Smart Contract Security Audit Report

The SlowMist Security Team received the APENFT team's application for smart contract security audit of the APENFT on May 21, 2021. The following are the details and results of this smart contract security audit:

Token name :

APENFT

The Contract address :

TFczxzPhnThNSqr5by8tvxsdCFRRz6cPNq

Link address :

https://tronscan.org/#/contract/TFczxzPhnThNSqr5by8tvxsdCFRRz6cPNq/code

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive authority audit	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False top-up” vulnerability Audit	-	Passed

9	Malicious Event Log Audit	-	Passed
10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : Passed

Audit Number : 0X002105210003

Audit Date : May 21, 2021

Audit Team : SlowMist Security Team

Summary: This is a token contract that contains the timelock section. The total amount of tokens in the contract can be changed, and the owner can burn his own tokens after a time delay through the requestRedeem function. SafeMath security module is used, which is a recommend approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following issues:

1. In the StandardTokenWithFees contract, since the basisPointsRate and maximumFee parameters are both 0 and they can never be changed, the fee calculation module is redundant. It is recommended to remove this redundant module.

2. The owner can upgrade the contract through the deprecate function.

3. The Owner role can mint tokens arbitrarily after a time delay through the requestIssue function.

The source code:

SafeMath.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.18;



/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */

//SlowMist// SafeMath security module is used, which is a recommend approach

library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;

    assert(c / a == b); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {

    assert(b <= a); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;

    assert(c >= a); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas
    return c;
  }
}
```

Ownable.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.18;



/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;



  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);



  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  function Ownable() public {
    owner = msg.sender;
  }



  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }
  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0)); //SlowMist// This check is quite good in avoiding losing control of the
contract caused by user mistakes
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }

}
```

Pausable.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.4.18;


import "./Ownable.sol";


/**
 * @title Pausable
 * @dev Base contract which allows children to implement an emergency stop mechanism.
 */
contract Pausable is Ownable {
  event Pause();
  event Unpause();

  bool public paused = false;


  /**
   * @dev Modifier to make a function callable only when the contract is not paused.
   */
  modifier whenNotPaused() {
    require(!paused);
    _;
  }

  /**
   * @dev Modifier to make a function callable only when the contract is paused.
   */
  modifier whenPaused() {
    require(paused);
    _;
  }

  /**
   * @dev called by the owner to pause, triggers stopped state
   */
  //SlowMist// Suspending all transactions upon major abnormalities is a recommended approach
  function pause() onlyOwner whenNotPaused public {
    paused = true;
    Pause();
  }

  /**
   * @dev called by the owner to unpause, returns to normal state
   */
  function unpause() onlyOwner whenPaused public {
    paused = false;
    Unpause();
  }
}
```

BasicToken.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.18;



import './SafeMath.sol';

/**
 * @title TRC20Basic
 * @dev Simpler version of TRC20 interface
 */
contract TRC20Basic {
  function totalSupply() public constant returns (uint);
  function balanceOf(address who) public view returns (uint256);
  function transfer(address to, uint256 value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}

/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is TRC20Basic {
  using SafeMath for uint256;

  mapping(address => uint256) balances;

  /**
  * @dev transfer token for a specified address
  * @param _to The address to transfer to.
  * @param _value The amount to be transferred.
  */
  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake leading

to the loss of token during transfer
    require(_value <= balances[msg.sender]);

    // SafeMath.sub will throw if there is not enough balance.
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    Transfer(msg.sender, _to, _value);
    return true; //SlowMist// The return value conforms to the TIP20 specification
  }

  /**
  * @dev Gets the balance of the specified address.
  * @param _owner The address to query the the balance of.
  * @return An uint256 representing the amount owned by the passed address.
  */
  function balanceOf(address _owner) public view returns (uint256 balance) {
    return balances[_owner];
  }

}
```

StandardToken.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.4.18;


import './BasicToken.sol';


/**
 * @title TRC20 interface
 */
contract TRC20 is TRC20Basic {
  function allowance(address owner, address spender) public view returns (uint256);
  function transferFrom(address from, address to, uint256 value) public returns (bool);
  function approve(address spender, uint256 value) public returns (bool);
  event Approval(address indexed owner, address indexed spender, uint256 value);
}

