Transaction Order Dependence

Description

The Ethereum network processes transactions in blocks with new blocks getting confirmed around every 17 seconds. The miners look at transactions they have received and select which transactions to include in a block, based who has paid a high enough gas price to be included. Additionally, when transactions are sent to the Ethereum network they are forwarded to each node for processing. Thus, a person who is running an Ethereum node can tell which transactions are going to occur before they are finalized. A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.

The simplest example of a race condition is when a smart contract gives a reward for submitting information. Say a contract will give out 1 token to the first person who solves a math problem. Alice solves the problem and submits the answer to the network with a standard gas price. Eve runs an Ethereum node and can see the answer to the math problem in the transaction that Alice submitted to the network. So, Eve submits the answer to the network with a much higher gas price and thus it gets processed and committed before Alice's transaction. Eve receives one token and Alice gets nothing, even though it was Alice who worked to solve the problem. A common way this occurs in practice is when a contract rewards people for calling out bad behavior in a protocol by giving a bad actor's deposit to the person who proved they were misbehaving.

The race condition that happens the most on the network today is the race condition in the ERC20 token standard. The ERC20 token standard includes a function called 'approve' which allows an address to approve another address to spend tokens on their behalf. Assume that Alice has approved Eve to spend n of her tokens, then Alice decides to change Eve's approval to m tokens. Alice submits a function call to approve with the value n for Eve. Eve runs a Ethereum node so knows that Alice is going to change her approval to m. Eve then submits a tranferFrom request sending n of Alice's tokens to herself but gives it a much higher gas price than Alice's transaction. The transferFrom executes first so gives Eve n tokens and sets Eve's approval to zero. Then Alice's transaction executes and sets Eve's approval to m. Eve then sends those m tokens to herself as well. Thus Eve gets n + m tokens even though she should have gotten at most max(n,m).

Remediation

A possible way to remedy for race conditions in submission of information in exchange for a reward is called a commit reveal hash scheme. Instead of submitting the answer the party who has the answer submits hash(salt, address, answer) [salt being some number of their choosing] the contract stores this hash and the sender's address. To claim the reward the sender then submits a transaction with the salt, and answer. The contract hashes (salt, msg.sender, answer) and checks the hash produced against the stored hash, if the hash matches the contract releases the reward.

The best fix for the ERC20 race condition is to add a field to the inputs of approve which is the expected current value and to have approve revert if Eve's current allowance is not what Alice indicated she was expecting. However this means that your contract no longer

conforms to the ERC20 standard. If it important to your project to have the contract conform to ERC20, you can add a safe approve function. From the user perspective it is possible to mediate the ERC20 race condition by setting approvals to zero before changing them.

Example:

Code:

```
pragma solidity ^0.4.16;

contract EthTxOrderDependenceMinimal {
    address public owner;
    bool public claimed;
    uint public reward;

    function EthTxOrderDependenceMinimal() public {
        owner = msg.sender;
    }

    function setReward() public payable {
        require (!claimed);

        require(msg.sender == owner);
        owner.transfer(reward);
        reward = msg.value;
    }

    function claimReward(uint256 submission) {
        require (!claimed);
        require(submission < 10);

        msg.sender.transfer(reward);
        claimed = true;
    }
}
```

Explanation:
ClaimReward allows a user to submit the submission value to get the reward. During the network's transaction approval phase, an attacker may inspect the submission value and resubmit it with a sufficiently large gas value to run before the user transaction.