guardCM modular approach

Introduction

The overall approach to manage the *GuardCM* contract is described here.

Note that there could be CM calls for L2 contracts embedded in the calldata of *schedule()* / *scheduleBatch()* functions. Thus, the L2 calldata needs to be detected and parsed accordingly such that it is verified for being whitelisted when called by the CM on L2-s via the Timelock.

As in case of pure L1 function calls, the function call for L2-s allowed to be called by the CM is set by the Timelock and voted for by the DAO.

Old approach

In the previous approach, in the *GuardCM* implementation there is a hardcoded data verification of a provided payload depending on the L2 chain the CM is trying to execute the transaction on. The downside of such an approach is that the *GuardCM* has to be updated every time the protocol starts interacting with a new L2 chain. This way the code base of the *GuardCM* contract is also extended every time the new logic is added.

New approach

Since each L2 network might have its own contract interaction implementation, but at the same time utilize the same message passing algorithm from chain to chain, the modular bridge verification approach is suggested. This implies the association of each L2 chain Id with the specifics of cross-chain interaction and dedicated constants / contracts corresponding to that chain Id.

Each module comes in the form of an external library (without its own storage) set up with required parameters. When the check takes place, the parsed calldata points to a bridge mediator contract. Using that information, a corresponding library is called to verify the rest of the calldata. If the verification fails, it raises a *revert()* on the verifier side.

Here is a diagram depicting the GuardCM modular architecture.

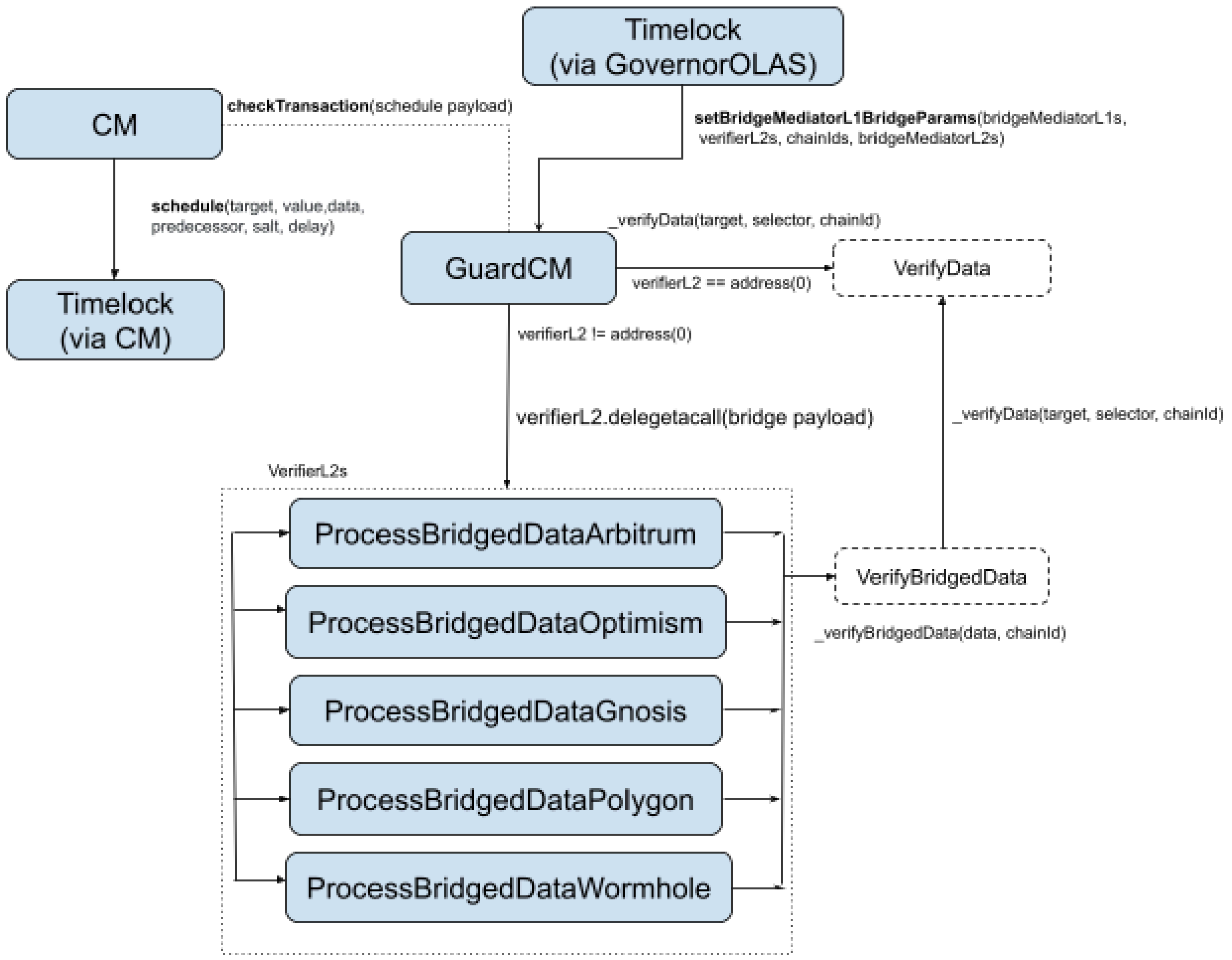
Timelock (via GovernorOLAS)
CM
checkTransaction(schedule payload)
setBridgeMediatorL1BridgeParams(bridgeMediatorL1s, verifierL2s, chainIds, bridgeMediatorL2s)
schedule(target, value,data, predecessor, salt, delay)
_verifyData(target, selector, chainId)
GuardCM
VerifyData
verifierL2 == address(0)
Timelock (via CM)
verifierL2 != address(0)
verifierL2.delegetacall(bridge payload)
_verifyData(target, selector, chainId)
VerifierL2s
ProcessBridgedDataArbitrum
ProcessBridgedDataOptimism
ProcessBridgedDataGnosis
ProcessBridgedDataPolygon
ProcessBridgedDataWormhole
VerifyBridgedData
_verifyBridgedData(data, chainId)