/**
 * @title Standard TRC20 token
 *
 * @dev Implementation of the basic standard token.
 */
contract StandardToken is TRC20, BasicToken {

  mapping (address => mapping (address => uint256)) internal allowed;


  /**
   * @dev Transfer tokens from one address to another
   * @param _from address The address which you want to send tokens from
   * @param _to address The address which you want to transfer to
   * @param _value uint256 the amount of tokens to be transferred
   */
  function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
    require(_to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    Transfer(_from, _to, _value);
    return true; //SlowMist// The return value conforms to the TIP20 specification
  }

  /**
   * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
   *
   * Beware that changing an allowance with this method brings the risk that someone may use both the old
   * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
   * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
   * @param _spender The address which will spend the funds.
   * @param _value The amount of tokens to be spent.
   */
  function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true; //SlowMist// The return value conforms to the TIP20 specification
  }

  /**
   * @dev Function to check the amount of tokens that an owner allowed to a spender.
   * @param _owner address The address which owns the funds.
   * @param _spender address The address which will spend the funds.
   * @return A uint256 specifying the amount of tokens still available for the spender.
   */
  function allowance(address _owner, address _spender) public view returns (uint256) {
    return allowed[_owner][_spender];
  }

  /**
   * approve should be called when allowed[_spender] == 0. To increment
   * allowed value is better to use this function to avoid 2 calls (and wait until
   * the first transaction is mined)
   * From MonolithDAO Token.sol
   */
  function increaseApproval(address _spender, uint _addedValue) public returns (bool) {
    allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }

  function decreaseApproval(address _spender, uint _subtractedValue) public returns (bool) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
      allowed[msg.sender][_spender] = 0;
    } else {
      allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }

}
```

StandardTokenWithFees.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.18;

import "./StandardToken.sol";
import "./Ownable.sol";

contract StandardTokenWithFees is StandardToken, Ownable {

  // Additional variables for use if transaction fees ever became necessary
  uint256 public basisPointsRate = 0;
  uint256 public maximumFee = 0;
  uint256 constant MAX_SETTABLE_BASIS_POINTS = 20;
  uint256 constant MAX_SETTABLE_FEE = 50;

  string public name;
  string public symbol;
  uint8 public decimals;
  uint public _totalSupply;

  uint public constant MAX_UINT = 2**256 - 1;
  //SlowMist// Since the basisPointsRate and maximumFee parameters are both 0 and they can
never be changed, the fee calculation module is redundant. It is recommended to remove this
redundant module
  function calcFee(uint _value) constant returns (uint) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    return fee;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    uint fee = calcFee(_value);
    uint sendAmount = _value.sub(fee);

    super.transfer(_to, sendAmount);
    if (fee > 0) {
      super.transfer(owner, fee);
    }
    return true; //SlowMist// The return value conforms to the TIP20 specification
  }

  function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
    require(_to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    uint fee = calcFee(_value);
    uint sendAmount = _value.sub(fee);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (allowed[_from][msg.sender] < MAX_UINT) {
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    }
    Transfer(_from, _to, sendAmount);
    if (fee > 0) {
      balances[owner] = balances[owner].add(fee);
      Transfer(_from, owner, fee);
    }
    return true; //SlowMist// The return value conforms to the TIP20 specification
  }

}
```

TimelockToken.sol:

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.4.18;

import "./StandardTokenWithFees.sol";

/**
 * @dev Contract module which acts as a timelocked Token. When set as the
 * owner of an `Ownable` smart contract, it enforces a timelock on all
 * `onlyOwner` maintenance operations. This gives time for users of the
 * controlled contract to exit before a potentially dangerous maintenance
 * operation is applied.
 *
 * This contract is a modified version of:
 * https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/governance/TimelockController.sol
 *
 */
contract TimelockToken is StandardTokenWithFees{
    uint256 internal constant _DONE_TIMESTAMP = uint256(1);

    mapping(uint256 => action) public actions;

    uint256 private _minDelay = 3 days;

    uint256 public nonce;

    enum RequestType{
        Issue,
        Redeem
    }

    struct action {
        uint256 timestamp;
        RequestType requestType;
        uint256 value;
    }
    /**
     * @dev Emitted when a call is scheduled as part of operation `id`.
     */
    event RequestScheduled(uint256 indexed id, RequestType _type, uint256 value,  uint256 availableTime);



