Secure your Autonomous service with token and whitelist operators

Background

Service Registry and Service Manager contracts of Autonolas v1 provide registration, management, and securing of autonomous services.

Currently, service owners have limitations on how they activate service registrations, and there is no way to allow a specific set of operator addresses to opt in to operate their services. In order to secure an autonomous service:

- the service owner provides a security deposit in ETH that will be returned to the service owner if the service is terminated;
- service operators deposit a security bond in ETH which can be slashed in case of misbehavior.

There are several motivations that require us to improve such mechanisms.

First, it is likely that a service owner requires explicit whitelisting of operators that can run their services. This opens up paths to introduce a more effective mechanism to incentivize good behavior based on a more sophisticated reputation of a system that takes into account the behavior of the service operator over time. In such a way, there are more benefits toward good behavior and it can help build trust among the service owners and operators. Operator whitelisting would also limit agent instance registration frontrunning issues by unwelcomed operators.

Second, allowing service owners to choose the token with which they aim to secure their services is crucial for the following reasons.

1. Provide utility to a third-party (ERC-20) token. That is relevant for regulatory requirements on a project's token.
2. DAO with an ERC20 token has the incentive to have their off-chain processes in the form of service and spend 'less' by securing their services by using their own native token, it can also help increase the liquidity and adoption of their native token, which, in turn, can lead to more use cases and higher demand for the token. This can ultimately benefit the overall ecosystem of the project.
3. The current mechanism does not provide any flexibility for service owners and operators to negotiate the terms of their agreement. Introducing the ability to use a third-party token can also help facilitate better agreements between the service owners and operators and ultimately lead to more successful and sustainable services.

Overall, improving the mechanisms for registering, managing, and securing autonomous services can help create a more efficient and effective ecosystem for autonomous services, which can ultimately benefit all stakeholders involved.

Proposals summary

Solution proposed for the first problem - whitelisting of operators

In order to let service owners to whitelist a set of operators with specific characteristic to opt into running their services, the simplest path would be

1. Deploy an OperatorWhitelist address that allows:
    a. service Owners to operate a whitelist of operator addresses for the corresponding serviceID that can opt-in to select only specific addresses to run their services;
    b. check if an address is whitelisted to run agent instance(s) for a serviceID, returning true or false corresponding to operator whitelist (de-whitelist), forming a set of operator addresses for a serviceID that can opt-in to select only specific addresses to register agent instances
2. Modify the Service Manager in such a way that:
    a. The *registerAgent()* method checks against the OperatorWhitelist if the service owner has set a check for Operators.
        i. If not, the original registerAgent service method allows operators to register agent instances.
        ii. If yes, each operator address that registers agent instances is checked to be whitelisted for the correspondent service ID. If whitelisted, the original *regisiterAgents()* method is called, otherwise the call is reverted.

Service owner workflow. If a non-zero OperatorWhitelist address is set by the service manager contract owner, service owners can proceed as follows.

1. If the service owner wants a permissioned set of operator addresses, they have to whitelist a set of operators addressed by calling the *setOperatorsStatuses* function and specify the operator check to be true or false, or set the Operator check equal to true or false by calling *setOperatorCheck(serviceId, true).*
    a. Note that, once the check is set to *true*, it is the responsibility of the service owner to make sure that some operator addresses are whitelisted. Otherwise no one, except for themselves, can register agent instances.
    b. Note that, once a set of operator addresses is whitelisted, it is the responsibility of the service owner to make sure that the check on the operators is set to true. Otherwise, operators are able to register agent instances in a permissionless manner.

2. The service owner can de-whitelist operator addresses by setting the operator statuses to *false* when calling the *setOperatorsStatuses()* method. Note that, if the service owner de-whitelists all the previously whitelisted operators but the *setOperatorCheck()* for a specific service ID is still set to true, no one, except for themself can register agent instances.
3. The service owner can set the *OperatorCheck()* to *false*, even if there are still whitelisted operators. In this case, the registration of agent instances of a service will be once again operated in a permissionless fashion.

