Axelar - Gateway Governance Audit

Prepared by: Yaar Hahn
Dates of Engagement: 24.07.2023 - 4.08.2023
Version: v1.0

Introduction

Axelar Foundation engaged Yaar to conduct a security audit on their blockchain node and smart contracts beginning on July 24th, 2023 and ending on August 8th, 2023. The security assessment was scoped to the upcoming release of the Gateway Governance feature.

Scope

The main scope for the audit are the additions to `axelar-core` as part of the `v0.34` version. The CGP governance contracts were audited as well.

Detailed scope

- `axelarnetwork/axelar-core`
  - Commit - `6d31bbf407ee7cbe47b9f87b7fe9f2485b5622ac`
  - Diffed against the tag `v0.33.1`, commit `403ff51e04bdc6a4afd10b3a60e813ae1cc362cd`
- `axelarnetwork/axelar-cgp-solidity`
  - Contracts
    - `contracts/governance`
    - `contracts/AxelarGateway.sol`
      - Setup/Upgrade flows
  - Tag `v5.0.0`, commit `9c7a260c848011f27d6e7ecb1cba88de79206ccc`
- `axelarnetwork/axelar-gmp-sdk-solidity`
  - Contracts
    - `Timelock.sol`
  - Tag `v4.0.1`, commit `b062627673a86d4497d59fdaddf54c07578e3bf9`

Mitigations reviewed:

- `axelar-core`
  - Commit `1b6043a1dd689ac9946c308e98d029841dad7ef2`.
- `axelarnetwork/axelar-cgp-solidity`
  - Commit `0f93a25fbb1486a1cdbbc19db4c4569cc1fbefbc`.

Summary

Level	Count
Critical	0
High	1
Medium	2
Low	5
Informational	4

Findings - `axelar-core`

[HIGH-1] `ConfirmDeposit` uses `GetBalance` instead of `SpendableCoins` causing a potential lock of deposit accounts

One of the possible ways to bridge tokens through Axelar from a Cosmos chain is to use a deposit address. The user will send funds to a dedicated deposit address on the Axelar chain through a token transfer (for Axelar->Any transfers) or an IBC ICS20 transfer. A relayer will then call `ConfirmDeposit` that will check the account's balance for a specific denom and initiate the bridge message with that amount. Later on the tokens will move to an escrow address or burned, using `SendCoins` or `SendCoinsFromAccountToModule` from the deposit address. In case the send operation fails, the deposit account is locked.

The issue resides in the `bankK.GetBalance` function. It returns the entire balance of the account, including unvested tokens. An attacker can front-run the token transfer to the deposit account, creating a vesting account in place of the deposit account, with some dust of that denom (e.g. `1uaxl`). The attacker will set the vesting end date to the far future. When the bridge message will be executed, the `SendCoins` call will fail, as the user cannot use the unvested tokens. This will permanently lock the user's deposit account until a fix is implemented for the Axelar chain. The vulnerable code is:

```
// ConfirmDeposit handles deposit confirmations
func (s msgServer) ConfirmDeposit(c context.Context, req
*types.ConfirmDepositRequest) (*types.ConfirmDepositResponse, error) {
    ctx := sdk.UnwrapSDKContext(c)

    depositAddr := nexus.CrossChainAddress{Address: req.DepositAddress.String(),
Chain: exported.Axelarnet}
    amount := s.bank.GetBalance(ctx, req.DepositAddress, req.Denom)
    if amount.IsZero() {
        return nil, fmt.Errorf("deposit address '%s' holds no funds for token %s",
req.DepositAddress.String(), req.Denom)
    }
```

Creating a vesting account is possible through the `vesting` module with the following command:

```
./axelard tx vesting create-vesting-account {DEPOSIT_ADDR} 1{DENOM} {FUTURE_DATE}
```

Deposit addresses are registered in the `axelarnet.Link` handler, where they are created using a randomized salt that make it hard to predict the final address:

```
func NewLinkedAddress(ctx sdk.Context, chain nexus.ChainName, symbol,
recipientAddr string) sdk.AccAddress {
    nonce := utils.GetNonce(ctx.HeaderHash(), ctx.BlockGasMeter())
    hash := sha256.Sum256([]byte(fmt.Sprintf("%s_%s_%s_%x", chain, symbol,
recipientAddr, nonce)))
```

```
    return hash[:address.Len]
}
```

While this holds true, the `Link` handler doesn't initialize the account in the `accountKeeper`, which means an attacker can create the vesting account at any time between the `Link` and the `ConfirmDeposit` calls. It means that time sensitive front-running is not needed.
Notice that this attack only affects new deposit accounts, therefore any existing deposit account is safe.