    /**
     * @dev Emitted when a call is performed as part of operation `id`.
     */
    event RequestExecuted(uint256 indexed id, RequestType _type, uint256 value);



    // Called when new token are issued
    event Issue(uint amount);

    // Called when tokens are redeemed
```

```
event Redeem(uint amount);

/**
 * @dev Emitted when operation `id` is cancelled.
 */
event Cancelled(uint256 indexed id);

/**
 * @dev Emitted when the minimum delay for future operations is modified.
 */
event DelayTimeChange(uint256 oldDuration, uint256 newDuration);


/**
 * @dev Initializes the contract with a given `minDelay`.
 */
constructor() public {
    emit DelayTimeChange(0, 3 days);
}

/**
 * @dev Returns whether an id correspond to a registered operation. This
 * includes both Pending, Ready and Done operations.
 */
function isOperation(uint256 id) public view returns (bool registered) {
    return getTimestamp(id) > 0;
}

/**
 * @dev Returns whether an operation is pending or not.
 */
function isOperationPending(uint256 id) public view returns (bool pending) {
    return getTimestamp(id) > _DONE_TIMESTAMP;
}

/**
 * @dev Returns whether an operation is ready or not.
 */
function isOperationReady(uint256 id) public view returns (bool ready) {
    uint256 timestamp = getTimestamp(id);
```

```
        // solhint-disable-next-line not-rely-on-time
        return timestamp > _DONE_TIMESTAMP && timestamp <= block.timestamp;
    }

    /**
     * @dev Returns whether an operation is done or not.
     */
    function isOperationDone(uint256 id) public view returns (bool done) {
        return getTimestamp(id) == _DONE_TIMESTAMP;
    }

    /**
     * @dev Returns the timestamp at with an operation becomes ready (0 for
     * unset operations, 1 for done operations).
     */
    function getTimestamp(uint256 id) public view returns (uint256 timestamp) {
        return actions[id].timestamp;
    }

    /**
     * @dev Returns the minimum delay for an operation to become valid.
     *
     * This value can be changed by executing an operation that calls `updateDelay`.
     */
    function getMinDelay() public view  returns (uint256 duration) {
        return _minDelay;
    }

    /**
     * @dev Schedule an operation.
     *
     * Emits a {RequestScheduled} event.
     *
     */
    function _request(RequestType _requestType, uint256 value) private {
        uint256 id = nonce;
        nonce ++;
        _schedule(id, _requestType, value,  _minDelay);
    }
    /**
     * @dev Schedule an operation that is to becomes valid after a given delay.
     */
    function _schedule(uint256 id, RequestType _type, uint256 value, uint256 delay) private {
        require(!isOperation(id), "TimelockToken: operation already scheduled");
        require(delay >= getMinDelay(), "TimelockToken: insufficient delay");
        // solhint-disable-next-line not-rely-on-time
        uint256 availableTime = block.timestamp + delay;
        actions[id].timestamp = availableTime;
        actions[id].requestType = _type;
        actions[id].value = value;
        emit RequestScheduled(id, _type, value,  availableTime);
    }

    /**
     * @dev Cancel an operation.
     *
     * Requirements:
     *
     * - the caller must have the 'owner' role.
     */
    function cancel(uint256 id) public onlyOwner {
        require(isOperationPending(id), "TimelockToken: operation cannot be cancelled");
        delete actions[id];
        emit Cancelled(id);
    }


    /**
     * @dev Checks before execution of an operation's calls.
     */
    function _beforeCall(uint256 id) private {
        require(isOperation(id), "TimelockToken: operation is not registered"); //SlowMist// It is recommended to use
isOperationPending to check id to save Gas
    }

