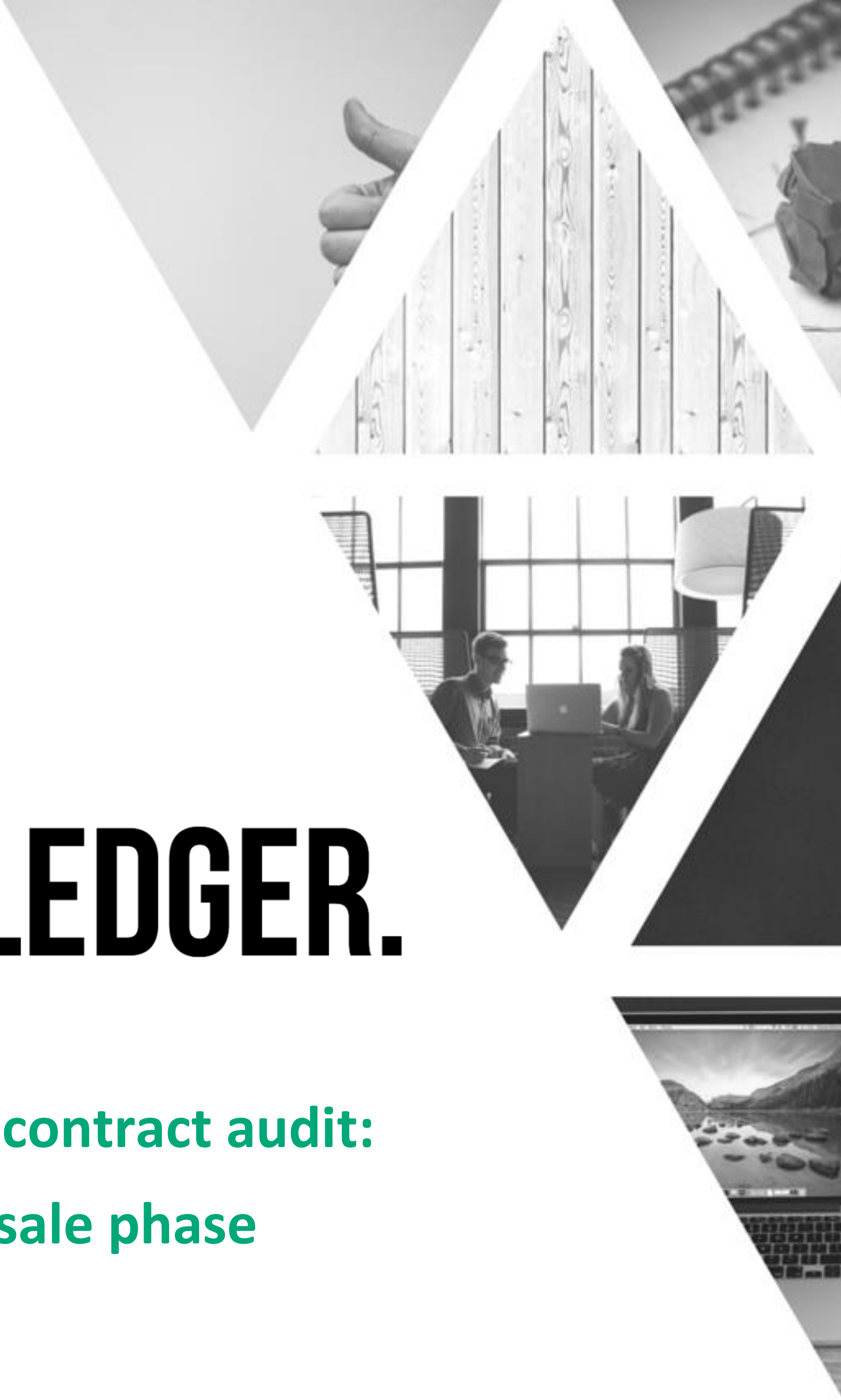

THELEDGER.

Identify contract audit: Presale phase

CONTACT INFO

TheLedger NV
Business Park King Square
Veldkant 33B
2550 Kontich
Belgium
Phone: +32 (0)3 871 99 67
VAT: BE0671.584.745

Contacts

Filip Francken
Co-Founder – TheLedger
filip.francken@theledger.be
+32 490 66 07 49

Andries Van Humbeeck
Co-Founder – TheLedger
andries.vanhumbeeck@theledger.be
+32 498 77 70 41

Kevin Leyssens
Blockchain Bandit – TheLedger
kevin.leyssens@theledger.be
+32 491 07 79 58

“If everyone is moving forward together, then success takes care of itself”

—Henry Ford

INDEX

INTRODUCTION

This document is the result of the audit of the presale phase done by TheLedger.

At the start of the audit, we did an analysis of all the written functions and contracts.

Afterwards we started writing tests to check for bugs and inconsistencies. These tests are summed up under each function and the total of tests are summed up under "Tests".

Then we brainstormed on how to attack the contracts through a malicious contract. Trying to break the code, extract tokens, invoking malicious transactions, etc. We wrote down our use case on the malicious code section.

Afterwards we ran the code through multiple open source tools to do a static analysis of the code. Testing for well-known vulnerabilities described under static analysis.

