Technical Documentation

Summary:

We created a smart contract that serves the function of a financial intermediary between AXA and its customers. It collects premium payments from the customer and transfers claim payments to the customer. Additionally, we implemented a governance model that is defined by a majority voting model. The multisig function that represents our governance model has been built into the smart contract to keep it secure from exploits. Functions with multisig only execute if 3 out of 5 Authorizers agree on the execution. We also implemented a timer which keeps some of the functions secure.

Detailed documentation:

Before the smart contract can be deployed to the blockchain, the person who deploys the contract must specify 5 addresses. These 5 addresses will receive authorization rights which gives them the authority to call certain functions in the contract. The contract creator is not necessarily an Authorizer himself after deploying the contract.

`Bloxure.sol`

This file contains all functions which are responsible for the transfer of premiums and claim payments. It imports from the file `Ownable.sol` which contains all functions of governance. Having the smart contract deployed to the blockchain, the customer can now transfer his premium payments to the contract.

To transfer funds to the smart contract, the customer calls the function `collectPremium()`, which is payable. The amount the customer transfers to the contract will be defined by the AXA webpage. By using Web3 integration the webpage will specify the price for the product and the customer only needs to call the function by agreeing to the transfer. The function `collectPremium()` is the only function that can be called by anyone. All other functions which use the modifier `onlyAuthorized` can only be called from addresses with authorization rights. The function also emits the event `premiumCollected` with the address input `_premiumPayer` and the value input `_premiumAmount`. Both inputs will be emitted as indexes.

The contract can also transfer funds. But before anyone can call the function for the transfer, the Authorizers must agree on it. For security reasons, we implemented a Multisig function so that the contract cannot be exploited from inside of AXA. One Authorizer cannot initiate a transfer without the consensus of the other Authorizers. They must vote and agree on a specified amount and a specified address before the transfer function can be executed.

With the function `votePayClaim(uint _claimAmount,address payable _claimReceiver)` the Authorizers can start a claim assessment. First, one of the Authorizers must specify two details for the vote, the claim amount (which is the first variable) and the claim address (which is the second variable). Executing the function with the specified information as inputs will check if the variable `claimAmount` equals Zero and the variable `claimAddress` equals `address(0)`. This should be the case if there is not a vote happening at the time. By starting a vote,

the function will change the variables `claimAmount` and `claimAddress` to the values of the input variables `_claimAmount` and `_claimReceiver`. Furthermore, the function will map the address of the function caller and changes its Boolean value to true, so that a single "Authorizer" address cannot execute the function more than once. Lastly, it will increase the vote counter variable `voteClaimCounter` by one.

After the voting has started, the remaining "Authorizers" can now vote if they agree with the specified claim amount and the specified claim address. They vote by using the same function as the first "Authorizer" that started the voting. But they have to specify the same inputs as the first "Authorizer" because function `votePayClaim(_claimAmount,_claimReceiver)` checks if the inputs `_claimAmount` and `_claimReceiver` equal their respective variables `claimAmount` and `claimAddress`. Additionally, the function checks if an "Authorizer" address has already voted. If the inputs equal their respective variables and the address has not yet voted, then the vote counter variable `voteClaimCounter` will increase by one. If the requirements have not been met, the Authorizer who called the function will receive an error message. Lastly, the function caller's address will be mapped as true so that they cannot vote a second time. We also implemented a timer variable `claimTimer` . This variable will be set to the time when the vote counter reaches 3 and the block has been created, plus 24 hours. This variable will be important for the next function of the smart contract.

`payClaim()` is a function that can only be executed by addresses with authorization rights. This function transfers the claim amount to the claim address on which the "Authorizers" voted with the function `votePayClaim(_claimAmount,_claimReceiver)`. The requirements for this function to work are as follows: The vote counter variable `voteClaimCounter` must be 3 or larger and the timer variable the current time of the call must be higher than the time that is stored in the timer variable `claimTimer`. The timer we implemented gives AXA the option to cancel the transfer within 24 hours if the need arises. After executing the function `payClaim()` and transferring the claim amount to the claim target, the function calls for the function `resetVotingClaim()` which resets the vote counter, claim amount and claim address variables to its initial states.

