Problem 1: Solidity bugs.

The following buggy Solidity contract implements a fund-raise for an independent film project by an unknown director. Anyone can send ether to the contract and receive tokens for their contribution, as implemented in the fallback function below. Once the film is finished, royalties from the film will be distributed to all participants based on token ownership. The film producer will fund the production of the film by withdrawing ether from the contract using the `withdraw` function below. The contract should ensure that every month the producer can only withdraw 1 more ether than has ever been withdrawn.

```
contract MovieToken {
  address owner;                            // record the producer's address
  mapping (address => uint256) balances;  // investor balances
  uint256 totalSupply;                      // total supply in contract
  uint256 lastWithdrawDate;                 // last withdrawal time

  function MovieToken() {                   // Constructor
    owner = msg.sender;
    balances[owner] = msg.value;            // The producer may fund the movie
    totalSupply     = balances[owner];
  }

  function withdraw(uint256 amount) {     // note: function is not payable

    if (msg.sender != owner) { throw; }   // Only the producer may withdraw funds
    if (amount == 0 || amount > this.balance) { throw; }
    if (now < lastWithdrawDate + (1 months))  { throw; } // Only once per month
    lastWithdrawDate = now;

    // Withdraw schedule: only withdraw 1 more ether than has ever
    // been withdrawn.
    uint256 maxAmount = (totalSupply - this.balance + (1 ether));
    if (amount > maxAmount) { throw; }

    if (!owner.send(amount)) { throw; }  // send the funds to the producer
  }

  // Fallback function: record a fund-raise contribution
  function () payable {
    balances[msg.sender] += msg.value;   // transfer tokens to investor
    totalSupply += msg.value;            // record amount
  }
}
```

A) What is the maximum amount that can be withdrawn in each of the first four months, assuming the maximum amount is withdrawn every month?

month 1: ____ , month 2: ____ , month 3: ____ , month 4: ____ .

B) Describe an attack that lets the producer withdraw all the funds sent to the project immediately on the first month.
Hint: if someone sends funds to a contract address *before* the contract is created, then upon creation of a contract at that address, the variable `this.balance` is equal to the amount of funds sent prior to creation. Note also that all variables in the `MovieToken` contract are *unsigned* integers, so that -1 is actually $2^{256} - 1$.

C) When a contract, say contract X, executes a `selfdestruct` with contract Y's address as an argument, this kills contract X and adds X's balance to Y, without triggering any code execution at Y. With this in mind, suppose we try to prevent your attack from part (B) by changing the second line in the constructor to:

```
balances[owner] = this.balance + msg.value;
```

Is there an attack that lets the producer withdraw all the funds on the first month?

D) How should the contract be fixed so that it correctly implements the intended withdrawal schedule, without changing the program logic?
Hint: add at most four lines to the contract.

Problem 2: **Hash functions and proofs of work.** In class we defined two security properties for a hash function, one called collision resistance and the other called proof-of-work security. Show that a collision-resistant hash function may not be proof-of-work secure.

Hint: let $H : X \times Y \to \{0, 1, \ldots, 2^n - 1\}$ be a collision-resistant hash function. Construct a new hash function $H' : X \times Y \to \{0, 1, \ldots, 2^m - 1\}$ (where m may be greater than n) that is also collision resistant, but for a fixed difficulty D (say, $D = 2^{32}$) is not proof-of-work secure with difficulty D. That is, for every puzzle $x \in X$ it should be trivial to find a solution $y \in Y$ such that $H'(x, y) < 2^m/D$. This is despite H' being collision resistant. Remember to explain why your H' is collision resistant, that is, explain why a collision on H' would yield a collision on H.

Problem 3: **Beyond binary Merkle trees.** Alice can use a binary Merkle tree to commit to a list of elements $S = (T_1, \ldots, T_n)$ so that later she can prove to Bob that $S[i] = T_i$ using an inclusion proof containing at most $\lceil \log_2 n \rceil$ hash values. The binding commitment to S is a single hash value. In this question your goal is to explain how to do the same using a k-ary tree, that is, where every non-leaf node has up to k children. The hash value for every non-leaf node is computed as the hash of the concatenation of the values of all its children.

a. Suppose $S = (T_1, \ldots, T_9)$. Explain how Alice computes a commitment to S using a ternary Merkle tree (i.e., $k = 3$). How does Alice later prove to Bob that T_4 is in S? What values are provided in the proof?

b. Suppose S contains n elements. What is the length of the proof that proves that $S[i] = T_i$, as a function of n and k?

c. For large n, if we want to minimize the proof size, is it better to use a binary or a ternary tree? Why?