Then we manually checked for the vulnerabilities which are described in tested vulnerabilities.

At the end we summed up 3 worst case scenarios.

Our conclusion louds that the codebase is production ready, even though we are aware a possible integer underflow and reentrancy is possible at the "approveAndCall" method in the CustomToken.sol smart contract. But this is not one of the critical kind.

Note that it is recommended to audit your codebase by more than one audit party.

WHITELIST.SOL

MODIFIERS

NOTPAUSED

```
modifier notPaused() {
        require(!paused);
        _;
    }
```

Tests:

- Should not add when paused

ONLYADMIN

```
modifier onlyAdmin() {
        require(isAdmin[msg.sender] || msg.sender == owner);
        _;
    }
```

Tests:

- Should not add other successfully to list when not an admin

FUNCTIONS

FALLBACK

```
function () payable public {
            msg.sender.transfer(msg.value);
    }
```

Tests:

- Should transfer ether back

CONSTRUCTOR

```
function Whitelist() public {
        require(addAdmin(msg.sender));
    }
```

Tests:

- Should be able to use the constructor

ISPARTICIPANT

```
function isParticipant(address _participant) public view returns
(bool) {
        require(address(_participant) != 0);
        return isParticipant[_participant];
    }
```

Tests:

- Should add 3 addresses to the whitelist
- Should add 5 addresses to the whitelist
- Should add 10 addresses to the whitelist

ADDPARTICIPANT

```
function addParticipant(address _participant) public notPaused
onlyAdmin returns (bool) {
        require(address(_participant) != 0);
        require(isParticipant[_participant] == false);

        isParticipant[_participant] = true;
        participantAmount++;
        AddParticipant(_participant);
        return true;
    }
```

Tests:

- Should add other successfully to list
- Should not add other successfully to list when not an address
- Should not add other successfully to list when not an admin
- Should added only once
- Should not add when paused
- Should resume Whitelist
- Should remove an existing account and update the count. After add this one again

REMOVEPARTICIPANT

```
function removeParticipant(address _participant) public onlyAdmin
returns (bool) {
        require(address(_participant) != 0);
        require(isParticipant[_participant]);
        require(msg.sender != _participant);

        delete isParticipant[_participant];
        participantAmount--;
        RemoveParticipant(_participant);
        return true;
    }
```

Tests:

- Should remove an existing account and update the count. After add this one again
- Should not remove self
- Should not remove other from list when not an address

ADDADMIN

```
function addAdmin(address _admin) public onlyAdmin returns
(bool) {
        require(address(_admin) != 0);
        require(!isAdmin[_admin]);

        isAdmin[_admin] = true;
        AddAdmin(_admin, now);
        return true;
    }
```

Tests:

- Should not be able to add admins when not an admin
- Admins can add admins
- Should remove admin

REMOVEADMIN

```
function removeAdmin(address _admin) public onlyAdmin returns
(bool) {
        require(address(_admin) != 0);
        require(isAdmin[_admin]);
        require(msg.sender != _admin);

        delete isAdmin[_admin];
        return true;
    }
```

Tests:

- Should remove admin
- Should not be able to remove self as admin

PAUSEWHITELIST

```
function pauseWhitelist() public onlyAdmin returns (bool) {
        paused = true;
        Paused(msg.sender,now);
        return true;
    }
```

Tests:

- Should not add when paused

RESUMEWHITELIST

```
function resumeWhitelist() public onlyAdmin returns (bool) {
        paused = false;
        Resumed(msg.sender,now);
        return true;
    }
```

Tests:

- Should resume Whitelist

ADDMULTIPLEPARTICIPANTS

```
function addMultipleParticipants(address[] _participants ) public
onlyAdmin returns (bool) {

        for ( uint i = 0; i < _participants.length; i++ ) {
            require(addParticipant(_participants[i]));
        }

        return true;
    }
```

Tests:

- Should add 3 addresses to the whitelist

ADDFIVEPARTICIPANTS

```
function addFiveParticipants(address participant1, address
participant2, address participant3, address participant4, address
participant5) public onlyAdmin returns (bool) {
        require(addParticipant(participant1));
        require(addParticipant(participant2));
        require(addParticipant(participant3));
        require(addParticipant(participant4));
        require(addParticipant(participant5));
        return true;
    }
```

Tests:

- Should add 5 addresses to the whitelist
- Should fail when not inputting enough parameters

ADDTENPARTICIPANTS

```
function addTenParticipants(address participant1, address
participant2, address participant3, address participant4, address
participant5,
     address participant6, address participant7, address
participant8, address participant9, address participant10) public
onlyAdmin returns (bool)
     {
        require(addParticipant(participant1));
        require(addParticipant(participant2));
        require(addParticipant(participant3));
        require(addParticipant(participant4));
        require(addParticipant(participant5));
        require(addParticipant(participant6));
        require(addParticipant(participant7));
        require(addParticipant(participant8));
        require(addParticipant(participant9));
        require(addParticipant(participant10));
        return true;
    }
```

Tests:

- Should add 10 addresses to the whitelist

CLAIMTOKENS

```
function claimTokens(address _claimtoken) onlyAdmin public
returns (bool) {
        if (_claimtoken == 0x0) {
            owner.transfer(this.balance);
            return true;
        }

        ERC20 claimtoken = ERC20(_claimtoken);
        uint balance = claimtoken.balanceOf(this);
        claimtoken.transfer(owner, balance);
        ClaimedTokens(_claimtoken, owner, balance);
        return true;
    }
```

TESTS

1. Should be able to use the constructor
2. Should add other successfully to list
3. Should not add other successfully to list when not an address
4. Should not add other successfully to list when not an admin
5. Should added only once
6. Should not add when paused
7. Should resume Whitelist
8. Should not be able to add admins when not an admin
9. Admins can add admins
10. Should remove admin
11. Should not be able to remove self as admin
12. Should remove an existing account and update the count. After add this one again
13. Should not remove self
14. Should not remove other from list when not an address
15. Should transfer ether back
16. Should add 3 addresses to the whitelist
17. Should add 5 addresses to the whitelist
18. Should add 10 addresses to the whitelist
19. Should fail when not inputting enough parameters

The total function coverage is 92 percent.

PRESALE.SOL

MODIFIERS

ISINWHITELIST

```
modifier isInWhitelist(address beneficiary) {
      require(whitelist.isParticipant(beneficiary));
    _;
  }
```

Tests:

- Should not be able to buy correct amount of tokens if not in whitelist
- Should be able to buy correct amount of tokens if in whitelist

WHENNOTPAUSED

```
modifier whenNotPaused() {
    require(!paused);
    _;
  }
```

Tests:

- Should not buy tokens when paused
- Should be able to resume Presale

WHENNOTFINALIZED

```
modifier whenNotFinalized() {
    require(!isFinalized);
    _;
  }
```

ONLYMULTISIGWALLET

```
modifier onlyMultisigWallet() {
    require(msg.sender == wallet);
    _;
  }
```

Tests:

- Should transfer ownership when invoking as multisigwallet
- Should not finalize presale when not multisig
- Should finalize presale when all requirements are met as multisigwallet + changed owner of token

FUNCTIONS

CONSTRUCTOR

```
function Presale(uint256 _startTime, address _wallet, address
_token, address _whitelist, uint256 _capETH, uint256
_capTokens, uint256 _minimumETH, uint256 _maximumETH)
public {

    require(_startTime >= now);
    require(_wallet != address(0));
    require(_token != address(0));
    require(_whitelist != address(0));
    require(_capETH > 0);
    require(_capTokens > 0);
    require(_minimumETH > 0);
    require(_maximumETH > 0);

    startTime = _startTime;
    endTime = _startTime.add(10 weeks);
    wallet = _wallet;
    tokenAddress = _token;
    token = Identify(_token);
    whitelist = Whitelist(_whitelist);
    capWEI = _capETH * (10 ** uint256(18));
    capTokens = _capTokens * (10 ** uint256(6));
    minimumWEI = _minimumETH * (10 ** uint256(18));
    maximumWEI = _maximumETH * (10 ** uint256(18));
  }
```

Tests:

- Should be able to use the constructor

FALLBACK

```
function () external payable {
   buyTokens(msg.sender);
  }
```

Tests:

- Should not be able to buy correct amount of tokens if not in whitesale
- Should not buy tokens if address is contract
- Should be able to buy correct amount of tokens if in whitesale
- Should not buy tokens when paused
- Should be able to resume Presale
- Should buy tokens to setup for further tests
- Should not buy tokens when not enough wei
- Should not buy tokens when to much wei
- Should not buy tokens over eth cap
- Should buy tokens to the ethercap
- Should not be able to buy if tokens go over cap
- Should drop minimumETH check when gap to cap is less than minimumETH

BUYTOKENS

```
function buyTokens(address beneficiary) isInWhitelist(beneficiary)
whenNotPaused whenNotFinalized public payable returns (bool) {
    require(beneficiary != address(0));
    require(validPurchase());
    require(!hasEnded());
    require(!isContract(msg.sender));

    uint256 weiAmount = msg.value;

       uint256 tokens = getTokenAmount(weiAmount);
    require(tokenRaised.add(tokens) <= capTokens);
       weiRaised = weiRaised.add(weiAmount);
    tokenRaised = tokenRaised.add(tokens);

    require(token.transferFrom(tokenAddress, beneficiary,
tokens));

    TokenPurchase(msg.sender, beneficiary, weiAmount, tokens);

    forwardFunds();
    return true;
  }
```

Tests:

- Should not be able to buy correct amount of tokens if not in whitesale
- Should not buy tokens if address is contract
- Should be able to buy correct amount of tokens if in whitesale
- Should not buy tokens when paused
- Should be able to resume Presale
- Should buy tokens to setup for further tests
- Should not buy tokens when not enough wei
- Should not buy tokens when to much wei
- Should not buy tokens over eth cap
- Should buy tokens to the ethercap
- Should not be able to buy if tokens go over cap
- Should drop minimumETH check when gap to cap is less than minimumETH

HASENDED

```
function hasEnded() public view returns (bool) {
    bool capReached = weiRaised >= capWEI;
    bool capTokensReached = tokenRaised >= capTokens;
    bool ended = now > endTime;
    return (capReached || capTokensReached) || ended;
  }
```

Tests:

- Should not finalize presale when not ended
- Should not finalize presale when not multisig
- Should finalize presale when all requirements are met as multisigwallet + changed owner of token
- Should drop minimumETH check when gap to cap is less than minimumETH

GETTOKENAMOUNT

```
function getTokenAmount(uint256 weiAmount) internal view
returns(uint256) {
      uint256 bonusIntegrated =
weiAmount.div(100000000000000).mul(rate).mul(bonusPercentag
e).div(100);
    return bonusIntegrated;
  }
```

Tests:

- Should be able to buy correct amount of tokens if in whitelist
- Should buy tokens to setup for further tests
- Should buy tokens to the ethercap

FORWARDFUNDS

```
function forwardFunds() internal returns (bool) {
    wallet.transfer(msg.value);
    return true;
  }
```

Tests:

- Should have 25 ETH in the wallet after a valid purchase of 25 ETH

VALIDPURCHASE

```
function validPurchase() internal view returns (bool) {
    bool withinPeriod = now >= startTime && now <= endTime;
    bool nonZeroPurchase = msg.value != 0;
    bool underMaximumWEI = msg.value <= maximumWEI;
    bool withinCap = weiRaised.add(msg.value) <= capWEI;
    bool minimumWEIReached;
       if ( capWEI.sub(weiRaised) < minimumWEI) {
      minimumWEIReached = true;
    } else {
      minimumWEIReached = msg.value >= minimumWEI;
    }
    return (withinPeriod && nonZeroPurchase) && (withinCap &&
(minimumWEIReached && underMaximumWEI));

  }
```

Tests:

- Should not buy tokens when not enough WEI
- Should not buy tokens when too much WEI
- Should not buy tokens over eth cap
- Should not be able to buy if tokens if over cap

TRANSFEROWNERSHIPTOKEN

```
function transferOwnershipToken(address newOwner)
onlyMultisigWallet public returns (bool) {
    require(token.transferOwnership(newOwner));
    return true;
  }
```

Tests:

- Should not transferownershiptoken when invoking as owner

TRANSFEROWNERSHIP

```
function transferOwnership(address newOwner) onlyMultisigWallet
public returns (bool) {
    require(newOwner != address(0));
    owner = newOwner;
    OwnershipTransferred(owner, newOwner);
    return true;
  }
```

Tests:

- Should not transfer ownership as owner

FINALIZE

```
function finalize() onlyMultisigWallet whenNotFinalized public
returns (bool) {
    require(hasEnded());

       if (!(capWEI == weiRaised)) {
           uint256 remainingTokens = capTokens.sub(tokenRaised);
           require(token.burn(tokenAddress, remainingTokens));
    }
    require(token.transferOwnership(wallet));
    isFinalized = true;
    return true;
  }
```

Tests:

- Should not finalize presale when not ended
- Should not finalize presale when not multisig
- Should finalize presale when all requirements are met as multisigwallet + changed owner of token

ISCONTRACT

```
function isContract(address _addr) constant internal returns
(bool) {
    if (_addr == 0) {
      return false;
    }
    uint256 size;
    assembly {
        size := extcodesize(_addr)
     }
    return (size > 0);
  }
```

CLAIMTOKENS

```
function claimTokens(address _claimtoken) onlyOwner public
returns (bool) {
    if (_claimtoken == 0x0) {
      owner.transfer(this.balance);
      return true;
    }

    ERC20 claimtoken = ERC20(_claimtoken);
    uint balance = claimtoken.balanceOf(this);
    claimtoken.transfer(owner, balance);
    ClaimedTokens(_claimtoken, owner, balance);
    return true;
  }
```

PAUSEPRESALE

```
function pausePresale() onlyOwner public returns (bool) {
    paused = true;
    Paused(owner, now);
    return true;
  }
```

Tests:

- Should not buy tokens when paused

RESUMEPRESALE

```
function resumePresale() onlyOwner public returns (bool) {
    paused = false;
    Resumed(owner, now);
    return true;
  }
```

Tests:

- Should be able to resume Presale

TESTS

1. Should be able to use the constructor
2. Owner of token should be presale after deployment
3. Should have 0 ETH in the wallet
4. Should not be able to buy correct amount of tokens if not in whitesale
5. Should not buy tokens if address is contract
6. Should be able to buy correct amount of tokens if in whitesale
7. Should have 25 ETH in the wallet after a valid purchase of 25 ETH
8. Should not buy tokens when paused
9. Should be able to resume Presale
10. Should not transferownershiptoken when invoking as owner
11. Should transferownershiptoken when invoking as multisigwallet
12. Should buy tokens to setup for further tests
13. Should not buy tokens when not enough wei
14. Should not buy tokens when to much wei
15. Should not buy tokens over eth cap
16. Should buy tokens to the ethercap
17. Should deploy another contract
18. Should not be able to buy if tokens go over cap
19. Should not transfer ownership as owner
20. Should transferownership when invoking as multisigwallet
21. Should not finalize presale when not ended
22. Should not finalize presale when not multisig
23. Should finalize presale when all requirements are met as multisigwallet + changed owner of token
24. Should drop minimumETH check when gap to cap is less than minimumETH

The total function coverage is 90 percent.