    /**
     * @dev Checks after execution of an operation's calls.
     */
    function _afterCall(uint256 id) private {
        require(isOperationReady(id), "TimelockToken: operation is not ready");
        actions[id].timestamp = _DONE_TIMESTAMP;
    }



    /**
     * @dev Execute an operation's call.
     */
    function _call(uint256 id, address owner) private {
        uint256 amount = actions[id].value;
        // solhint-disable-next-line avoid-low-level-calls
        if(actions[id].requestType == RequestType.Issue) {
            balances[owner] = balances[owner].add(amount);
            _totalSupply = _totalSupply.add(amount);
            emit Transfer(address(0), owner, amount);
            emit Issue(amount);
        }
        else if(actions[id].requestType == RequestType.Redeem) {
            _totalSupply = _totalSupply.sub(amount);
            balances[owner] = balances[owner].sub(amount);
            emit Transfer(owner, address(0), amount);
            emit Redeem(amount);
        }
    }

    /*
     * Schedule to issue a new amount of tokens
     * these tokens are deposited into the owner address
     *
     * @param _amount Number of tokens to be issued
     * Requirements:
     *
     * - the caller must have the 'owner' role.
     */
```

//SlowMist// The Owner role can mint tokens arbitrarily after a time delay through the requestIssue function

```
    function requestIssue(uint256 amount) public onlyOwner {
        _request(RequestType.Issue, amount);
    }

    /*
     * Schedule to redeem a new amount of tokens
     * these tokens are deposited into the owner address
     *
     * @param _amount Number of tokens to be redeemed
     * Requirements:
     *
     * - the caller must have the 'owner' role.
     */
    function requestRedeem(uint256 amount) public onlyOwner {
        _request(RequestType.Redeem, amount);
    }

    /*
     * execute   a request
     *
     * @param id the target action id of the request
     * Requirements:
     *
     * - the caller must have the 'owner' role.
     */

    function executeRequest(uint256 id) public onlyOwner {
        _beforeCall(id);
        _call(id, msg.sender);
        _afterCall(id);
    }

}
```

APENFT.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.18;


import "./TimelockToken.sol";
import "./Pausable.sol";
```

```
contract UpgradedStandardToken is StandardToken {
    // those methods are called by the legacy contract
    // and they must ensure msg.sender to be the contract address
    uint public _totalSupply;
    function transferByLegacy(address from, address to, uint value) public returns (bool);
    function transferFromByLegacy(address sender, address from, address spender, uint value) public returns (bool);
    function approveByLegacy(address from, address spender, uint value) public returns (bool);
    function increaseApprovalByLegacy(address from, address spender, uint addedValue) public returns (bool);
    function decreaseApprovalByLegacy(address from, address spender, uint subtractedValue) public returns (bool);
}

contract APENFT is Pausable, TimelockToken{

    address public upgradedAddress;
    bool public deprecated;

    //  The contract can be initialized with a number of tokens
    //  All the tokens are deposited to the owner address
    function APENFT() public {
        _totalSupply = 999990000000000000000;
        name = "APENFT";
        symbol = "NFT";
        decimals = 6;
        balances[owner] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
        deprecated = false;
    }

    // Forward TRC20 methods to upgraded contract if this one is deprecated
    function transfer(address _to, uint _value) public whenNotPaused returns (bool) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).transferByLegacy(msg.sender, _to, _value);
        } else {
            return super.transfer(_to, _value);
        }
    }

    // Forward TRC20 methods to upgraded contract if this one is deprecated
    function transferFrom(address _from, address _to, uint _value) public whenNotPaused returns (bool) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).transferFromByLegacy(msg.sender, _from, _to, _value);
        } else {
            return super.transferFrom(_from, _to, _value);
        }
    }

    // Forward TRC20 methods to upgraded contract if this one is deprecated
    function balanceOf(address who) public constant returns (uint) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).balanceOf(who);
        } else {
            return super.balanceOf(who);
        }
    }

    // Allow checks of balance at time of deprecation
    function oldBalanceOf(address who) public constant returns (uint) {
        if (deprecated) {
            return super.balanceOf(who);
        }
    }

    // Forward TRC20 methods to upgraded contract if this one is deprecated
    function approve(address _spender, uint _value) public whenNotPaused returns (bool) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).approveByLegacy(msg.sender, _spender, _value);
        } else {
            return super.approve(_spender, _value);
        }
    }

    function increaseApproval(address _spender, uint _addedValue) public whenNotPaused returns (bool) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).increaseApprovalByLegacy(msg.sender, _spender,
_addedValue);
        } else {
            return super.increaseApproval(_spender, _addedValue);
        }
    }
    function decreaseApproval(address _spender, uint _subtractedValue) public whenNotPaused returns (bool) {
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).decreaseApprovalByLegacy(msg.sender, _spender,
_subtractedValue);
        } else {
            return super.decreaseApproval(_spender, _subtractedValue);
        }
    }

    // Forward TRC20 methods to upgraded contract if this one is deprecated
    function allowance(address _owner, address _spender) public constant returns (uint remaining) {
        if (deprecated) {
            return StandardToken(upgradedAddress).allowance(_owner, _spender);
        } else {
            return super.allowance(_owner, _spender);
        }
    }

    // deprecate current contract in favour of a new one
```