Problem 4: (Ethereum programming) The contract code presented below is an attempt to implement a two-player game (with a wager on the line) of Tic-Tac-Toe, also known as Noughts and Crosses:

This implementation contains *at least* 9 serious bugs which compromise the security of the game. Identify 4 of them and briefly describe how they might be fixed. Recall that Ethereum initializes all storage to zero. Assume that the function `checkGameOver()` is correctly implemented and returns true if either player has claimed three squares in a row on the current board.

```
contract TicTacToe {
    // game configuration
    address[2] _playerAddress;     // address of both players
    uint32     _turnLength;        // max time for each turn

    // nonce material used to pick the first player
    bytes32    _p1Commitment;
    uint8      _p2Nonce;

    // game state
    uint8[9]   _board;             // serialized 3x3 array
    uint8      _currentPlayer;     // 0 or 1, indicating whose turn it is
    uint256    _turnDeadline;      // deadline for submitting the next move


    // Create a new game, challenging a named opponent.
    // The value passed in is the stake which the opponent must match.
    // The challenger commits to its nonce used to determine first mover.
    function TicTacToe(address opponent, uint32 turnLength,
                                         bytes32 p1Commitment) {
        _playerAddress[0] = msg.sender;
        _playerAddress[1] = opponent;
        _turnLength = turnLength;
        _p1Commitment = p1Commitment;
    }

    // Join a game as the second player.
    function joinGame(uint8 p2Nonce) {
        // only the specified opponent may join
        if (msg.sender != _playerAddress[1])
            throw;

        // must match player 1's stake.
        if (msg.value < this.balance)
            throw;

        _p2Nonce = p2Nonce;
    }

    // Start the game by revealing player 1's nonce to choose who goes first.
    function startGame(uint8 p1Nonce) {
        // must open the original commitment
        if (sha3(p1Nonce) != _p1Commitment)
            throw;

        // XOR both nonces and take the last bit to pick the first player
        _currentPlayer = (p1Nonce ^ _p2Nonce) & 0x01;

        // start the clock for the next move
        _turnDeadline = block.number + _turnLength;
    }

    // Submit a move
```

```
    function playMove(uint8 squareToPlay) {
        // make sure correct player is submitting a move
        if (msg.sender != _playerAddress[_currentPlayer])
            throw;

        // claim this square for the current player.
        _board[squareToPlay] = _currentPlayer;

        // If the game is won, send the pot to the winner
        if (checkGameOver())
            suicide(msg.sender);

        // Flip the current player
        _currentPlayer ^= 0x1;

        // start the clock for the next move
        _turnDeadline = block.number + _turnLength;
    }

    // Default the game if a player takes too long to submit a move
    function defaultGame() {
        if (block.number > _turnDeadline)
            suicide(msg.sender);
    }
}
```

Problem 5: (Ethereum programming) The following Solidity contract implements a distributed ticket sales system. Anybody can create an event (specifying the initial price and number of tickets). Anybody can then purchase one of the initial tickets or sell those tickets peer-to-peer. At the event, gate agents will check that each attendee is listed in the final attendees list on the blockchain.

Surprise! This contract is poorly implemented. Identify at least 5 different problems that cause a specific functionality loss or security vulnerability (not simply things that are generally bad practice). For each problem, state what the implications are (e.g. how it can be exploited or what intended functionality won't work) as well as how it could be fixed.

```
pragma solidity ^0.4.0;
contract TicketDepot {

   struct Event{
        address owner;
        uint64 ticketPrice;
        uint16 ticketsRemaining;
        mapping(uint16 => address) attendees;
   }

   struct Offering{
        address buyer;
        uint64 price;
        uint256 deadline;
   }

   uint16 numEvents;
   address owner;
   uint64 transactionFee;
   mapping(uint16 => Event) events;
   mapping(bytes32 => Offering) offerings;

   function ticketDepot(uint64 _transactionFee){
       transactionFee = _transactionFee;
       owner = tx.origin;
   }

   function createEvent(uint64 _ticketPrice, uint16
_ticketsAvailable) returns (uint16 eventID){
```

```
        numEvents++;
        events[numEvents].owner = tx.origin;
        events[numEvents].ticketPrice = _ticketPrice;
        events[numEvents].ticketsRemaining = _ticketsAvailable;
        return numEvents;
    }

    modifier ticketsAvailable(uint16 _eventID){
        _;
        if (events[_eventID].ticketsRemaining <= 0) throw;
    }

    function buyNewTicket(uint16 _eventID, address _attendee)
ticketsAvailable(_eventID) payable returns (uint16 ticketID){
        if (msg.sender == events[_eventID].owner || msg.value >
events[_eventID].ticketPrice + transactionFee){
            ticketID = events[_eventID].ticketsRemaining--;
            events[_eventID].attendees[ticketID] = _attendee;
            events[_eventID].owner.send(msg.value - transactionFee);
            return ticketID;
        }
    }

    function offerTicket(uint16 _eventID, uint16 _ticketID, uint64
_price, address _buyer, uint16 _offerWindow) {
        if (msg.value < transactionFee) throw;
        bytes32 offerID = sha3(_eventID+_ticketID);
        if (offerings[offerID] != 0) throw;
        offerings[offerID].buyer = _buyer;
        offerings[offerID].price = _price;
        offerings[offerID].deadline = block.number + _offerWindow;
    }

    function buyOfferedTicket(uint16 _eventID, uint16 _ticketID,
address _newAttendee) payable{
        bytes32 offerID = sha3(_eventID+_ticketID);
        if (msg.value > offerings[offerID].price &&
            block.number < offerings[offerID].deadline &&
            (msg.sender == offerings[offerID].buyer ||
             offerings[offerID].buyer == 0)) {

            events[_eventID].attendees[_ticketID]
                .send(offerings[offerID].price);


            events[_eventID].attendees[_ticketID] = _newAttendee;
            delete offerings[offerID];
        }
    }
}
```