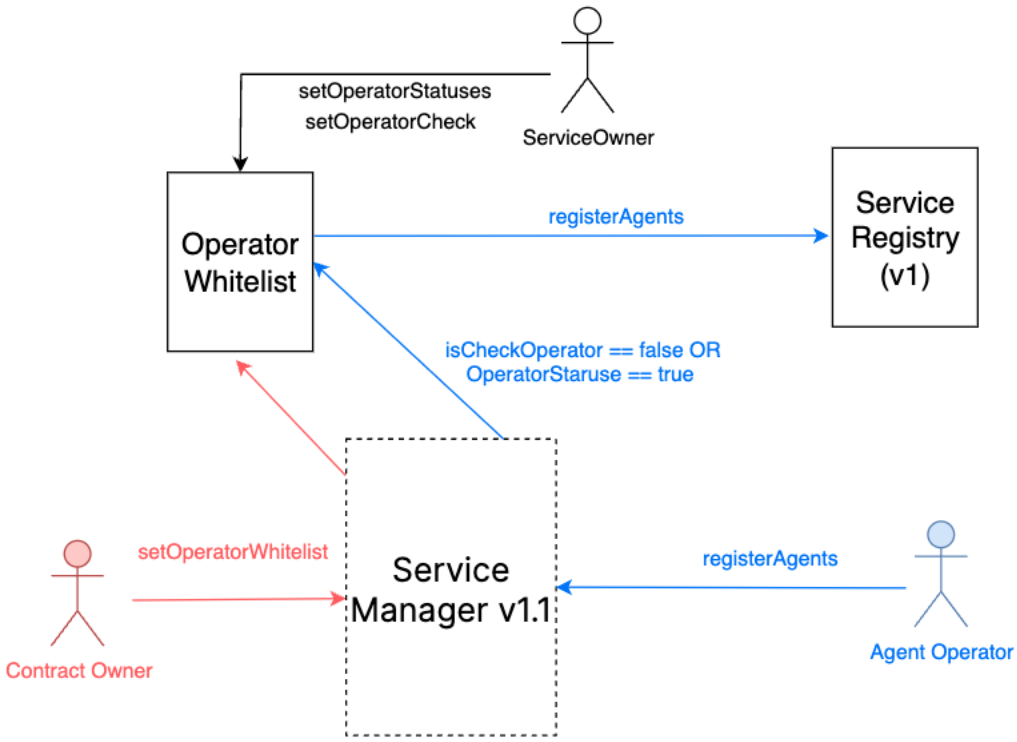

Fig. 1 Once the service registry has an operatorWhitelist, either service owners set a check for permissioned operations of their services and only whitelisted operators are able to register agent instances, or in case of permissionless operations, any operators are able to register agent instances for the service.

Technical specs

OperatorWhitelist contract

Data structures needed.

- *mapping (uint256 ⇒ bool) mapServiceIdOperatorsCheck;* this maps the *serviceId* into the status of the whitelistedOperator setting. (*True* if there is a whitelist set by the service owner, *false* otherwise)
- *mapping (uint256 ⇒ mapping (address ⇒ bool)) mapServiceIdOperators;* this maps the *serviceId* and the *operatorAddresses* into the operator whitelisted status. (*True* if for *serviceId* the operator address is whitelisted, *false* otherwise)

Methods.

1. *Change ownership and manager related functionalities.*
2. *setOperatorsCheck(uint256 serviceId,* bool *setCheck*): *true* if the service requires an operator whitelist, *false* otherwise. Only the service owner can call this method. This method can only be called for existing services.
3. *setOperatorsStatuses(uint256 serviceId,* address[] memory *operators,* bool[] memory *status, bool setCheck):* Allows the service owner to control the operators' whitelist status and to require or not the check of the operators whitelist. Specifically, when *operators.length = status.length and*, and the *operator's* array is not-empty, the whitelisted state in *status* for each operator in *operators* is set. Moreover, the corresponding relevant operator whitelist check requirement is set.
4. *isOperatorWhitelisted(uint256 serviceId, address operatorAddress)*: Returns *true* if the operator address is whitelisted for the service ID.

