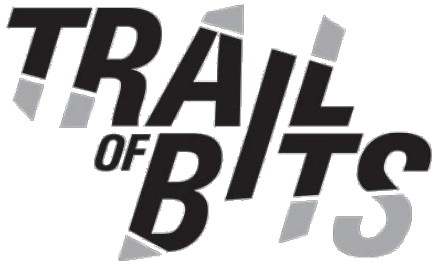

PISHON FINANCE

Security Assessment

November 14, 2020

Prepared For:
Song Lao | ***PISHON***
pishonTron@gmail.com

Prepared By:
Sam Moelius | ***Trail of Bits***
sam.moelius@trailofbits.com

Sam Sun | ***Trail of Bits***
sam.sun@trailofbits.com

Changelog:

November 9, 2020:	Initial report delivered
November 14, 2020:	Added Appendix E. Fix Log
November 15, 2020:	Audit results

Executive Summary

From November 9 through November 18, 2020, PIS (PISHON FINACNE) engaged Trail of Bits to review the security of the PIS contracts on Tron Blockchain. Trail of Bits conducted this assessment over the course of two person-weeks with two engineers working from various zip archives provided via email and Slack.

we verified that we could compile the TRON contracts. We also verified that the TRON unit tests pass, and collected their code coverage using `solidity-coverage`. We ran Slither (Trail of Bits' Solidity source code analyzer) over the TRON contracts. Finally, we began manual review of the TRON contracts.

Our efforts resulted in 12 findings ranging from medium to informational severity. The one medium-severity finding concerns how expired or soon-to-be-expired backups could be reintroduced (pispool). The one low-severity finding concerns how proposed method IDs are computed (pusdswap). The remaining 10 findings are informational.

In addition to the 12 findings, we prepared multiple appendices. Notably, Appendix C contains findings that do not have immediate or obvious security implications, while Appendix D describes our threat model for the TRON contracts.

Our main recommendation is that additional unit tests be developed for the TRON contracts. The TRON contracts' "happy" (i.e., successful) paths are tested quite extensively. However, they would benefit from additional tests of their "sad" (i.e., failing) paths. following each code modification, for example. Something analogous to "`truffle test`" would be ideal. A comprehensive set of unit tests that can be run locally will help expose errors and protect against regressions.

A secondary recommendation concerns the use of `this` calls in the TRON pusdswap contracts. The contracts protect functions intended for `this` calls with the `allowSelfCallsOnly` modifier. However, our concern is that an attacker could benefit from making a `this` call to a publicly exposed function that is not protected by the `allowSelfCallsOnly` modifier. To address this concern, we recommend adopting one of the following strategies:

B Explicitly whitelist all `this` calls. Specifically, for each logic contract, maintain a list of function identifiers allowed for `this` calls. Before making a `this` call, ensure it involves a function whose identifier is on the list. Note that the `ProposalLogic` contract already employs such a mechanism for proposed method IDs.

C Use build scripts to ensure new public functions are not introduced. Specifically, for each logic contract, write a build script to verify that its ABI includes only public functions from a known, good ABI. Consider the presence of any additional public functions a build failure. If a code modification introduces a new public function to a contract's ABI, consider whether the function should be made internal, or whether the known, good ABI should be updated.

Adopting one of these two strategies will help protect your users from bugs like the one found in `executeProposal` just before this assessment.

Finally, the informational severity of some of the findings is subject to the correct usage of an API (earnedRefer, GetUserDatalite, StakedPusd, PisSwapToUsdt, buySuperPower, stakePusd, withdraw, daybonus, getReward, top49AdressCommpowerBonus, top200AddresspowerBonus, PisTotalSupply, totalAllUsers, totalPusdStake, PissSwapToUSDT, totalSupply, totalSupplyRefer, earnedRefer, GetUserDatalite, getTop49, getTop200,BuySuperPower, Top49CommPowerBonus, Top200PowerBonus, DaoPowerBonus, StakedPusd, RewardPaid, PissSwapToUsdt, trxInToUsdToken, pusdTotrx, usdtInToPusdToken, pusdTousdt, TrxInToPusdToken, UsdtInToPusdToken, PusdTotrx, PusdTpusdt).

Project Dashboard

Application Summary

Name	PISHON FINANCE
Version	`PISHON-TRON-Contracts-review-20201109.zip`
Type	Solidity
Platforms	TRON

Engagement Summary

Dates	November 9 through November 18, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	0	
Total Low-Severity Issues	1	
Total Informational-Severity Issues	10	
Total	11	

Category Breakdown

Auditing and Logging	1	
Data Validation	2	
Denial of Service	1	
Patching	3	
Timing	1	
Undefined Behavior	2	
Total	10	

Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning.

