Object Computing, Inc.
12140 Woodcrest Executive Dr.
Suite 300
St. Louis, MO 63141
+1 314.579.0066

TeleportToken Contract Audit (ETH & EOS)

https://github.com/Alien-Worlds/alienteleport/blob/master/contracts/teleporteth/contracts/TeleportToken.sol

Audit Date: 3/22/22
Auditors: Jesse Hanner & Jack DiSalvatore

Solidity Scope:

The contract has been audited for the following:

1. Security vulnerabilities
2. Logical errors
3. Performance optimization
4. Conformity to Solidity best practices (https://github.com/crytic/building-secure-contracts)

Solidity Methodology:

The Solidity audit was conducted using the slither solidity static analyzer, followed by manual review. Slither checks the code for 76 security vulnerabilities and code inefficiencies and outputs a list of suspected issues, each having a confidence and severity level. Once the list of potential issues is identified, each item is manually reviewed in the contract to ensure the analyzer correctly found a problem, then it is verified against the solidity docs.

Severity Description:

Remark

Remarks are instances in the code that are worthy of attention but in no way represent a security flaw in the code. These issues might cause problems with the user experience, confusion with new developers working on the project or other inconveniences.
Things that fall under remarks would include:

- Instances where best practices are not followed
- Spelling and grammar mistakes

- Inconsistencies in the code styling and structure

Minor

Minor severity can cause problems in the code, but would not cause the code to crash unexpectedly or for funds to be lost. It might cause results that would be unexpected to users, or minor disruptions in operations. Minor problems are prone to become major problems if not addressed appropriately.
Things that fall under remarks would include:

- Logic flaws (excluding those that cause crashes or loss of funds)
- Code duplication
- Ambiguous code

Major

Major security issues can cause the code to crash unexpectedly, or lead to deadlock situations.
Things that fall under remarks would include:

- Logic flaws that cause crashes
- Timeout exceptions
- Incorrect ABI file generation
- Unrestricted resource usage (for example, users can lock all RAM on contract)

Critical

Critical issues cause a loss of funds or severely impact contract usage.
Things that fall under remarks would include:

- Missing checks for authorization
- Logic flaws that cause loss of funds
- Logic flaws that impact economics of system
- All known exploits (for example, on_notification fake transfer exploit)

Solidity Summary:

Remark	Minor	Major	Critical
10	5	1	0

1. We **highly** recommend writing unit tests for any smart contract to help spot potential errors, currently **TeleportToken.sol** has none: **Remark/Major**
2. Usage of `` **`pragma experimental ABIEncoderV2`** `` is not recommended: **Remark**
3. **_totalSupply** (line #234) and **return _totalSupply** (line #245) should use **SafeMath** to avoid mathematical errors: **Remark**
4. **TeleportToken.verifySigData(bytes)** uses timestamp for comparisons (line #379):
    a. It is recommended to avoid using **block.timestamp** and instead use **block.number**, as miners have the ability to adjust timestamps slightly **Minor**
5. **Owned.transferOwnership(address)** _newOwner (line #160) lacks a **zero-check** **Minor**
6. **Verify.splitSignature(bytes)** (lines l#40-47) and **TeleportToken.verifySigData(bytes)** use **assembly** (lines #360-368).
    a. Inline assembly is a way to access the Ethereum Virtual Machine at a low level. **This bypasses several important safety features and checks of Solidity.** Assembly requires EVM expertise. Do not write EVM code if you have not mastered the yellow paper. **Remark/Minor**
7. **TeleportToken.updateThreshold(uint8)** should **emit** an event for: threshold = newThreshold (line #428) **Remark**
8. **TeleportToken.updateChainId(uint8)** should **emit** an event for: thisChainId = newChainId (line #439) **Remark**
9. **Endian.reverse16(uint16)** (lines #83-88), **SafeMath.div(uint256,uint256)** (lines #107-110) and **SafeMath.mul(uint256,uint256)** (lines #103-106) are never used and should be removed. **Remark**
10. Why were **SafeMath, Owned** and **ERC20Interface** rewritten instead of imported from **OpenZepplin**? It's recommended to use well-tested libraries. Importing code from well-tested libraries will reduce the likelihood that you write buggy code. If you want to write an ERC20 contract, use OpenZeppelin: **Remark**
11. **totalSupply(), balanceOf(address), allowance(address,address), transfer(address,uint256), approve(address,uint256), transferFrom(address,address,uint256),receiveApproval(address,uint256,address,bytes), transferOwnership(address), acceptOwnership(), regOracle(address),**