//SlowMist// The owner can upgrade the contract through the deprecate function

```
    function deprecate(address _upgradedAddress) public onlyOwner {
        require(_upgradedAddress != address(0));
        deprecated = true;
        upgradedAddress = _upgradedAddress;
        Deprecate(_upgradedAddress);
    }

    // deprecate current contract if favour of a new one
    function totalSupply() public constant returns (uint) {
        if (deprecated) {
            return StandardToken(upgradedAddress).totalSupply();
        } else {
            return _totalSupply;
        }
    }



    // Called when contract is deprecated
    event Deprecate(address newAddress);

}
```

MultiSigWallet.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.10; //SlowMist// The compiler version is too low, it is suggested to use a higher

version for deployment


/// @title Multisignature wallet - Allows multiple parties to agree on transactions before execution.
/// @author Stefan George - <stefan.george@consensys.net>
contract MultiSigWallet {

    uint constant public MAX_OWNER_COUNT = 50;

    event Confirmation(address indexed sender, uint indexed transactionId);
    event Revocation(address indexed sender, uint indexed transactionId);
    event Submission(uint indexed transactionId);
    event Execution(uint indexed transactionId);
    event ExecutionFailure(uint indexed transactionId);
    event Deposit(address indexed sender, uint value);
    event OwnerAddition(address indexed owner);
    event OwnerRemoval(address indexed owner);
    event RequirementChange(uint required);

    mapping (uint => Transaction) public transactions;
    mapping (uint => mapping (address => bool)) public confirmations;
    mapping (address => bool) public isOwner;
    address[] public owners;
    uint public required;
    uint public transactionCount;

    struct Transaction {
        address destination;
        uint value;
        bytes data;
        bool executed;
    }

    modifier onlyWallet() {
        if (msg.sender != address(this))
            throw;
        _;
    }

    modifier ownerDoesNotExist(address owner) {
        if (isOwner[owner])
            throw;
        _;
    }

    modifier ownerExists(address owner) {
        if (!isOwner[owner])
            throw;
        _;
    }

    modifier transactionExists(uint transactionId) {
        if (transactions[transactionId].destination == 0)
            throw;
        _;
    }

    modifier confirmed(uint transactionId, address owner) {
        if (!confirmations[transactionId][owner])
            throw;
        _;
    }

    modifier notConfirmed(uint transactionId, address owner) {
        if (confirmations[transactionId][owner])
            throw;
        _;
    }

    modifier notExecuted(uint transactionId) {
        if (transactions[transactionId].executed)
```

```
            throw;
        _;
    }

    modifier notNull(address _address) {
        if (_address == 0)
            throw;
        _;
    }

    modifier validRequirement(uint ownerCount, uint _required) {
        if (    ownerCount > MAX_OWNER_COUNT
            || _required > ownerCount
            || _required == 0
            || ownerCount == 0)
            throw;
        _;
    }