Recommendation: Replace `GetBalance` with `SpendableCoins` to only count the available coins of the deposit address.

[LOW-1] Missing check for `0x` in EVM address prefix can cause a bypass of the minimal deposit requirement

When the proposal is submitted, the contract address is verified to be a valid address for the relevant chain type:

```
func (h Hooks) AfterProposalSubmission(ctx sdk.Context, proposalID uint64) {
    proposal := funcs.MustOk(h.gov.GetProposal(ctx, proposalID))
...
            crossChainAddress := nexus.CrossChainAddress{Chain: chain, Address:
contractCall.ContractAddress}
            if err := h.nexus.ValidateAddress(ctx, crossChainAddress); err != nil
{
                panic(err)
            }
...
}
```

And this is the validation for EVM addresses:

```
func NewAddressValidator() nexus.AddressValidator {
    return func(ctx sdk.Context, address nexus.CrossChainAddress) error {
        if !common.IsHexAddress(address.Address) {
            return fmt.Errorf("not an hex address")
        }

        return nil
    }
}
```

We can see that addresses without the `0x` prefix are accepted:

```
func IsHexAddress(s string) bool {
    if has0xPrefix(s) {
        s = s[2:]
```

```
    }
    return len(s) == 2*AddressLength && isHex(s)
}
```

In the same time, this is the check for the minimal deposit for a given contract address:

```
func (m callContractProposalMinDepositsMap) Get(chain nexus.ChainName,
contractAddress string) sdk.Coins {
    c := strings.ToLower(chain.String())
    address := strings.ToLower(contractAddress)

    if _, ok := m[c]; !ok {
        return sdk.Coins{}
    }

    return m[c][address]
}
```

We can see that if the address is not found in the minimal deposit map, a zero amount is used.
An attacker can submit a governance proposal to a contract that appears in the deposit map without a deposit by sending the contract address without the `0x` prefix.

Recommendation: Add the prefix of `0x` before searching for the address in the `CallContractProposalMinDeposits` map. The search can be queried twice, once with and once without adding the prefix.
I also recommend checking if there are any other parts of the system affected by the issue, where checks for EVM addresses can be bypassed.

[LOW-2] `AxelarGMPAccount` is compared as string, missing upper-case addresses

When Axelar receives an ICS20 transfer packet, it checks if the receiver is `AxelarGMPAccount`. If it's not it receives the tokens normally, and if it is it parses the memo of the transfer for a GMP message. The address check is currently:

```
func OnRecvMessage(ctx sdk.Context, k keeper.Keeper, ibcK keeper.IBCKeeper, n
types.Nexus, b types.BankKeeper, r RateLimiter, packet ibcexported.PacketI)
ibcexported.Acknowledgement {
...
    // skip if packet not sent to Axelar message sender account
    if data.GetReceiver() != types.AxelarGMPAccount.String() {
        // Rate limit non-GMP IBC transfers
        // IBC receives are rate limited on the Incoming direction (tokens coming
in to Axelar hub).
        if err := r.RateLimitPacket(ctx, packet, nexus.Incoming,
types.NewIBCPath(packet.GetDestPort(), packet.GetDestChannel())); err != nil {
            return channeltypes.NewErrorAcknowledgement(err)
        }
```

```
        return ack
    }
...
```

We can see that `data.GetReceiver()` is the receiver address as string taken directly from the ICS20 packet. `types.AxelarGMPAccount.String()` will always return the lower case bech32 address.
In Cosmos-SDK, addresses can either be lower or upper case. In case a user will send a transfer to `AxelarGMPAccount` in upper case, the check will not be matched, marking this transfer as a normal transfer. This can cause GMP messages to be dropped by Axelar.

Notice that this is a very delicate change, as some chains might rely on only having the lower-case address blocklisted on the sender side.

Recommendation: If this won't affect other chains, I recommend changing the check to first decode the bech32 address of the receiver and then byte-compare the addresses. If decided not to, I would consider blocking transfers to the upper-case account, instead of them being locked on the chain.