Remark. This is a much cheaper implementation compared to storing the cyclic map for each service Id or having a counter that forces the check of each address reading from the storage. However, the service owner has to correctly follow the workflow. Specifically, it is a simplest optimistic approach that accounts for the service owner's sanity check. In particular:

- If the service owner sets the check *mapServiceIdOperators* to true but does not whitelist any address, no operator -except for the service owner- is able to register an agent instance;
- it is possible that the service owner can set the check *mapServiceIdOperators* to false and whitelisted addresses in order to have a permission set of operators. In this case, the service can be operated in a permissionless manner.

ServiceManager v1.1 contract

Data structures to add with respect to the previous version of the contract.

1. *state public variable operatorWhitelist* for the operator whitelist contract address

Methods.
All the previous methods, except for the *registerAgents* method, will be the same. The method that allows the contract owner to update the OperaterWhitlist contract is required. Finally, the *registerAgents* will be modified as follows.

1. *registerAgents(uint256 serviceId, address[] memory agentInstances, uint32[] memory agentIds):*
    a. *Check* if the operatorWhitelist's contract address was settled.
        i. If yes, it also checks whether the operator address is whitelisted for the serviceID.
            1. If the operator is whitelisted, e.g. the operatorWhitelist contract method *OperatorWhitelisted(uint256 serviceId, address operatorAddress)* returns *true*, the corresponding method on the core service contracts is called.
            2. Otherwise, the call is reverted.
        ii. *Otherwise,* the corresponding method on the core service contracts is called.

Solution proposed for the second problem - secure autonomous services with token

To let service owners using a token different from ETH to secure their services, the simplest path would be

1. Deploy an additional core contract ServiceRegistryTokenUtility which allows wrapping the implementation of the original Service Registry contract in order to manage the extra storage variable containing all the custom ERC20 token information. This will moreover contain the slash and drain functionalities.
2. Replace ServiceManager contract with the combination of calls to ServiceRegistry when the service is secured with ETH and ServiceRegistryTokenUtility when the service is secured with the custom ERC20 token.