unregOracle(address), approveAndCall(address,uint256,bytes), teleport(string,uint256,uint256), claim(bytes,bytes[]), updateThreshold(uint8), updateChainId(uint8) and **transferAnyERC20Token(address,uint256)** should be declared **external (line #118 - 128)**:

   a. **public** functions that are never called by the contract should be declared **external** to save gas: **Remark (Optimization)**

12. **constructors (line #150, #230)** don't need **public** keyword**: Remark**
13. **Emit** should be at end of function (line #165 after #167): **Remark (Convention)**
14. Add:
    a. **require(tokens <= balances[msg.sender])** and **require(to != address(0))** (after line #262) [openzepplin]
    Minor
    b. **require(spender != address(0))** (after line #278) [openzepplin] **Minor**
15. Line #329 says "tokens : number of tokens in satoshis", this contract in no way is supposed to interact with BTC correct? **Remark**
16. Line #349 **verifySigData()** doesn't actually verify signatures, maybe rename **buildSigData(): Remark**
17. Line# 415 check that address(0) has the quantity to subtract. Otherwise, if address(0)'s balance is zero, then subtracting would cause an overflow because it is a unit **Minor**

Contracts

1. SafeMath - why not use the OpenZeppelin library?
2. ERC20Interface - why not use the OpenZeppelin library?
3. Owned - why not use the OpenZeppelin Ownable library?
4. TeleportToken - main token contract. Why derive from the ERC20Interface and re-implement the functions? Instead you could just derive from ERC20 and not have to re-implement the functions.

Methodology EOSIO

1. **Known Security Issues Checked**

    a. **Numerical Overflow:** when doing token math, use the `asset` object operands and do not perform operations on the `uint256` that comes from `asset.amount`

    b. **Authorization Check:** make sure function parameters are consistent with the caller and use `require_auth` to check

    c. **Apply Check:** if the contract implements an `apply` function, bind each key action and code to meet the requirements, in order to avoid abnormal and illegal calls.

    d. **Transfer Error Prompt:** When processing a notification triggered by `require_recipient`, ensure that `transfer.to` is `_self`

    e. **Random Number Practice:** Random number generator algorithm should not introduce controllable or predictable seeds

    f. **Rollback Attack:** There is a time difference between when an API node initiates the request and synchronizes to the BP node. (In the context of a gambling dapp) Attackers use this vulnerability to place bets on the API node. If they find that there is no winning, they can return the EOS for each bet so that they can rejoin the game without cost. Attackers can repeat this process until they have a winning bet. One solution to this is for your dapp to use read/write separation and have Read-Only and Write-Only nodes to interact with the smart contract.

2. **Business Flow Documentation vs Implementation**
   Review contract documentation and validate that the code implements the known business requirements (no logical flaws).