    /// @dev Fallback function allows to deposit ether.
    function()
        payable
    {
        if (msg.value > 0)
            Deposit(msg.sender, msg.value);
    }

    /*
     * Public functions
     */
    /// @dev Contract constructor sets initial owners and required number of confirmations.
    /// @param _owners List of initial owners.
    /// @param _required Number of required confirmations.
    function MultiSigWallet(address[] _owners, uint _required)
        public
        validRequirement(_owners.length, _required)
    {
        for (uint i=0; i<_owners.length; i++) {
            if (isOwner[_owners[i]] || _owners[i] == 0)
                throw;
            isOwner[_owners[i]] = true;
        }
        owners = _owners;
        required = _required;
    }

    /// @dev Allows to add a new owner. Transaction has to be sent by wallet.
    /// @param owner Address of new owner.
    function addOwner(address owner)
        public
        onlyWallet
        ownerDoesNotExist(owner)
        notNull(owner)
        validRequirement(owners.length + 1, required)
    {
        isOwner[owner] = true;
        owners.push(owner);
        OwnerAddition(owner);
    }

    /// @dev Allows to remove an owner. Transaction has to be sent by wallet.
    /// @param owner Address of owner.
    function removeOwner(address owner)
        public
        onlyWallet
        ownerExists(owner)
    {
        isOwner[owner] = false;
        for (uint i=0; i<owners.length - 1; i++)
            if (owners[i] == owner) {
                owners[i] = owners[owners.length - 1];
                break;
            }
        owners.length -= 1;
        if (required > owners.length)
            changeRequirement(owners.length);
        OwnerRemoval(owner);
    }

    /// @dev Allows to replace an owner with a new owner. Transaction has to be sent by wallet.
```

```
/// @param owner Address of owner to be replaced.
/// @param owner Address of new owner.
function replaceOwner(address owner, address newOwner)
    public
    onlyWallet
    ownerExists(owner)
    ownerDoesNotExist(newOwner)
{
    for (uint i=0; i<owners.length; i++)
        if (owners[i] == owner) {
            owners[i] = newOwner;
            break;
        }
    isOwner[owner] = false;
    isOwner[newOwner] = true;
    OwnerRemoval(owner);
    OwnerAddition(newOwner);
}

/// @dev Allows to change the number of required confirmations. Transaction has to be sent by wallet.
/// @param _required Number of required confirmations.
function changeRequirement(uint _required)
    public
    onlyWallet
    validRequirement(owners.length, _required)
{
    required = _required;
    RequirementChange(_required);
}

/// @dev Allows an owner to submit and confirm a transaction.
/// @param destination Transaction target address.
/// @param value Transaction ether value.
/// @param data Transaction data payload.
/// @return Returns transaction ID.
function submitTransaction(address destination, uint value, bytes data)
    public
    returns (uint transactionId)
{
    transactionId = addTransaction(destination, value, data);
```

```
        confirmTransaction(transactionId);
    }

    /// @dev Allows an owner to confirm a transaction.
    /// @param transactionId Transaction ID.
    function confirmTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        transactionExists(transactionId)
        notConfirmed(transactionId, msg.sender)
    {
        confirmations[transactionId][msg.sender] = true;
        Confirmation(msg.sender, transactionId);
        executeTransaction(transactionId);
    }

    /// @dev Allows an owner to revoke a confirmation for a transaction.
    /// @param transactionId Transaction ID.
    function revokeConfirmation(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
    {
        confirmations[transactionId][msg.sender] = false;
        Revocation(msg.sender, transactionId);
    }

    /// @dev Allows anyone to execute a confirmed transaction.
    /// @param transactionId Transaction ID.
    //SlowMist// Any user can execute a confirmed transaction through the executeTransaction
function
    function executeTransaction(uint transactionId)
        public
        notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction tx = transactions[transactionId];
            tx.executed = true;
            if (tx.destination.call.value(tx.value)(tx.data))
                Execution(transactionId);
            else {
                ExecutionFailure(transactionId);
                tx.executed = false;
            }
        }
    }