MULTISIGWALLET.SOL

MODIFIERS

ONLYWALLET

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

    }
```

Tests:

- All tests

OWNERDOESNOTEXIST

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

Tests:

- Should fail when submitting a function when is not in the owners list
- Should fail when confirming a function when is not in the owners list

OWNEREXISTS

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

Tests:

- Should not replace owner after confirmations if owner already is an owner

TRANSACTIONEXISTS

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

Tests:

- Should fail when confirm transaction that already is confirmed by this owner
- Should fail when executing transaction that already is executed
- Should fail when executing transaction that not has enough confirmations

CONFIRMED

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

Tests:

- Should fail when confirm transaction that already is confirmed by this owner

NOTCONFIRMED

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

Tests:

- Should fail when executing transaction that not has enough confirmations

NOTEXECUTED

```
modifier notExecuted(uint transactionId) {
        if (transactions[transactionId].executed) {
            revert();
        }
        _;
    }
```

Tests:

- Should fail when confirm transaction that already is confirmed by this owner
- Should fail when executing transaction that not has enough confirmations

NOTNULL

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

Tests:

- Should fail when owners address is invalid

VALIDREQUIREMENT

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

Tests:

- Should change requirement from 2 to 1 and back to 2

FUNCTIONS

FALLBACK

```
function() public
        payable
    {
        if (msg.value > 0) {
            Deposit(msg.sender, msg.value);
        }
    }
```

Tests:

- Test fallback function

CONSTRUCTOR

```
function MultiSigWallet(address[] _owners, uint _required,
address _token)
        public
        validRequirement(_owners.length, _required)
    {
        require(_token != address(0));

        for (uint i = 0; i < _owners.length; i++) {
            if (isOwner[_owners[i]] || _owners[i] == 0 || _owners[i]
== address(0)) {
                revert();
            }
            isOwner[_owners[i]] = true;
        }
        owners = _owners;
        required = _required;
        token = Identify(_token);
    }
```

Tests:

- Should be able to use the constructor

ADDOWNER

```
function addOwner(address owner)
        public
        onlyWallet
        ownerDoesNotExist(owner)
        notNull(owner)
        validRequirement(owners.length + 1, required)
        returns (bool)
    {
        isOwner[owner] = true;
        owners.push(owner);
        OwnerAddition(owner);
        return true;
    }
```

Tests:

- Should add owner after confirmations+ required stays at 2

REMOVEOWNER

```
function removeOwner(address owner)
        public
        onlyWallet
        ownerExists(owner)
        returns (bool)
    {
        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);
        return true;
    }
```

Tests:

- Should remove owner after confirmations + required stays at 2

REPLACEOWNER

```
function replaceOwner(address owner, address newOwner)
        public
        onlyWallet
        ownerExists(owner)
        ownerDoesNotExist(newOwner)
        returns (bool)
    {
        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);
        return true;
    }
```

Tests:

- Should replace owner after confirmations + required stays at 2
- Should not replace owner after confirmations if owner already is an owner

CHANGEREQUIREMENT

```
function changeRequirement(uint _required)
        public
        onlyWallet
        validRequirement(owners.length, _required)
        returns (bool)
    {
        required = _required;
        RequirementChange(_required);
        return true;
    }
```

Tests:

- Should change requirement from 2 to 1 and back to 2

SUBMITTRANSACTION

```
function submitTransaction(address destination, uint value, bytes
data)
        public
        returns (uint transactionId)
    {
        transactionId = addTransaction(destination, value, data);
        confirmTransaction(transactionId);
    }
```

Tests:

- All tests

CONFIRMTRANSACTION

```
function confirmTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        transactionExists(transactionId)
        notConfirmed(transactionId, msg.sender)
        returns (bool)
    {
        confirmations[transactionId][msg.sender] = true;
        Confirmation(msg.sender, transactionId);
        executeTransaction(transactionId);
        return true;
    }
```

Tests:

- Should change requirement from 2 to 1 and back to 2
- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction
- Should fail when confirm transaction that already is confirmed by this owner
- Should fail when executing transaction that already is executed
- Should transfer ownership of token through presale to account_one
- Should add owner after confirmations+ required stays at 2
- Should remove owner after confirmations + required stays at 2
- Should replace owner after confirmations + required stays at 2
- Should not replace owner after confirmations if owner already is an owner
- Should fail when confirming a function when is not in the owners list
- Should transfer Identify tokens to an account after confirmation

REVOKECONFIRMATION

```
function revokeConfirmation(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
        returns (bool)
    {
        confirmations[transactionId][msg.sender] = false;
        Revocation(msg.sender, transactionId);
        return true;
    }
```

Tests:

- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction

EXECUTETRANSACTION

```
function executeTransaction(uint transactionId)
        public
        notExecuted(transactionId)
        returns (bool)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage _tx = transactions[transactionId];
            _tx.executed = true;
            if (_tx.destination.call.value(_tx.value)(_tx.data)) {
                Execution(transactionId);
            } else {
                ExecutionFailure(transactionId);
                _tx.executed = false;
            }
        }
        return true;
    }
```

Tests:

- Should fail when executing transaction that already is executed
- Should fail when executing transaction that not has enough confirmations

ISCONFIRMED