   EOS -> ETH
    1. Contract account registers oracle account's with `regoracle` action
    2. End user deposits TLM tokens into contract with a standard transfer (no memo required)
    3. End user calls the `teleport` action to start the teleport

   ETH -> EOS
    1. Oracle calls `received` with the amount of TLM to be teleported to EOS
    2. Two more oracles call `received` to meet the threshold
    3. Oracle calls `claimed` to mark teleport as complete
3. **Unit Test Review**
    a. **regoracle**
        i. Fails without correct auth
        ii. Succeeds with correct auth and add oracles to the `oracles` table
        iii. Succeeds at adding oracle2
        iv. Succeeds at adding oracle3
        v. Succeeds at adding oracle4
            1. `oracles` table row is added

b. **unregoracle**
   i. Fails with incorrect auth
   ii. Succeeds with correct auth
      1. `oracles` table row is removed

c. **received** (BSC/ETH)
   i. Fails with un-registered oracle
   ii. Fails with registered oracle and wrong auth
   iii. Succeeds with registered oracle and correct auth
   iv. `receipt` table row is added
   v. Succeeds with another registered oracles with correct auth and adds another receipt to the existing row in `receipt` table
      1. No check on the `receipt` table row to make sure modified
   vi. Fails with quantity mis-match
   vii. Fails with user account mis-match
   viii. Fails with already signed error when same oracle signs twice
   ix. Succeeds with approval from 3 oracles
      1. Checks tokens are transfered
      2. Checks `receipt` table is updated

d. **recrepair**
   i. Fails with wrong auth
   ii. Fails with correct auth and non-existing `receipt`
   iii. Fails with correct auth, existing `receipt`, and invalid quantity
   iv. Fails with correct auth, existing `receipt`, and negative quantity
   v. Succeeds with correct auth, existing `receipt`, and correct quantity.
      1. `receipt` table will be updated

e. **teleport**
   i. Fails without valid auth
   ii. Fails with valid auth and invalid quantity
   iii. Fails with valid auth and quantity below the minimum
   iv. Fails with valid auth and no deposits available
   v. Fails with valid auth and when not enough tokens are deposited
   vi. Succeeds with valid when there are enough tokens deposited
      1. `teleports` table row is updated/added

Summary EOSIO:

Remark	Minor	Major	Critical
4	4	0	0

Findings

1. **teleporteos.test.ts#171** says it should add another receipt but does not have code to check that it does. **Minor**
2. Missing comment on **teleport.hpp#46** **Remark**
3. Comment on **teleport.hpp#61** looks incorrect **Remark**

4. **teleporteos.cpp#15** use asset comparison instead of amount comparison. (i.e. `check(quantity >= asset(100'0000, “TLM”));` **Minor**
5. **teleporteos.cpp#56** use asset comparison **Minor**
6. **teleporteos.cpp#63-70 “**tokens owned by this contract are inaccessible so just remove the deposit record”. Does this mean this contract will end up holding a balance of “TLM” tokens that can’t be retrieved? If so, why not burn the tokens to contract the supply? **Minor**
7. **teleporteos.cpp#98-101** code says cancellation can only take place after 32 days, but documentation says users cancel after 30 days **Remark**
8. **teleporteos.cpp#252 and teleporteos.cpp#261** dangerous functions could delete user data. **Remark**

Recommendations

- Be consistent with comment formatting styles
- Limiting the deletion for the **teleport** and **receipts** table. You could provide a int32/64 parameter to the action, which allows indefinite removal , unless the parameter is set to 1 or greater in which case it limits it to that many. Therefore you could always call delteleport(0), unless it fails and then call cleantable(x). This would also allow you to cleanup the table while the table is active.

Contracts Audited

1. **teleporteos**

   teleporteos.hpp

   Tables
   - deposits
   - teleports
   - cancels
   - oracles
   - receipts

   teleporteos.cpp

   Actions
   - withdraw
   - teleport
   - cancel
   - logteleport
   - sign
   - received
   - repairrec
   - claimed
   - regoracle
   - unregoracle
   - delreceipts
   - delteles

   Notify Handlers
   - transfer

Oracle Audit (Javascript)

Methodology Oracle

Oracle files audited:

- config-example.js
- ecosystem.config.example.js
- eth_abis.js

- [JACK] oracle-eos.js: `This oracle listens to the EOSIO chain for teleport actions using a state history node.  After receiving a teleport action, it will sign the data and then send it to the EOS chain where the client can pick. it up and then send it to the ethereum chain in a claim action`
- [JESSE] oracle-eth.js: `` This oracle listens to the ethereum blockchain for `Teleport` events. When an event is received, it will call the `received` action on the EOS chain ``
- [JESSE] oracle-eth.ts: `` This oracle listens to the ethereum blockchain for `Teleport` events. When an event is received, it will call the `received` action on the EOS chain (types aren't being used) ``
- [JESSE] check-eth.js: `Checks a transaction on ETH/BSC and reports completion status`

- [JACK] incomplete-eos.js: `Lists all incomplete teleports from eos -> eth`
- [JESSE] incomplete-eth.js: `Lists all incomplete teleports from eos -> eth`

- [JACK] txdispatch.js

Script files audit:

- teleport.ts
- get_sign_data.ts

Summary Oracle:

oracle-eos.js

1. line#14 `Serialize` import from `eosjs` is not used and can be removed
2. line#20 & 25 `web3` is never used and can be removed
3. line#77 **chain_data** is never used and can be removed

oracle-eth.js

1. line#164 **if (tokens <= 0)** should be **if (tokens = 0)** because data type is uint256 unsigned which can't be negative
2. line#26 **network const** is unused

oracle-eth.ts

1. (types aren't being used)

check-eth.js

1. line#14 **hyperion_endpoint** is unused
2. line#18 **signatureProvider** is unused
3. line#29 **while loop** should use a variable that is initially set to **true**, then changed to **false** in the **else** case. Not **else { break; }**

incomplete-eos.js

1. line#10 `Serialize` import from `eosjs` is not used and can be removed
2. line#21 `eos_api` is unused and can be removed
3. line#27 **while loop** should use a variable that is initially set to **true**, then changed to **false** in the **else** case. Not **else { break; }**
4. lines#69-72 When an error happens the url request is retired, but if the error continues to happen the code will be stuck in a failing retry loop. Considered adding a retry limit to prevent infinite loops.

incomplete-eth.js

1. line#10 **Serialize** import isn't used
2. line#15 **hyperion_endpoint** const isn't used
3. line#21 **eos_api** const isn't used
4. line#27 **while loop** should use a variable that is initially set to **true**, then changed to **false** in the **else** case. Not **else { break; }**

txdispatch.js

1. line#4 `Serialize` import from `eosjs` is not used and can be removed

Teleport Process Step-by-step:

https://alienworlds.zendesk.com/hc/en-us/articles/1500012518322-Teleport-Update-Tokens-Travel-Throughout-the-Metaverse

Setup

1. On the EOS chain, the contract owner of *teleporteos* registers 3 oracle accounts

Execution - Teleport Canceled EOS

1. End user **deposits** TLM into EOS contract
2. End user start the **teleport** on EOS contract
3. After 32 days end user **cancels** the teleport on EOS contract
4. EOS contract will send canceled teleport tokens back to end user

Execution - Teleport Success EOS to ETH

1. End user **deposit**s TLM token into EOS contract.
2. End user start the **teleport** on EOS contract
3. A registered oracle listens to the **teleport** action on the EOS chain via a State History Node
4. After receiving a teleport action, oracle calls **sign** on the EOS contract
5. On the Ethereum side, end user calls the **claim** function, which verifies the oracles signatures
    a. Subtracts TLM tokens from address 0
    b. Add TLM tokens to end user
6. Oracle sends **claimed** action to EOS contract marking the teleport complete

Execution - Teleport Success ETH to EOS

1. End user calls **teleport** on ETH contract, deposits TLM into address(0) on chain_id
2. Oracle sees **teleport** event
3. Oracles call **received** on the EOS contract
4. Another oracle calls **received** on the EOS contract, and the confirmations increase
5. A third oracle calls **received** on the EOS contract, completing all the confirmations, and sending the TLM to the end user on EOS