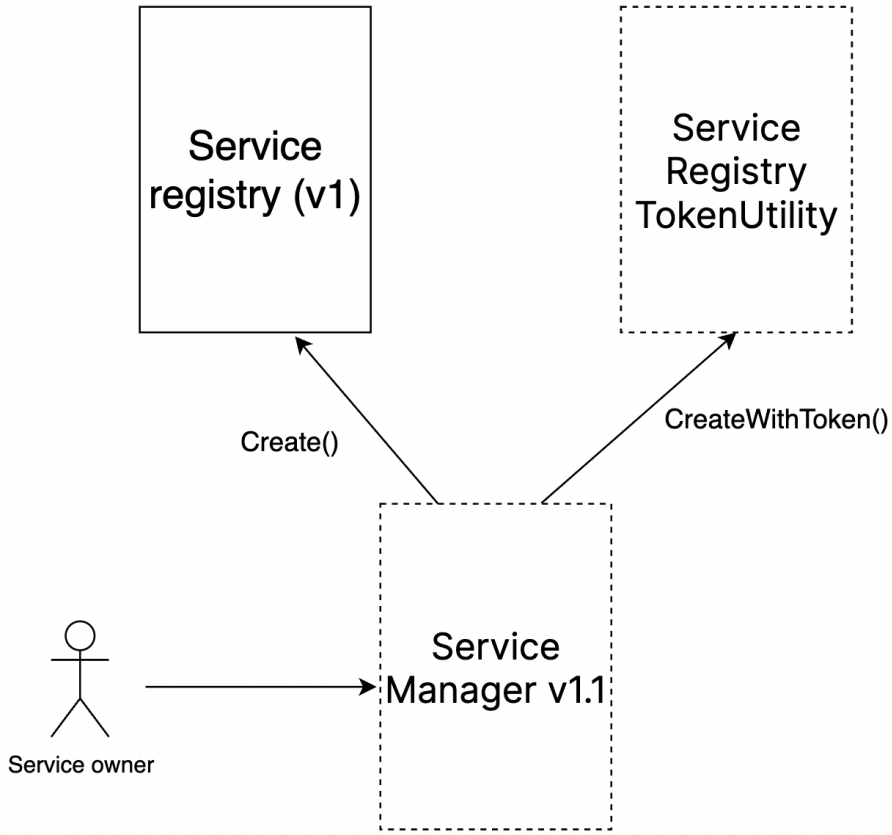

Fig. 2 Secure autonomous services with any token.

FSM

Service states and transition maps do not change from the actual v1 https://github.com/valory-xyz/autonolas-registries/blob/main/docs/FSM.md.

When a service is secured with ETH, only the serviceRegistry contract is involved. While, if a service is secured with a token, serviceRegistry, and serviceRegistryTokenUtility are involved in the call. All the methods but *deploy()* and *update()* which allow a *state* transition and require token-related information are on the serviceRegistryTokenUtility as well, which only handles the new token-related information and the checks against those. E.g. transition from active registration and finished-registration happens in the following way:

1. Agents instance gets registered and provides a token bond to the serviceRegistryTokenUtility contract.

2. If 1 is successful, the agent instance registration and one wei-wrapped bond are made to the serviceRegistry contract.

Steps 1 and 2 repeat, until in step2, it is checked that the total number of agent instances is reached. In this case, the finished-registration state is reached.

Technical specs

ServiceManagerToken

All the methods in ServiceManager.sol except for *deploy(), slash() and drain()* will be implemented in such a way when the service is secured with Token (isTokenEnabled)

- all the methods that require checking on token-related struct are called and handled on the ServiceRegistryUtilityToken contract implemented in the optimistic way;
- the service workflow and the original checks are properly handled on the ServiceRegistry contract, where the security deposit and bond are set equal to one wei.

If the service is secured with ETH, the original service registry functions will be called.

Finally, the update() function is implemented in the following way:

- If the service is updated such that it becomes ETH-bonded, it is added a zero-check on the bond value, the correspondent data is removed from the ServiceRegistryTokenUtility, and original ServiceRegistry *update()* function is called;
- If the service becomes token-secured, there is a zero-check on the bond value, then the token related data is communicated to the ServiceRegistryTokenUtility contract via the *createWithToken()* function, as it is the same for both create and update paths. The original ServiceRegistry *update()* function is also called to check the required data correctness.

Note that all the token data is processed in the ServiceRegistryTokenUtility contract in an optimistic way, considering that the correspondent ServiceRegistry data validation is handled correctly and is always called together with the ServiceRegistryTokenUtility counterpart.

Here is a breakdown of what the
update(address token, bytes32 configHash, uint32[] memory agentIds, IService.AgentParams[] memory agentParams, uint32 threshold, uint256 serviceId)
function does and how will be implemented:

- It first checks if the token address is not zero, and if it is, it throws an error.
- If the token is the default ETH token, it is checked that the agent slot is non-zero and the agent bond is non-zero, and it calls the *update()* function of the original ServiceRegistry contract, passing in the necessary parameters. It then resets the

service token-based data by calling the *resetServiceToken()* function implementation of the ServiceTokenUtility contract.

- If the token is not the default ETH token, it first creates an array of bonds, one for each agent, based on the actual bond values for agents that have at least one slot in the updated service. It then wraps the bonds with the *BOND_WRAPPER* value for the original ServiceRegistry contract, and calls the *update()* function of the original ServiceRegistry contract, passing in the necessary parameters. Finally, it updates the relevant data in the ServiceTokenUtility contract by calling the *createWithToken()* function, passing in the service ID, the token address, the agent IDs, and the bonds.

