequent nonce value for the **same** operator address must be manually incremented by 1 based on the initial nonce value for the first register agents' message via the *getOperatorRegisterAgentsNonce()* function.

The value field of this proposal part is not going to be zero. If the service is ETH-secured, then the value is equal to the calculated operator's bond (in wei), or the *totalBond* if a single operator registers all the agent instances. Otherwise, in case of a custom ERC20 token-secured service, the value is equal to the number of registered agent instances (in wei). Here we continue with the custom ERC20 token path and assume that the new service setup is going to have 4 agent instances registered by a single operator. Thus, the value for this payload is equal to 4 (wei), and the actual *totalBond* in custom ERC20 tokens will be transferred from the service owner during the register agents' transaction.

The amount of register agents-related payloads is equal to the number of registering operators.

Proposal arrays after step 5:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress]
values[0, 0, 0, 0, securityDeposit(1), totalBond(4)]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId]),
serviceManager.interface.encodeFunctionData("unbondWithSignature",
[operator.address, serviceId, unbondSignatureBytes]),
serviceManager.interface.encodeFunctionData("update", [token.address,
newConfigHash, newAgentIds, newAgentParams, newMaxThreshold, serviceId]),
serviceManager.interface.encodeFunctionData("activateRegistration",
[serviceId]),
serviceManager.interface.encodeFunctionData("registerAgentsWithSignature",
[operator.address, serviceId, agentInstancesAddresses, agentIds,
registerSignatureBytes])]
```

6. Redeploy the service.

Assuming all the required operators provided their signatures to register all the service's agent instance slots, the last payload deploys the service. If the multisig of the service needs to be preserved from the previous setup, the data that swaps the current multisig owner (*timelock* address) has to be constructed beforehand as a *packedData* bytes array. This step is out of the scope of this document. If the service multisig is going to be overwritten, then the *packedData* can be left as zero bytes.

Ultimately, the final proposal dataset is as follows.

Proposal arrays after step 5:

```
targets[ServiceRegistryTokenUtilityAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress,
ServiceManagerTokenAddress, ServiceManagerTokenAddress]
values[0, 0, 0, 0, securityDeposit(1), totalBond(4), 0]
payloads[token.interface.encodeFunctionData("approve",
[serviceRegistryTokenUtility.address, approveAmount]),
serviceManager.interface.encodeFunctionData("terminate", [serviceId]),
serviceManager.interface.encodeFunctionData("unbondWithSignature",
[operator.address, serviceId, unbondSignatureBytes]),
serviceManager.interface.encodeFunctionData("update", [token.address,
newConfigHash, newAgentIds, newAgentParams, newMaxThreshold, serviceId]),
serviceManager.interface.encodeFunctionData("activateRegistration",
[serviceId]),
serviceManager.interface.encodeFunctionData("registerAgentsWithSignature",
[operator.address, serviceId, agentInstancesAddresses, agentIds,
registerSignatureBytes]), serviceManager.interface.encodeFunctionData("deploy",
[serviceId, multisigImplementationAddress, packedData])]
```

7. Vote and execute.

Once the proposal data is carefully constructed, it can be voted on and executed.

Mock of the one-shot proposal code

The simulation of the scenario described in this document can be found under the following link: https://github.com/valory-xyz/autonolas-registries/blob/5b84e372fea0b87b34b240f4da8641efb4301674/test/ServiceManagementWithOperatorSignatures.js#L569-L783