[LOW-3] Governance proposals' titles and descriptions cannot contain underscores

Proposals are validated with a `ValidateBasic` function that checks the description and title:

```
func (p CallContractsProposal) ValidateBasic() error {
    if err := utils.ValidateString(p.Title); err != nil {
        return err
    }

    if err := utils.ValidateString(p.Description); err != nil {
        return err
    }
...
}
```

Notice that `ValidateString`'s default behavior is to disallow underscores:

```
func ValidateString(str string, forbidden ...string) error {
    var f string
    if len(forbidden) == 0 {
        f = DefaultDelimiter
...
```

While this is obviously not a security issue, I believe it is not intended.

Recommendation: Add a new function to `utils/slice.go` which validates strings and allows for zero forbidden characters.

[INFO-1] Missing check for chain activation

When initializing a new contract call in `axelarnet.CallContract` the chain is checked to be activated:

```
func (s msgServer) CallContract(c context.Context, req *types.CallContractRequest)
(*types.CallContractResponse, error) {
...
    if !s.nexus.IsChainActivated(ctx, chain) {
        return nil, fmt.Errorf("chain %s is not activated yet", chain.Name)
    }
...
    if err := s.nexus.SetNewMessage(ctx, msg); err != nil {
        return nil, sdkerrors.Wrap(err, "failed to add general message")
    }
...
```

This check is missing in the case of a contract call that was initialized by a governance proposal:

```
func NewProposalHandler(k keeper.Keeper, nexusK types.Nexus, accountK
types.AccountKeeper) govtypes.Handler {
    return func(ctx sdk.Context, content govtypes.Content) error {
        switch c := content.(type) {
        case *types.CallContractsProposal:
            // YH: `s.nexus.IsChainActivated` check is missing, compared to normal
contract call flow.
            for _, contractCall := range c.ContractCalls {
                ...
                if err := nexusK.SetNewMessage(ctx, msg); err != nil {
                    return sdkerrors.Wrap(err, "failed to add general message")
                }
...
```

Recommendation: Add `IsChainActivated` to the proposal handler. The check can also be added to `hooks.AfterProposalSubmission` to fail the proposal as soon as it is submitted.

[INFO-2] Consider using `axelarbankkeeper` for `reward` module

Axelar implemented a bank keeper wrapper for `SendCoins` operations. The wrapper verifies that no blocked addresses are used for coin transfers.
The only module that's not initialized with the keeper is `reward`, probably because it doesn't have any `SendCoins` operations. I recommend considering adding the wrapper in case other checks are added to the wrapper or to `reward`.
It should be noted that `reward` does use `SendCoinsFromModuleToAccount`, which does check for `BlockedAddr` unlike `SendCoins`.

[INFO-3] `GetBalance` is used instead of `SpendableCoins` for proxy activation

When validators register their proxies, a balance check is performed for the new proxy account, as it should hold a minimal amount of the bond denom:

```
func (k Keeper) ActivateProxy(ctx sdk.Context, operator sdk.ValAddress, proxy
sdk.AccAddress) error {
...

    minBalance := k.GetMinProxyBalance(ctx)
    denom := k.staking.BondDenom(ctx)
    if balance := k.bank.GetBalance(ctx, proxy, denom);
balance.Amount.LT(minBalance) {
        return fmt.Errorf("account %s does not have sufficient funds to become a
proxy (minimum %s%s, actual %s)",
            proxy.String(), minBalance.String(), denom, balance.String())
    }
...
```

This issue is similiar to the `ConfirmDeposit` issue - an attacker can create a proxy account that has the bond denom vested but doesn't actually have the coins sepndable.

Recommendation: This doesn't seem like a security violation as the proxy account's tokens are not used for anything but for gas. Still, I recommend replacing `GetBalance` with `SpendableCoins` to only count the available coins of the proxy address.

[INFO-4] Use a dedicated token instead of `1uaxl` for GMP IBC transfers

When sending GMP messages without tokens through IBC, a non-zero token is expected in the ICS20 packet. Axelar bypassed the issue by sending `1uaxl` in the packet (as it has negligible value). Because the sender might not hold enough AXL for the message, Axelar added a `feegranter` mechanism where the sender pre-approves another fee payer. In case of a relayer issue, e.g. downtime, users should be able to call `RouteMessage` themselves to free stuck messages. The `feegranter` mechanism complicates this flow for less technical users.

Recommendation: A new token, e.g. `1ugmp`, can be minted everytime a message is sent and passed with the packet. The `axelarnet` module already has minting capabilities.