Problem 6:

Ethereum lottery. Suppose the state of California decides to implement its lottery system as an Ethereum contract. The contract should support the following methods:

- buyTicket: any user can send the price of a ticket in Ether to the contract and the contract will record that user's address.
- doLottery: this method is called by the state lottery system once a week to randomly select that week's winner, if any. If there is a winner, the contract sends 90% of the pot to the winner's address and the remaining 10% rolls over to the following week. If there is no winner, the entire pot rolls over to the following week. Either way, the set of users resets to the empty set.

The contract proceeds in epochs, where each epoch is seven days, starting from the moment that the lottery contract is created. Say n users participate in a particular epoch. Each user is assigned an ID between 0 and $n-1$ in sequential order.

The doLottery method can only be called by the state of California within a 10 minute window after the end of each epoch. The method selects a winner by computing the current block hash modulo $2n$, and if the number matches a user ID, that user is the winner. Otherwise there is no winner, which happens with probability 1/2. The block hash modulo $2n$ can be computed as `(blockhash(block.number)%(2*n))`.

A) Write the solidity code to implement this contract. Use the back of this sheet.

B) Is this a good idea? Are there parties that can manipulate the lottery to greatly increase their chances of winning? If so explain how, if not explain why not.

Problem 7. Bob posts the following wallet contract to Ethereum to manage his personal finances:

A)

```
contract BobWallet {
    function pay(address dest, uint amount) {
       if (tx.origin == HardcodedBobAddress) dest.send(amount);
} }
```

The function `pay` lets Bob send funds to anyone he wants. Suppose Mallory can trick Bob into calling a method on a contract she controls. Explain how Mallory can transfer all the funds out of Bob's wallet into her own account.

Hint: Make sure you understand the semantics of `tx.origin`.

B) Consider the following Solidity code:

```
pragma solidity ^0.8.0;
contract ERC20 is IERC20 {
  mapping(address => uint256) private _balances;
  event Transfer(address indexed from, address indexed to, uint256 value);
  function _transfer(address sender, address recipient, uint256 amount) {
    emit Transfer(sender, recipient, amount);
}}
```

Suppose this code is deployed in two contracts: a contract at address X and a contract at address Y. Which of the following can read the state of `_balances` in contract X? Circle the correct answer(s).

A Code in the `_transfer()` function in contract `ERC20` at address X

B Code in the `_transfer()` function in contract `ERC20` at address Y

C An enduser using `etherscan.io`

C) Continuing with part (B), which of the following can read the log entry `Transfer` emitted when the function `_transfer()` is called? Circle the correct answer(s).

A Code in the function `getBalance()` defined in contract `ERC20` at address X

B Code in the function `getBalance()` defined in contract `ERC20` at address Y

C An enduser using `etherscan.io`

D) Consider the following Solidity function contained in a Ethereum contract:

```
function foo(uint256 x, uint256 y) {
    while (uint256(sha3(x)) != y)
        x++;
}
```

Using an x and a y for which this function never terminates, explain how a malicious miner could use this function to waste other miners' resources, without spending any of its own resources.

Reference: All these problems come from Stanford's CS251 Cryptocurrency course.