```
function isConfirmed(uint transactionId)
        public
        view
        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;
            }
        }
    }
```

Tests:

- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction

ADDTRANSACTION

```
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);
    }
```

Tests:

- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction

GETCONFIRMATIONCOUNT

```
function getConfirmationCount(uint transactionId)
        public
        view
        returns (uint count)
    {
        for (uint i = 0; i < owners.length; i++) {
            if (confirmations[transactionId][owners[i]]) {
                count += 1;
            }
        }
    }
```

Tests:

- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction

GETTRANSACTIONCOUNT

```
function getTransactionCount(bool pending, bool executed)
        public
        view
        returns (uint count)
    {
        for (uint i = 0; i < transactionCount; i++) {
            if (pending && !transactions[i].executed || executed &&
transactions[i].executed) {
                count += 1;
            }
        }
    }
```

Tests:

- Should return 13 at getTransactionCount
- Should have 8 executed transactions

GETOWNERS

```
function getOwners()
        public
        view
        returns (address[])
    {
        return owners;
    }
```

Tests:

- Should add owner after confirmations+ required stays at 2
- Should remove owner after confirmations + required stays at 2
- Should replace owner after confirmations + required stays at 2
- Should not replace owner after confirmations if owner already is an owner
- Should fail when submitting a function when is not in the owners list
- Should fail when confirming a function when is not in the owners list

GETCONFIRMATIONS

```
function getConfirmations(uint transactionId)
        public
        view
        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];
        }
    }
```

Tests:

- Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction

GETTRANSACTIONIDS

```
function getTransactionIds(uint from, uint to, bool pending, bool
executed)
        public
        view
        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];
        }
    }
```

Tests:

- Should return getTransactionIds properly
- Should fail when more than 13 transactions at getTransactionIds

GETIDENTIFYAMOUNT

```
function getIdentifyAmount() public view returns (uint256
balance) {
        return token.balanceOf(this);
    }
```

Tests:

- Should display Identify tokens properly
- Should transfer Identify tokens to an account after confirmation

TRANSFERIDENTIFY

```
function transferIdentify(address _to, uint256 _value) onlyWallet
public returns (bool) {
        require(_to != address(0));
        require(_value > 0);
        require(token.transfer(_to,_value));
        return true;
    }
```

Tests:

- Should transfer Identify tokens to an account after confirmation

TESTS

1. Should be able to use the constructor
2. Test fallback function
3. Should change requirement from 2 to 1 and back to 2
4. Submit transaction + get confirmation count + isconfirmed + revoke confirmation + confirm 2 x transaction + execute transaction
5. Should fail when confirm transaction that already is confirmed by this owner
6. Should fail when executing transaction that already is executed
7. Should fail when executing transaction that not has enough confirmations
8. Should get owners properly
9. Should transfer ownership of token through presale to account_one
10. Should display Identify tokens properly
11. Should add owner after confirmations+ required stays at 2
12. Should remove owner after confirmations + required stays at 2
13. Should replace owner after confirmations + required stays at 2
14. Should not replace owner after confirmations if owner already is an owner
15. Should fail when submitting a function when is not in the owners list
16. Should fail when confirming a function when is not in the owners list
17. Should transfer Identify tokens to an account after confirmation
18. Should return getTransactionIds properly
19. Should fail when more than 13 transactions at getTransactionIds
20. Should return 13 at getTransactionCount
21. Should have 8 executed transactions
22. Should fail when owners address is invalid

The total function coverage is 100 percent.

IDENTIFY.SOL

MODIFIERS

ONLYOWNER

```
modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }
```

Tests:

- Should not transferfrom tokens if not owner of contract
- Should throw error when disable transfer is invoked by owner
- Should not transfer ownership when not owner of contract
- Should transfer ownership when owner of contract

WHENTRANSFERENABLED

```
modifier whenTransferEnabled() {
    require(enableTransfer);
    _;
  }
```

Tests:

- Should throw error when disable transfer is invoked by owner

FUNCTIONS

OWNABLE

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

TRANSFEROWNERSHIP

```
function transferOwnership(address newOwner) public onlyOwner
returns (bool) {
    require(newOwner != address(0));
    owner = newOwner;
    OwnershipTransferred(owner, newOwner);
    return true;
  }
```

Tests:

- Should not transfer ownership when not owner of contract
- Should transfer ownership when owner of contract

CONSTRUCTOR

```
function Identify() public {
    totalSupply = INITIAL_SUPPLY;
    balances[this] = INITIAL_SUPPLY;
    Transfer(0x0, this, INITIAL_SUPPLY);
  }
```

Tests:

- Should be able to use the constructor

TOTALSUPPLY

```
function totalSupply() public view returns (uint256) {
    return totalSupply;
  }
```

Tests:

- Should burn tokens and update the totalsupply of it

BALANCEOF

```
function balanceOf(address _owner) public view returns (uint256
balance) {
    return balances[_owner];
  }
```

Tests:

- Should transfer tokens correctly
- Should fail when address for transfer is invalid
- Should approve successful tokens and let the approver spend them
- Should not spend more than approved
- Should increase correctly

TRANSFER

