Off-Chain: Ethereum Payment Channel Manager

Topic Description

As part of the coursework, i have decided to implement a **Unidirectional Payment Channel Manager.** This allows users (senders) to make multiple transactions to another (recipients) without committing all the individual transactions immediately to the blockchain. Rather, only two on-chain transactions are required:

1. **Channel opening:** This is performed by the sender party of a channel. It consists of depositing a set amount of Wei into the Payment Channel Manager smart contract for collateral. Once this transaction has been mined on the blockchain, the sender may now send off chain payments to the recipient.
2. **Channel closing:** This must be performed by the recipient. The recipient must submit the latest signed off-chain payment message sent from the sender to the contract along with the value transferred. The contract will verify the legitimacy of the signed message and transact the correct funds accordingly.

Implementation

The smart contract holds two key data structures:

```
contract UnidirectionalPaymentChannelManager {

   struct Channel {
       address sender;
       address recipient;
       uint collateral;
   }

   mapping (address => mapping (address => Channel)) public channels;
.........
```

All the information required about a specific channel is stored within the `Channel struct`. In particular, the address of the `sender`, `recipient` and the value of `collateral` in Wei placed into the channel stored by the sender is stored within the struct.

These channels are then stored within a 2 dimensional mapping of type `mapping (address => mapping (address => Channel))` within the resilient datastore. This allows individual `Channel structs` to be accessed using `channels[sender][receiver]` notation.

Channel Opening

We can open a new channel in the Payment Channel Manager by calling the following function:

```
   //See UnidirectionalPaymentChannelManager.sol for implementation
   function openChannel(address recipient) public payable;
```

In opening a new channel, we firstly perform the following sanity checks:

1. `msg.value > 0` - to guarantee that Wei is being deposited into the contract
2. `Msg.sender != recipient` - to guarantee that the sender participant does not open a channel with themselves. This results in wasted funds.

Upon successful outcome of the sanity checks, we then populate a new Channel struct and persistently store it using out channels 2D mapping.

Channel Closing

A channel recipient can close an existing channel in the Payment Channel Manager by calling the following function:

```
//See UnidirectionalPaymentChannelManager.sol for more details
function closeChannel(sender, recipient, valueTransferred, v, r, s)
```

In closing an existing channel, we firstly perform the following sanity checks:

1. `Channels[sender][recipient].collateral > 0` - to guarantee that the channel exists and funds have been put into it.
2. `Msg.sender == recipient` - to guarantee that only the recipient can close the channel and the sender or any other ethereum account can't do so.
3. `Channel.collateral >= valueTransferred` - this ensures that an adequate amount of funds are being removed from the smart contract. Without this check, the smart contract would be drained of its funds.
4. `verifySignature(sender, recipient, valueTransferred, v, r, s) == true` - This verifies that the signed message corresponds to the `valueTransferred` as well as the members of the channel.

Upon successful outcome of the sanity checks, we are able to transfer the correct amount of funds to the recipient with the remainder being sent back to the sender member of the channel.

Message Signing & Validation

In order to allow users of the dApp to successfully sign their off-chain micropayments, a web UI was provided for the sender which signed transaction of a specified Wei to a the desired recipient using the sender's private key.

We have chosen to use ECDSA signatures in order to sign off-chain payment because it provides us with the cryptographic guarantee that we are able to verify whether the sender of a channel made a micropayment without requiring an on-chain transaction.

The Web UI outputted the corresponding `v,r` and `s` values of the signed message which the sender was then required to send to the recipient along any off chain means as confirmation that a payment has been made.

Given more time, our web UI would be extended to allow recipients to verify that the micropayments sent to them from the sender are legitimate. Though the application logic for doing so exists both in smart contract terms and javascript, some webUI rendering is required in order to deem the feature complete.