If the necessity arises and the function `payClaim()` has not been executed yet, the Authorizers can call the function `cancelTransfer()`. The function will check if the Authorizer address has already voted. We implemented a fail-safe so that only "Authorizers" that agreed in the voting can cancel a claim payment. After checking the address's status the function `resetVotingClaim()` will be called and resets the voting process.

The function `resetVotingClaim()` can only be called internally, which means it can only be called from within other functions. This function resets the entire voting process and resets the variables `voteClaimCounter` and `claimAmount` to zero as well as the variable `claimAddress` to `address(0)`. Furthermore, it sets the mapped addresses to false again, so that the Authorizers can vote again.

Ownable.sol

This file contains all functions that give the “Authorizers” their rights and checks if the caller’s address is one with authorization rights. Additionally, this file contains the functions to transfer authorization rights from an old “Authorizer” address to a new one via Multisig, in case an “Authorizer” loses his private key or an old “Authorizer’s” rights should be revoked. All functions in this file can only be executed from addresses with authorization rights.

The modifier `onlyAuthoizer()` has a simple requirement statement. It checks if the caller address is an “authorizer”. The function `isAuthorized()` returns a Boolean value with the mapping `isAuthorizer`. If the value is `true`, the requirement will be accepted, and if the value is `false`, the modifier will not allow the execution of a function with the modifier attached.

To change the authorization rights from one address to another, the “Authorizers” must vote on the change in the same sense as they would vote for the claim assessment in `SmartContract.sol`. First, the voting must start with the function `voteChangeAuthorizer(_oldAuthorizer, _newAuthorizer)`. To be able to call the function, an “Authorizer” must specify two inputs, the first one being the “Authorizer’s” addresses from which the authorization rights will be revoked and the second one being the new address that is not an “Authorizer” yet. `_oldAuthorizer` is the input for the old address and `_newAuthorizer` is the input for the new address. The function will check if the variables `oldAuthorizer` and `newAuthorizer` are both equal to `address(0)`. This is the case when there is no vote happening at that time. If the two variables are equal to `address(0)` then a new voting will begin. Furthermore, the variable `oldAuthorizer` will be set to the input `_oldAuthorizer` and the variable `newAuthorizer` will be set to the input `_newAuthorizer`. Additionally, the timer variable `voteDeadline` will be set to the time value from when the voting has started and the block has been created, plus 24 hours. By the end of the function the vote counter variable `voteAuthorizerCounter` increases by one. The timer variable is important for the function `cancelVote()`. Lastly, the address of the caller will be set to true via mapping.

The other “Authorizers” can use the same function with which the voting was started to cast their vote. But they must specify the same inputs as the one that started the voting. Otherwise, the function will return an error and the vote counter will not increase. In addition, the function uses the mapping `voteChangeAuthorizer` to check whether the caller address has already voted once. This keeps “Authorizers” from voting more than once. If the requirements have been met, the vote counter variable `voteAuthorizerCounter` increases by one, and the caller address will be mapped as `true`.

After the vote counter variable `voteAuthorizerCounter` reaches 3 or more, an “Authorizer” can call the function `transferAuthorizer()` which transfers the authorization rights from the old address to the new one. For the function to work, the vote counter must be 3 or more, else the function will not be executed, and an error message will show. If the requirement has been met, the function will look for the index of the old address with the help of the internal function `getIndex()`. This function looks in the array `authorized` for the index of the old address. With this index number, the old address will be replaced with the new address in the array `authorized` and changes the Boolean values for the old address to `false` and the new address to `true`. This happens in the mapping `isAuthorizer`. In the end, the function calls the function `resetVotingCounter()`.

If “Authorizers” do not agree with the vote to change the authorization rights from an old address to a new one, they can call the function `cancelVote()`. This function can only be called if the current time value from calling the function is higher than the value of the timer variable `voteDeadline`. This keeps old Authorizer from exploiting this function. If the requirements have been met, the function `resetVotingCounter()` will be executed.

The internal function `resetVotingCounter()` resets the voting process to its initial state. It sets the variables `oldAuthorizer` and `newAuthorizer` to the address `address(0)`. It also turns the Boolean values in the mapping `votedChangeAuthorizer` to false so that all “Authorizers” can start a new vote if they see it as necessary.