Corrigé Mini Audit - Smart contract

Partie 1

1. Verrouiller pragma à une version précise et utiliser la bonne version
2. Utilisez "SPDX-License-Identifier : UNLICENSED" pour un code non open-source. Pour plus d'informations, veuillez consulter le site.
3. Importer la librairie SafeMath.
4. Utiliser constructor au lieu de “function Crowdsale”.
5. Utiliser msg.sender au lieu de tx.origin.
6. La fonction fallback doit être external et payable pour recevoir les ethers.
7. L’adresse escrow doit être payable, car elle va recevoir des ethers.
8. Fonction fallback (ou receive): Il est recommandé d’utiliser .transfer au lieu de .send car .transfer revert si une exception se déclenche alors que .send retourne false et continue l’exécution. Pour éviter ce problème qui peut fausser les résultats sur notre smart contract, vous avez deux façons de faire :
   - Garder .send, gérer la valeur du retour et mettre un require .

   ```
   bool success;
        (success) = escrow.send(msg.value);
        require(success);
   ```

   - Utiliser tout simplement .transfer qui revert si une exception.
   La fonction fallback (ou receive) doit avoir une visibilité “external”.
9. Fonction withdrawPayments :
   - L’adresse **payee** doit être déclarée en payable.
   - Attention, à la réentrance au niveau du paiement. Un attaquant peut faire appel à la fonction n fois avant la fin de l’instruction `payee.send(payment);`
   - Même problème avec .send (référez vous à l’explication point 6). - Améliorer le code de votre fonction, en ajoutant des conditions de vérification sur la balances de l’investisseur ainsi que la balance du smart contract pour éviter de dépenser du gas sur une fonction qui va échouer.

   ```
        require(payment != 0);
        require(address(this).balance >= payment);
   ```

   - Pour résoudre l’attaque de réentrance, il faut remettre la balance de l’investisseur à 0 avant d’effectuer le remboursement.

   ```
        savedBalance = savedBalance.sub(payment);
        balances[payee] = 0;
        payee.transfer(payment);
   ```

10. Revoir la logique du remboursement et l’aligner avec le processus de la collecte des fonds. Il est fortement conseillé d’utiliser le pattern PullPayment.
    Explication : Actuellement, un investisseur ne peut pas être sur de pouvoir récupérer ses fonds envoyés. En effet, lors de l’envoi des ethers vers le Crowdsale par un investisseur le smart contract effectue immédiatement un transfert du montant reçu vers l’escrow wallet.
    Imaginons le scénario suivant: - Un investisseur envoie 2 ETH au Crowdsale. - Après 2h, l’investisseur n’est plus emballé et préfère récupérer ses 2 ETH. - L’investisseur fait alors appelle à la fonction withdrawPayments, sauf que cette dernière va lever une exception et les fonds ne seront pas récupérés.
    Pourquoi ? Car, notre contrat à 0 ETH et donc il ne peut pas effectuer le remboursement.
    Solution ? - Laisser les ETHs sur le Crowdsale (pas recommandé et il y a un risque d’attaques). Effectuer un transfert des fonds collectés de l’adresse du wallet escrow vers le smart contract Crowdsale. - Déterminer une période de remboursement, avant de déclencher cette période il faut penser à effectuer un transfert des fonds collectés de l’adresse du wallet escrow vers le smart contract Crowdsale.
11. Ajouter des events pour mieux tracer l’activité sur votre smart contract.
12. Revoir la visibilité des variables

Partie 2

Version 1 avec escrow

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.11;
import "github.com/OpenZeppelin/openzeppelin-solidity/contracts/math/SafeMath.sol";
contract Crowdsale {
  using SafeMath for uint256;
  address public owner; // the owner of the contract
  address payable public escrow; // wallet to collect raised ETH
  uint256 public savedBalance = 0; // Total amount raised in ETH
  mapping (address => uint256) public balances; // Balances in incoming Ether
  // Event to record each time Ether is paid out
  event PayEther(
  address indexed _receiver,
  uint256 indexed _value,
  uint256 indexed _timestamp
  );
  // Initialization
  constructor (address payable _escrow) public{
      owner = msg.sender;
      // add address of the specific contract
      escrow = _escrow;
  }
   // function to receive ETH
  receive() payable external {
      balances[msg.sender] = balances[msg.sender].add(msg.value);
      savedBalance = savedBalance.add(msg.value);
      escrow.transfer(msg.value);
      emit PayEther(escrow, msg.value, now);
  }
   // refund investisor
  function withdrawPayments() public{
      address payable payee = msg.sender;
      uint256 payment = balances[payee];
      require(payment != 0);
      require(address(this).balance >= payment);

      savedBalance = savedBalance.sub(payment);
      balances[payee] = 0;

      payee.transfer(payment);
      emit PayEther(payee, payment, now);
  }
}
```

Version 2 sans escrow

```
/ SPDX-License-Identifier: MIT
pragma solidity 0.6.11;
import "github.com/OpenZeppelin/openzeppelin-solidity/contracts/math/SafeMath.sol";
contract Crowdsale {
  using SafeMath for uint256;
  uint256 public savedBalance = 0; // Total amount raised in ETH
  mapping (address => uint256) public balances; // Balances in incoming Ether
  // Event to record each time Ether is paid out
  event PayEther(
  address indexed _receiver,
  uint256 indexed _value,
  uint256 indexed _timestamp
  );
  // function to receive ETH
  receive() payable external {
      balances[msg.sender] = balances[msg.sender].add(msg.value);
      savedBalance = savedBalance.add(msg.value);
      emit PayEther(address(this), msg.value, now);
  }
   // refund investisor
  function withdrawPayments() public{
      address payable payee = msg.sender;
      uint256 payment = balances[payee];
      require(payment != 0);
      require(address(this).balance >= payment);

      savedBalance = savedBalance.sub(payment);
      balances[payee] = 0;

      payee.transfer(payment);
      emit PayEther(payee, payment, now);
  }
}
```