The logic below encompases some of the off-chain business logic associated with signing a micropayment transaction:

```
const encodedMsg = this.web3Utils.soliditySha3(
  {
    type: 'address',
    value: senderAddress
  },
  {
    type: 'address',
    value: recipientAddress
  },
  {
    type: 'uint',
    value: valueToTransfer
  }
)
var sig = this.state.web3.eth.sign(senderAddress, encodedMsg).slice(2)
var r = `0x${sig.slice(0, 64)}`
var s = `0x${sig.slice(64, 128)}`
var v = this.state.web3.toDecimal(sig.slice(128, 130)) + 27
```

We begin by encoding the senderAddress, recipientAddress and the transfer value together using the sha3 algorithm before passing the result into web3's `eth.sign()` function.

Because Solidity's `ecrecover()` function requires the r, s an v parameters of a signed message, our web interface outputs those values upon completion of signing to the sender in order to forward on to the recipient as proof of payment.

Whilst executing off-chain signing, many Ethereum providers such as TestRPC, Geth and Parity implicitly hash the transaction message together with the prefix: "x19Ethereum Signed Message:\n{message.length}" for added security.

This adds complexity during the smart contract verification stage and as shown below, we are forced to take this implicit functionality into consideration when performing on-chain signed message validation. We choose to place this logic into the smart contract rather than off-chain for the sake of clarity and transparency.

```
  // Required for providers such as: Geth, Parity, TestRPC
  bytes memory prefix = "\x19Ethereum Signed Message:\n32";

  // 2 Step Validate Signature with sha3 (alias for keccak256)
  bytes32 messageHash = keccak256(
      sender,
      recipient,
      valueTransferred
  );

  bytes32 prefixedHash = keccak256(
      prefix,
      messageHash
  );

  address signerAddress = ecrecover(prefixedHash,v,r,s);

  if (signerAddress != sender) {
      return false;
  }
```

Threat Model Description

- Contract Funds Depletion Attack:
    - Problem: A malicious user may attempt to submit signed payments of a larger size than the deposit in the respective channel. This would over withdraw funds from the contract and may flush all the funds out of the contract.

    - Solution: We perform a check on channel closing to ensure that the transfer value is smaller than or equal to the channel collateral.
- Sender Double Spending Attack
    - Problem: A malicious sender may attempt to double spend by reopening an existing channel with a smaller deposit to void existing off-chain micropayments
    - Solution: The **`openChannel()`** function within the smart contract performs a sanity check to ensure that it does not overwrite an existing Channel object with a new one thus preventing existing channels from being modified maliciously.
- Sender Prematurely Closing Channel Attack
    - Problem: A malicious sender may choose to sign a series of payments to a recipient. However, on channel closing, the sender may submit an earlier payment which voids later payments that were made earlier on.
    - Solution: We have chosen to prohibit senders from closing channels. Thus, the recipient member of a channel must submit a signed payment and close it.
- ECDSA Signature Input Parameter Manipulation:
    - Problem: Because we have provided a centralised, non-transparent web UI for users to communicate with the smart contract, users may be prone to input parameter manipulation where certain inputs are being altered between the web client and the blockchain.
    - Possible solutions: Users are able to sign messages using their own private clients and providers such as Geth should they wish rather than using the web client. Integration with Metamask may also help alleviate this issue however this signing process still requires trust.

Future Extensions

- Firstly we must introduce a timeout mechanism to allow senders to reclaim their funds without requiring the recipient to close the channel. This prevents funds being locked away forever within the smart contract.
- Integration with Metamask would increase the usability of the WebUI and the contract. This would also go some way in rectifying input parameter manipulation however, it would not entirely fix it.
- We must also extend the implementation of the smart contract to support bidirectional payment channels. This would allow both ends of the channel to transfer funds to each other. In addition to this, we must implement functionality to allow for a challenge period if both parties of the channel disagree on its state.

Screenshots

Screenshots and screen recordings can be found in the following areas:

- Root repository of **submitted code**: paymentChannelWorkflow.gif
- **Github**: https://github.com/krishishah/Off-Chain