```
function transfer(address _to, uint256 _value)
whenTransferEnabled public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);

       balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    Transfer(msg.sender, _to, _value);
    return true;
  }
```

Tests:

- Should transfer tokens correctly
- Should fail when address for transfer is invalid
- Should throw error when insufficient amount
- Should throw error when disable transfer is invoked by owner
- Should approve successful tokens and let the approver spend them

TRANSFERFROM

```
function transferFrom(address _from, address _to, uint256
_value) whenTransferEnabled public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[_from]);


    if (msg.sender!=owner) {
      require(_value <= allowed[_from][msg.sender]);
      allowed[_from][msg.sender] =
allowed[_from][msg.sender].sub(_value);
      balances[_from] = balances[_from].sub(_value);
      balances[_to] = balances[_to].add(_value);
    }  else {
      balances[_from] = balances[_from].sub(_value);
      balances[_to] = balances[_to].add(_value);
    }

    Transfer(_from, _to, _value);
    return true;
  }
```

Tests:

- Should transfer tokens correctly
- Should not transferfrom tokens if not owner of contract
- Should only transferfrom from other accounts when owner
- Should approve successful tokens and let the approver spend them
- Should not spend more than approved
- Should increase correctly

APPROVE

```
function approve(address _spender, uint256 _value)
whenTransferEnabled public returns (bool) {
                require((_value == 0) ||
(allowed[msg.sender][_spender] == 0));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
  }
```

Tests:

- Should approve successful tokens and let the approver spend them
- Should not spend more than approved
- Should fail when overwriting approval
- Should decrease correctly

APPROVEANDCALLASCONTRACT

```
function approveAndCallAsContract(address _spender, uint256
_value, bytes _extraData) onlyOwner public returns (bool
success) {
              allowed[this][_spender] = _value;
    Approval(this, _spender, _value);


require(_spender.call(bytes4(bytes32(keccak256('receiveApproval
(address,uint256,address,bytes)'))), this, _value, this,
_extraData));
    return true;
  }
```

Tests:

- Should use approveandcallascontract properly

APPROVEANDCALL

```
function approveAndCall(address _spender, uint256 _value, bytes
_extraData) whenTransferEnabled public returns (bool success) {
        require((_value == 0) || (allowed[msg.sender][_spender]
== 0));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);


require(_spender.call(bytes4(bytes32(keccak256('receiveApproval
(address,uint256,address,bytes)'))), msg.sender, _value, this,
_extraData));
    return true;
  }
```

Tests:

- Should use the approveandcall method
- Should fail when already approved through approveandcall method

Threats:

- Possible integer underflow
- Reentrancy

The threats above are not of a critical kind. They won’t have an impact.

ALLOWANCE

```
function allowance(address _owner, address _spender) public
view returns (uint256) {
    return allowed[_owner][_spender];
  }
```

Tests:

- Should decrease correctly
- Should use the approveandcall method

INCREASEAPPROVAL

```
function increaseApproval(address _spender, uint _addedValue)
whenTransferEnabled public returns (bool) {
    allowed[msg.sender][_spender] =
allowed[msg.sender][_spender].add(_addedValue);
    Approval(msg.sender, _spender,
allowed[msg.sender][_spender]);
    return true;
  }
```

Tests:

- Should increase correctly

DECREASEAPPROVAL

```
function decreaseApproval(address _spender, uint
_subtractedValue) whenTransferEnabled 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;
  }
```

Tests:

- Should decrease correctly

BURN

```
function burn(address _burner, uint256 _value) onlyOwner public
returns (bool) {
    require(_value <= balances[_burner]);
          balances[_burner] = balances[_burner].sub(_value);
    totalSupply = totalSupply.sub(_value);
    Burn(_burner, _value);
    return true;
  }
```

Tests:

- Should burn tokens and update the totalsupply of it
- Should not burn more tokens than owner has
- Should not burn tokens if not owner of contract

ENABLETRANSFER

```
function enableTransfer() onlyOwner public returns (bool) {
    enableTransfer = true;
    EnableTransfer(owner, now);
    return true;
  }
```

Tests:

- Should throw error when disable transfer is invoked by owner

DISABLETRANSFER

```
function disableTransfer() onlyOwner whenTransferEnabled public
returns (bool) {
    enableTransfer = false;
    DisableTransfer(owner, now);
    return true;
  }
```

Tests:

- Should throw error when disable transfer is invoked by owner

TESTS

1. Should be able to use the constructor
2. Should transfer tokens correctly
3. Should fail when address for transfer is invalid
4. Should not transferfrom tokens if not owner of contract
5. Should throw error when insufficient amount
6. Should throw error when disable transfer is invoked by owner
7. Should burn tokens and update the totalsupply of it
8. Should not burn more tokens than owner has
9. Should only transferfrom from other accounts when owner
10. Should not burn tokens if not owner of contract
11. Should not transfer ownership when not owner of contract
12. Should approve successful tokens and let the approver spend them
13. Should not spend more than approved
14. Should fail when overwriting approval
15. Should increase correctly
16. Should decrease correctly
17. Should use the approveandcall method
18. Should fail when already approved through approveandcall method
19. Should use approveandcallascontract properly
20. Should transfer ownership when owner of contract

The total function coverage is 93.75 percent.

FOUND THREATS

1. Possible integer underflow
2. Reentrancy

We are aware a possible integer underflow and reentrancy is possible at the “approveAndCall” method in the CustomToken.sol smart contract. But this is not one of a critical kind.

SAFETY FUNCTIONS IMPLEMENTATION

PAUSE FUNCTION

Implementing a pause function is recommended. The function can only be invoked by the owner (or after modification by multiple admins or owners). This is a safety function when things go wrong, or a bug is found, the damage will be limited. To implement this function, see the code block below.

This function is implemented in the 'Whitelist.sol' and 'Presale.sol'.