Findings - Smart Contracts

[MED-1] Multisig votes doesn't have a unique identifier, causing a partial replay attack

`MultisigBase.sol` implements a multi-signature voting mechanism for governance proposals. Every key holder signs the same contract call to the multisig contract with the same data. The `onlySigners` modifier checks whether enough voters voted on the proposal, according to a set threshold (fixed number of voters). If the threshold is not passed, the vote count is incremented and the voter is marked as voted. If it is passed, the modifier will first clear the voter count and the voter indication and then pass the contract call and execute it. The issue resides in the case that more voters sign the proposal than needed. The first voter to pass the threshold will execute it and clear the counters. The next voters will now vote on a new proposal, with the same topic.
At the extreme case that the threshold for the multisig is lower or equal to 50% (e.g. 2-of-4 multisig), and all of them broadcast their vote, the proposal will be executed twice. Even in the less extreme case where only some of the voters broadcast their vote, or the threshold is higher than 50%, the issue still exists as now a smaller group of voters can practically decide on executing the proposal. The new group is smaller than the real threshold, which breaches the expected behavior. This can obviously lead to major damages, e.g. if the proposal transfers funds that will be double spent.

There are two contracts that inherit `MultisigBase.sol`, `AxelarServiceGovernance.sol` and `Multisig.sol`.
`Multisig` is vulnerable to this attack as it has no way to detect this "replay" attack. It has a single `onlySigners` function which is `execute` which simply executes a contract call as the multisig contract. `AxelarServiceGovernance` has a single `onlySigners` function which is `executeMultisigProposal`. This function relies on pre-approving the multisig to execute the call. It means that in order to exploit this issue, a governance proposal that re-approves the multisig call needs to be in place. This might be possible as I assume that multisig calls will be approved for routine procedures (e.g overriding mint limits). Therefore I believe that the "replay" protection drastically reduces the risk of exploitation, but doesn't completely prevent the attack.
`MultisigBase` has another `onlySigners` function which is `rotateSigners`. Everytime the signers are rotated the epoch is incremented, which also resets the voting counters. Replaying the same proposal might rotate the signers, but to the same set of accounts and threshold. A downgrade attack is prevented by the epoch increment. Therefore the `rotateSigners` is not vulnerable.

It should be noted that in real world applications, multisig votes are usually coordinated in an off-chain communication, so usually there will not be more than needed signers for the proposal.

Recommendation: Implement some kind of nonce mechanism for multisig votes. This is complicated with the current implementation's design where `topic = keccak256(msg.data)`. A possible fix is to keep a list of executed proposals and deny calling the same proposal again. This is problematic as I assume that calling the same contract calls will be a common use case.

[MED-2] Role transfers should be executed in a two steps process

Transfer of governance is a delicate and irreversible process, as it could leave a contract useless. It's recommended to use a two step process, `transferGovernance` and `acceptGovernance`. In case of an issue with the `acceptGovernance` phase, the ownership can be transferred back to the original owner.
CGP has two roles that can be transferred, Governor and MintLimiter. We can see that an issue with the MintLimiter role is reversible with a governance proposal or gateway upgrade, but the Governor role is not.
This is the current `transferGovernance` function:

```
function transferGovernance(address newGovernance) external override
onlyGovernance {
    if (newGovernance == address(0)) revert InvalidGovernance();

    _transferGovernance(newGovernance);
}

function _transferGovernance(address newGovernance) internal {
    emit GovernanceTransferred(getAddress(KEY_GOVERNANCE), newGovernance);

    _setAddress(KEY_GOVERNANCE, newGovernance);
}
```

Recommendation: Split `transferGovernance` to a two-step process.

[LOW-4] Governance proposal's target address is not verified to be a contract

When a governance proposal or multisig operation is executed, the target contract is being called with the `Caller._call()` function. The function doesn't verify that the target is a contract (as expected). In case of a wrong address on a proposal, the contract call might be sent to a non-contract address which might lead to a loss of funds.

Recommendation: Add a code length check to the target address. In order to support native fund transfers, you can consider only checking that in case the call data is not empty. I believe the best place for the check is `InterchainGovernance._execute` and `AxelarServiceGovernance.executeMultisigProposal`.

[LOW-5] (Mitigations) Missing withdraw function in `Multisig.sol`

After modifying `Caller.sol` to only allow calls to contracts, every contract that holds native balances should have a dedicated withdraw function that will send native funds to EOA addresses.
Currently there are two contracts that inherit `Caller`, `InterchainGovernance` and `Multisig`. While the first was modified accordingly, `Multisig` is missing the function.

Recommendation: Add a `withdraw` function to `Multisig`.