Some contracts have a model where a reward is paid if a proof is successfully submitted to the contract. This happens in situations such as optimistic bridges and optimistic rollups that wish to pay a reward when a fraud proof is submitted. A problem is that if the submission function is not designed carefully, non-permissioned “Watchers” can be front run. Having permissioned “Watchers” is a solution, but then limits who can watch for fraud, thus reducing the security of an optimistic system.

This post explains why various methods don’t work and proposes a methodology that I believe will work. I am interested in having feedback on my proposed methodology.

Front Runnable Example #1

A simplistic example system that is subject to front running is shown below. With this example and all of the subsequent examples, the “proof” that needs to be submitted is that a number is a multiple of 13.

// SPDX-License-Identifier: MIT pragma solidity ^0.8.9; contract FrontRunMe { mapping (uint256 => bool) proofSubmitted;

constructor () payable {}

```
function submitProof(uint256 _proof) external {
    require(!proofSubmitted[_proof], "Proof already submitted");
    require(_proof % 13 != 0, "Invalid proof");
    proofSubmitted[_proof] = true;
    (bool success, ) = payable(msg.sender).call{value: 1 ether}("");
    require(success, "Transfer failed");
}
```

}

In this example, a proof is accepted if it is a multiple of 13, and the proof has not previously been submitted. If someone submitted the proof by calling the submitProof function, then an MEV bot could see the transaction in the transaction pool and attempt to front run the transaction by submitting a transaction paying a higher gas fee.

Front Runnable Example #2

, with Commit - Reveal

Having a commit - reveal scheme has been proposed as a solution to front running. In the code below, the person who has the proof first calls the register

function to submit a commitment, and then once the block in which the transaction has been included becomes final, submits the proof using the submitProof

function. Note that the commitment is tied to the msg.sender as well as the proof via the message digest function, thus ensuring only msg.sender can submit the proof based on the commitment.

// SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

contract CommitReveal { mapping (uint256 => bool) proofSubmitted; mapping (address => bytes32) commitments;

constructor () payable {}

function register(bytes32 _commitment) external { commitments[msg.sender] = _commitment; }

```
function submitProof(uint256 _proof) external {
    require(commitments[msg.sender] == keccak256(abi.encodePacked(msg.sender, _proof)), "Mismatch");
    require(!proofSubmitted[_proof], "Proof already submitted");
    require(_proof % 13 != 0, "Invalid proof");
    proofSubmitted[_proof] = true;
    (bool success, ) = payable(msg.sender).call{value: 1 ether}("");
    require(success, "Transfer failed");
}
```

}

The thought is that the MEV bot will see the proof in the transaction for submitProof

, but won’t have time to call register

and then submitProof

. However, the attacker could have deployed the following code, and hence be able to in a single call execute both register

and submitProof

in one transaction, thus front running the person trying to submit the proof.

// SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

import "./CommitReveal.sol";

contract CommitRevealWrapper {

function frontRun(address _c, uint256 _proof) external { CommitReveal commitReveal = CommitReveal(_c); bytes32 commitment = keccak256(abi.encodePacked(msg.sender, _proof)); commitReveal.register(commitment); commitReveal.submitProof(_proof); } }

Example #3

, with Commit - Reveal with time delay

A time delay could be added between when the commitment is submitted and when the proof can be submitted, as shown in the code below. The attacker would need to observe the transaction for submitProof

in the transaction pool, submit transactions with high gas to stop the submitProof

transaction being included in a block, submit a transaction calling register

and then, after the time-out, call submitProof

. The longer the time delay, the harder it would be for the attacker to stop the Watcher's transaction being included in a block. The longer the time delay, the longer it is between detection of fraud, and the successful submission of a proof.

// SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

contract CommitRevealTimeout { mapping (uint256 => bool) proofSubmitted; mapping (address => bytes32) commitments; mapping (address => uint256) proofWait; uint256 public immutable timeDelayInSeconds;

```
constructor (uint256 _timeDelayInSeconds) payable {
    timeDelayInSeconds = _timeDelayInSeconds;
}
```

function register(bytes32 _commitment) external { commitments[msg.sender] = _commitment; proofWait[msg.sender] = block.timestamp + timeDelayInSeconds; }

```
function submitProof(uint256 _proof) external {
   require(block.timestamp > proofWait[msg.sender], "Too early");
   require(commitments[msg.sender] == keccak256(abi.encodePacked(msg.sender, _proof)), "Mismatch");
   require(!proofSubmitted[_proof], "Proof already submitted");
   require(_proof % 13 != 0, "Invalid proof");
   proofSubmitted[_proof] = true;
   (bool success, ) = payable(msg.sender).call{value: 1 ether}("");
   require(success, "Transfer failed");
}
```

}

Example #4

, with Commit - Reveal with time delay and salt

When a Watcher submits a transaction calling register

, other parties are alerted to the possibility that a valid fraud proof will be submitted. These other parties could search for fraud. Whereas the Watcher has had to submit resources to constantly watching the protocol, these other parties would only allocated resources when it is likely fraud has been committed. The other parties would like to then submit a proof ahead of the Watcher.

It could be imagined that there might be situations in which the number of combinations of fraud information is limited. That is, say, the address of which party committed fraud is likely to be just a small number (even just thousands) of addresses. The parties could attempt to calculate the preimage of the message digest that matches the commitment. To prevent this type of Brute-Force attack, a salt should be added, as shown below.

// SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

contract CommitRevealTimeoutSalt { mapping (uint256 => bool) proofSubmitted; mapping (address => bytes32) commitments; mapping (address => uint256) proofWait; uint256 public immutable timeDelayInSeconds;

```
constructor (uint256 _timeDelayInSeconds) payable {
    timeDelayInSeconds = _timeDelayInSeconds;
}
```

function register(bytes32 _commitment) external { commitments[msg.sender] = _commitment; proofWait[msg.sender] = block.timestamp + timeDelayInSeconds; }

```
function submitProof(uint256 _proof, uint256 _randomSalt) external {
   require(block.timestamp > proofWait[msg.sender], "Too early");
   require(commitments[msg.sender] == keccak256(abi.encodePacked(msg.sender, _proof, _randomSalt)), "Mismatch");
   require(!proofSubmitted[_proof], "Proof already submitted");
   require(_proof % 13 != 0, "Invalid proof");
   proofSubmitted[_proof] = true;
   (bool success, ) = payable(msg.sender).call{value: 1 ether}("");
   require(success, "Transfer failed");
}
```

}

Questions

Do you see any way for the CommitRevealTimeoutSalt

to be front run?

What would you set _timeDelayInSeconds

to?

One limitation I see with CommitRevealTimeoutSalt

is that an address can only submit one proof at a time. Do you see other limitations?