    /// @dev Returns the confirmation status of a transaction.
    /// @param transactionId Transaction ID.
    /// @return Confirmation status.
    function isConfirmed(uint transactionId)
        public
        constant
        returns (bool)
    {
        uint count = 0;
        for (uint i=0; i<owners.length; i++) {
            if (confirmations[transactionId][owners[i]])
                count += 1;
            if (count == required)
                return true;
        }
    }

    /*
     * Internal functions
     */
    /// @dev Adds a new transaction to the transaction mapping, if transaction does not exist yet.
    /// @param destination Transaction target address.
    /// @param value Transaction ether value.
    /// @param data Transaction data payload.
    /// @return Returns transaction ID.
    function addTransaction(address destination, uint value, bytes data)
        internal
        notNull(destination)
        returns (uint transactionId)
    {
        transactionId = transactionCount;
        transactions[transactionId] = Transaction({
            destination: destination,
            value: value,
            data: data,
            executed: false
        });
        transactionCount += 1;
        Submission(transactionId);
    }

    /*
     * Web3 call functions
     */
    /// @dev Returns number of confirmations of a transaction.
    /// @param transactionId Transaction ID.
    /// @return Number of confirmations.
    function getConfirmationCount(uint transactionId)
        public
        constant
        returns (uint count)
    {
        for (uint i=0; i<owners.length; i++)
            if (confirmations[transactionId][owners[i]])
                count += 1;
    }

    /// @dev Returns total number of transactions after filers are applied.
    /// @param pending Include pending transactions.
    /// @param executed Include executed transactions.
    /// @return Total number of transactions after filters are applied.
    function getTransactionCount(bool pending, bool executed)
        public
        constant
        returns (uint count)
    {
        for (uint i=0; i<transactionCount; i++)
            if (   pending && !transactions[i].executed
                || executed && transactions[i].executed)
                count += 1;
    }
```

```
/// @dev Returns list of owners.
/// @return List of owner addresses.
function getOwners()
    public
    constant
    returns (address[])
{
    return owners;
}

/// @dev Returns array with owner addresses, which confirmed transaction.
/// @param transactionId Transaction ID.
/// @return Returns array of owner addresses.
function getConfirmations(uint transactionId)
    public
    constant
    returns (address[] _confirmations)
{
    address[] memory confirmationsTemp = new address[](owners.length);
    uint count = 0;
    uint i;
    for (i=0; i<owners.length; i++)
        if (confirmations[transactionId][owners[i]]) {
            confirmationsTemp[count] = owners[i];
            count += 1;
        }
    _confirmations = new address[](count);
    for (i=0; i<count; i++)
        _confirmations[i] = confirmationsTemp[i];
}

/// @dev Returns list of transaction IDs in defined range.
/// @param from Index start position of transaction array.
/// @param to Index end position of transaction array.
/// @param pending Include pending transactions.
/// @param executed Include executed transactions.
/// @return Returns array of transaction IDs.
function getTransactionIds(uint from, uint to, bool pending, bool executed)
    public
```

```
        constant
        returns (uint[] _transactionIds)
    {
        uint[] memory transactionIdsTemp = new uint[](transactionCount);
        uint count = 0;
        uint i;
        for (i=0; i<transactionCount; i++)
            if (    pending && !transactions[i].executed
                || executed && transactions[i].executed)
            {
                transactionIdsTemp[count] = i;
                count += 1;
            }
        _transactionIds = new uint[](to - from);
        for (i=from; i<to; i++)
            _transactionIds[i - from] = transactionIdsTemp[i];
    }
}
```

Migrations.sol:

```
//SlowMist// This is a redundant contract, it is recommended to remove unused contracts

pragma solidity ^0.4.4;
/* solhint-disable var-name-mixedcase */



contract Migrations {
    address public owner;
    uint public last_completed_migration;

    modifier restricted() {
        if (msg.sender == owner) _;
    }

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

    function setCompleted(uint completed) public restricted {
        last_completed_migration = completed;
    }


    function upgrade(address newAddress) public restricted {
        Migrations upgraded = Migrations(newAddress);
        upgraded.setCompleted(last_completed_migration);
    }
}
```