```
bool public paused = false;

modifier notPaused() {
   require(!paused);
   _;
}

function pauseContract() public onlyOwner returns (bool) {
   paused = true;
   return true;
}


function resumeContract() public onlyOwner returns (bool) {
   paused = false;
   return true;
}

function thisFunctionCanBeRunnedWhenNotPaused() public notPaused
{
   // Some logic
}
```

ENABLE TRANSFER FUNCTION

The enable transfer will only be used in the transfer and transferFrom (and functions derived from it) functions. Your contract can still work properly, just the transferring of tokens will be enabled or disabled. This can be useful when a malicious person has found a way to extract tokens. Through the enableTransfer this will be stopped. When implementing a burn function, these tokens can be burned.

This function is implemented in the 'CustomToken.sol.

```
bool public transfersEnabled = true;

function enableTransfers(bool _transfersEnabled) public onlyOwner {
   transfersEnabled = _transfersEnabled;
}

function transfer(address _to, uint256 _value) public returns (bool) {
   //Check if transfer enabled.
   //Can also be done as a modifier like the pause function.
   require(transferEnabled);
   // Some logic
   return true;
}
```

ISCONTRACT FUNCTION

Contracts cannot participate in the presale. This choice is obviously made to reduce the attack vector.

This function is implemented in the ‘Presale.sol’.

```
/**
   * @dev Internal function to determine if an address is a contract
   * @param _addr The address being queried
   * @return True if `_addr` is a contract
   */

  function isContract(address _addr) constant internal returns (bool) {
    if (_addr == 0) {
      return false;
    }
    uint256 size;
    assembly {
        size := extcodesize(_addr)
     }
    return (size > 0);
  }
```

WRITING MALICIOUS CODE

At the ‘CustomToken.sol’ contract there is a way to do a reentrancy. This is due to the low-level ‘call’ function. Using the ‘approveandcall’ method. Even though a malicious contract can call this function, this attack has no impact on the contract.

TESTED VULNERABILITIES

In the table below, we summed up some well-known vulnerabilities. Divided in 3 levels: Solidity, EVM and blockchain. On the last column a description is of possibility is written down.

Level	**Cause of vulnerability**	**Possible**
Solidity	Call to the unknown (DAO)	Only in the approveAndCall method. But this has no critical effects.
	Gasless send	No.
	Exception disorders	Using ‘required’ keywords to check input parameters.
	Type casts	Not applicable.
	Reentrancy	In the CustomToken approveandcall method due to the low-level call method. But checks are in place to not harm the contract when this is executed.
	Keeping secrets	No secrets are necessary.

EVM	Immutable bugs	Tests are written for finding bugs. Note that there can never be a 100% certainty of bug-free contracts.
	Ether/token lost in transfer	Don’t send to ‘orphan’ addresses. Meaning addresses that are not associated to any user or contract. But these tokens can be transferred back by the owner of the token.
Blockchain	Unpredictable state	Multiple checks are in place.
	Time constraints	Only for begin and end time of presale. But miners can only modify time in a small way. These small modifies have no serious impact on the presale.

STATIC ANALYSIS RESULT[1]

We ran the codebase through multiple open-source analysis tools to check for well-known bugs, errors and attack vectors.

Attack vector	Result
EVM Code Coverage[2]	100%
Callstack Depth Attack Vulnerability	False
Transaction-Ordering Dependence (TOD)	False
Timestamp Dependency	False
Re-Entrancy Vulnerability	False
Parity multisig bug 2	False

[1] Using multiple open source tools like Oyente and Mythril
[2] Number of opcodes executed / total number of opcodes

WORST CASE SCENARIO

LOSS OF PRIVATE KEY

Shared (Identify and TheLedger.) uploaders account

There are multiple backups of this account and key. If someone loses his key, the wallet can still be accessed by the backup.

Owner of multisig wallet account

The owner of the multisig wallet that has lost its private key can be removed by the other owners and a new wallet can be added if necessary. In this case, there still need to be more owners than required signatures to use this solution.

THE PRIVATE KEY IN WRONG HANDS

Shared (Identify and TheLedger.) uploaders account

The malicious person could call the pause method on the whitelist and presale contract and steal the remaining ETH that is still in the account after uploading the contracts. The ETH that is raised through the presale is safe in the multisig wallet account. When noticed before the remaining ETH is stolen, it can be transferred to a new safe address.

Owner of multisig wallet account

No damage can be done, because multiple signatures are needed to execute a transaction. Remove the account from the owners list and add a new one.

BUG IN OWN CODE

The damage that is done depends on the contract that is compromised. We will pause the contract, debug the contract and then go from there to find the best solution for the problem at hand.