ServiceRegistryUtilityToken

Data structure needed.

1. struct *TokenSecurityDeposit {address token; uint256 securityDeposit;}*: structure for the address of the *token* used to secure a service registration.
2. Variables
    a. *address public immutable serviceRegistry:* address of the serviceRegistry contract
    b. *address public owner:* address of the contract owner
    c. *address public manager:* address of the manager contract
    d. *address public drainer:* address of the contract to which the slashed funds are sent

3. Maps:
    a. *mapping(uint256 ⇒TokenSecurityDeposit) public mapServiceIdTokenDeposit:* maps the serviceID into the TokenSecurityDeposit struct
    b. *mapping(address ⇒uint256) public mapServiceAndAgentIdAgentBond:* maps the service Id and the canonical agent id into the registration bond
    c. *mapping(address ⇒uint256) public mapOperatorAndServiceIdOperatorBalances:* maps the operator address and service Id into the instance bond
    d. *mapping(address ⇒uint256) public mapShashedFunds:* maps the token address into the token slashed amount.

Methods.

1. *Change ownership and manager related functionalities.*
2. *safeTransferFrom(address token, address form, address to, uint256 amount):* The implementation is taken from the audited MIT-licensed solmate code repository. While the original library imports the ERC20 abstract token contract and thus embeds all that contract-related code, in this version, ERC20 is swapped with the address representation. Also, the final require statement is modified with this contract's own revet statement.
3. *safeTransferFrom(address token, address to, uint256 amount):* The implementation is taken from the audited MIT-licensed solmate code repository. While the original library imports the ERC20 abstract token contract and thus embeds all that contract-related code, in this version, ERC20 is swapped with the address representation. Also, the final require statement is modified with this contract's own revet statement.
4. *createWithToken(uint256 serviceId, address token, uint256 memory agentIds, uint256 memory bonds):* Only manager can call this method which wraps the serviceRegistry *create()* function and is implemented as follows.
    a. It is checked that the *token* implements the *balanceOf()* method and throws an error otherwise.
    b. It is checked that the values of *bonds* do not overflow.
    c. the *securityDeposit* will be calculated as the maximum of the agent *bonds*.
    d. The *serviceAgent* key obtained using *serviceId and agentsIds[i]* is calculated and the *bonds[i]* related information is pushed in the map *mapServiceAndAgentIdAgentBond[serviceAgent].*
    e. the *SecurityDeposit* is calculated and then it is pushed together with the *token* address information in the map *mapServiceIdTokenDeposit[serviceId].*
    f. Only the manager contract is able to trigger this call.
2. *ResetServiceToken(uint 256 serviceId)*: Only the manager can call this method which is called to clean the token related information if the *serviceId* will be updated and then secured with ETH
3. *ActiveRegistrationTokenDeposit(serviceId)*: Only the manager can call this method.
    A. If token address in the *mapServiceIdTokenDeposit(serviceID)* is not zero:
        a. It is checked that the service owner of serviceID has approved this contract as spender for at least the tokenSecurityDeposit
            - If yes, the *isTokenSecured* flag is set to *true*, it checks the balances of the contract before and after the safetransferFrom(token,serviceOwner,address(this), TokenDecurityDeposit) to transfer the *SecurityDeposit*.

- If the balance before is larger the balance after or if the balance after is not the sum of the balance before and the security deposit, it is reverted
- Otherwise the event *TokenDeposit(serviceOwner, token, SecurityDeposit)* is emitted and it is set.

- Otherwise it is reverted.

B. If token address in the mapServiceIdTokenDeposit(serviceID) is zero:
    a. it is set isTokenSecured equal to false
C. This implementation is protected against reentrancy attacks by preventing the function from being executed again until the current execution has been completed.