Fig 1. GuardCM new workflow: from schedule to verification

In Fig. 1, one can observe the setup of L2 data verifiers by the means of a *DAO* vote. Note that, after the DAO vote is executed, the Timelock (owner of the *GuardCM*) will update the relevant information in the *GuardCM* contract. Each L2 verifier has a unique association with the L1 bridge mediator that relays data between L1 and L2. Thus, when a call to the L1 bridge mediator is made by the CM, the supported L2 chain has a corresponding verifier contract processing the data validity for that specific L2 chain.

Once the data is decoded and pre-processed by a higher verifier level, including the check for the correct L2 bridge mediator that receives a data message on the L2 side, it is then passed to a *VerifyBridgedData* contract inherited by all the L2 verifiers. There, the data is further decomposed into a set of single transactions, and verified by an atomic *VerifyData* contract function, inherited by L2 verifiers and by *GuardCM* itself. At that stage, each transaction is decomposed into its target, selector and chain Id. If the specified set of arguments has been whitelisted by the DAO, the transaction is verified. The CM is able to complete all the calls after all the transactions have been processed and validated.

Applied code changes

Here are the code changes necessary from the previous approach for implementing the new modular bridge data verification approach.

1. *Timelock*-owned function
setBridgeMediatorChainIds(`address[] memory bridgeMediatorL1s, address[] memory bridgeMediatorL2s, uint256[] memory chainIds`) ***external***

becomes

setBridgeMediatorL1BridgeParams(`address[] memory bridgeMediatorL1s, address[] memory verifierL2s, uint256[] memory chainIds, address[] memory bridgeMediatorL2s`) ***external***

The additional *verifierL2s* parameter simply means the set of L2 data verifier contract addresses on L1, carrying the logic of L2 data pre-processing corresponding to *bridgeMediatorL1s* contract addresses, as described in the suggested approach section.

2. Function ***_verifyData***(`address target, bytes memory data, uint256 chainId`) ***internal***
gets extracted into its own abstract contract, since it is inherited by both *GuardCM* and *L2 verifiers* contracts.

3. Function ***_verifyBridgedData***(`bytes memory data, uint256 chainId`) ***internal***
gets extracted into its own abstract contract that serves as a base of all the *L2 verifiers* contracts.

4. Function
_processBridgeData(`bytes memory data, address bridgeMediatorL2, uint256 chainId`) ***internal***

becomes

processBridgeData(`bytes memory data, address bridgeMediatorL2, uint256 chainId`) ***external***

as each of its L2 specific logic gets added into a corresponding *L2 verifier* contract.

5. Function ***getBridgeMediatorChainId***(`address bridgeMediatorL1`) **external view**
is removed as it becomes obsolete after the parsing of *address* and *uint32* values from a *uint256* value is substituted with a struct, and a default view funcion ABI is created for it.