4. *RegistrerAgents(address operator, uint256 serviceId,uint32[] memory agentIds*): Only the manager can call this method.
    A. If token address in the mapServiceIdTokenDeposit(serviceID) is not zero:
        a. the function then checks that the totalBond fee for all the agents is sufficient by summing up the bond fee for each agent
        b. It then checks the operator's allowance to the contract in the specified token.
            i. If yes, it checks the balances of the contract before and after the safeTransferFrom(token,operator,address(this), totalBond) to transfer the operator balance.
                1. If the balance before is larger the balance after or if the balance after is not the sum of the balance before and the security deposit, it is reverted
                2. Otherwise the event *TokenDeposit(operator, token, totalBond)* is emitted and the *isTokenSecured* flag is set to true.
            ii. Otherwise, it is reverted with an error.
    B. If the token address in the *mapServiceIdTokenDeposit(serviceID*) is zero:
        a. it is set *isTokenSecured* equal to *false.*
    C. This implementation is protected against reentrancy attacks by preventing the function from being executed again until the current execution has been completed.
5. *terminateTokenRefund(uint256 serviceId):* Only the manager can call this method.
    A. The function executes actions only if token address in the *mapServiceIdTokenDeposit(serviceID)* is not zero:
        a. It is called s*afeTransfer(token,serviceOwner,securityDeposit)* to transfer the *SecurityDeposi*t. The transfer is not checked for correctness since it relies on token implementation. The event *TokenRefund(serviceOwner,token,securityDeposit*) is emitted.

B. This implementation is protected against reentrancy attacks by preventing the function from being executed again until the current execution has been completed.

6. *unbondTokenRefund(address operator, uint256 serviceId)*: Only the manager can call this method.
    a. The function executes actions only if token address in the *mapServiceIdTokenDeposit(serviceID)* is not zero:
        i. It retrieved the amount of bond to refund to the operator. If not zero, it is called *safeTransfer(operator,refund)* to transfer the bond fee. The transfer is not checked for correctness since it relies fully on the token implementation.The event *TokenRefund(operator,token,refund)* is emitted.
    b. This implementation is protected against reentrancy attacks by preventing the function from being executed again until the current execution has been completed.

7. *slash(uint32[] memory agentInstances, uint32[] memory amounts, uint256 serviceId)*: Only the service multisig can call this method.
    1. The function checks that *agentInstances* and *amounts* have the same length and that the service is in deployed state on the original serviceRegistry contract.
    2. The function executes actions only if the token address in the *mapServiceIdTokenDeposit(serviceID)* is not zero and reverts otherwise.
        1. If the checks pass, there is a loop on the agent instances.
        2. For each agent instance, get the address of the operator who registered the agent instance for the service
        3. Calculate the key operatorService using the operator and serviceId variables. This key is used to keep track of the total balance of tokens for each operator who has registered agents for the service
        4. Check if the amount[i] to be slashed is greater than or equal to the current balance of the operator for the service. If it is, add the current balance to the total slashedFunds and set the balance to 0. Otherwise, subtract the amounts[i] from the balance and add the amounts[i] to the slashedFunds.
        5. Update the balance of the operator for the service with the new balance
        6. Emit an event to record the number of token slashed for the operator
        7. After all agent instances have been processed, add the slashedFunds to the total slashed funds for the token being slashed and update the mapSlashedFunds mapping for the token

        8. Return success as true to indicate that the function executed successfully.
2. *drain(address token):* Only the contract owner can call this method.
    1. If the drainer address is non-zero and the amount to be drained is bigger than zero, the function updates the mapSlashedFunds[token] to zero and it is called safeTransfer(token,drainer,amount) to transfer the slashed amount. The event TokenDrain(msg.sender,token,amount) is emitted.
    2. This implementation should protect against reentrancy attacks by preventing the function from being executed again until the current execution has been completed.