Category Name	Description
Access Controls	**Satisfactory.** Though we found no issues related to the use of `this` calls in the Tron logic contracts, we recommend additional mitigations to protect against their misuse .
Arithmetic	**Satisfactory.** We found one issue regarding the handling of backup expiries. No other significant arithmetic issues were found.
Assembly Use	**Satisfactory.** An issue was found in the assembly code that computes proposed method IDs. No other significant assembly use issues were found.
Centralization	**Satisfactory.** The `LogicManager` contract is owned by a multi-sig contract. The `AccountCreator` contract is multi-owned, making it more susceptible to takeover. However, this would only affect newly deployed accounts, not existing ones.
Contract Upgradeability	**Strong.** Applies to the `AccountCreator`'s ability to update its `LogicManager`, and the `LogicManager`'s ability to update its individual logic contracts. No issues were found regarding either.
Function Composition	**Moderate.** We found the extensive use of `this` calls in the pispool contracts, and the use of transaction forwarding in the pusdswap contracts, to be confusing. We recommend reducing or eliminating these practices where possible.
Front-Running	**Strong.** No front-running possibilities were noted.
Monitoring	**Not Reviewed.**
Specification	**Strong.** A specification accompanies the implementation. No significant deviations were found between the specification and the implementation.
Testing & Verification	**Weak.** Few of the contract's unsuccessful paths are tested. The file `newpispool.sol` is largely untested.

Contract :

Contract Name	Description
pusdswap	PUSD issues a destruction contract. The contract method is mainly to synthesize PUSD through USDT or TRX. Anyone can issue PUSD. At the same time holding PUSD can also exchange PUSD into TRX or USDT
PUSDToken	PUSD is synthesized by Pusdswap through mortgage USDT or TRX. The total amount is not online, and the mortgage is issued
Pistoken	PisToken has a total of 100 million pieces, which are mined in four years. Users can obtain Pis by purchasing computing power, invitations, and rankings. Pis will be permanently destroyed after being exchanged in the superconducting pool
pispool	pispool is a superconducting pool contract. The superconducting pool contract is mainly used for Pis exchange. The exchanged pis will be permanently destroyed
IRewardDistributionRecipient	The contract method is mainly to trigger system dividends

Event：

BuySuperPower	User purchase hashrate function, pass in the address of the superior, the value of usdt (with precision)
Top49CommPowerBonus	Top49 community computing power dividends, dividend users, dividend pis value (with precision)
Top200PowerBonus	Top200 rushing calculation power dividend, return: dividend address, dividend value, user's superconducting power
DaoPowerBonus	Dao computing power dividend, user address, dividend usdt value (with precision)
StakedPusd	Pledged pusd, pledged user, pledged pusd value
RewardPaid	The user receives his own mining income and returns the number of PIS
PisSwapToUsdt	Pis exchange usdt, amount passed in value, usdtchange exchange value

Method：

buySuperPower	User purchase hashrate function, pass in the address of the superior, the value of usdt (with precision)
stakePusd	The user's current pledged pusd, using pusd to pledge can increase their computing power by 3 times
withdraw	The user extracts the pusd interface and receives the pusd reward
daysbonus	Daily dividend of dao computing power, which needs to be called by the deployer
getReward	Withdraw all mining rewards PIS
top49AdressCommpowerBonus	49 members of the community Dividends are distributed regularly, address is passed in, and the back-end needs to be called regularly
top200AddresspowerBonus	The first 200 people who purchase computing power will get dividends, and the address and the number of them will be passed in, starting from 0

Engagement Goals

The engagement was scoped to assess the security of PISHON on tron network contracts.

Specifically, we sought to answer the following questions:

- Can an attacker change the price of TRX and synthesize more PUSD?
- Can an attacker change the ratio of USDT to synthesize PUSD and synthesize more PUSD?
- Can an attacker illegally request and obtain more superconducting computing power?

- Can an attacker falsely destroy the pusd?
- Attacking this can change the PIS price and redeem more USDT?

Coverage

PIS contract. Statically analyzed using Slither. Unit tests verified to pass and reviewed for code coverage. Manually reviewed.

Pispool contract. Statically analyzed using Slither. Unit tests verified to pass and reviewed for code coverage. Manually reviewed.

pusdswap contract. Statically analyzed using Slither. Unit tests verified to pass and reviewed for code coverage. Manually reviewed.

PUSD contract. Statically analyzed using Slither. Unit tests verified to pass and reviewed for code coverage. Manually reviewed.

IRewardDistributionRecipient contracts (). Statically analyzed using Slither. Unit tests verified to pass and reviewed for code coverage. Wrote additional unit tests. Manually reviewed.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

Measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

Add unit tests. Ideally, every contract document should be tested at least once, and every contract method should be tested at least once. Write unit tests for newpispool, pistoken, pusd, pusdswap contracts.

Findings Summary

#	Title	Type	Severity
1	Solidity compiler optimizations can be dangerous	Undefined Behavior	Informational
2	Insufficient unit tests	Patching	Informational
3	Race condition on `Account.init`	Timing	Informational
4	Use of `abi.encodePacked` could result in collisions	Data Validation	Informational
5	Function `removeBackup` does not sufficiently validate backups	Data Validation	Medium
6	Miner address is calculated incorrectly	Data Validation	Low
7	Unsafe storage load	Data Validation	Informational
8	Newpispool Assets are safe，There is no possibility of being called	Patching	Informational
9	`LogicInitialized` events cannot be trusted	Auditing and Logging	Informational
10	`DecodeBase58` implementation is not up to date	Patching	Informational
11	Incorrect definition for `approveprop_action`	Undefined Behavior	Informational
12	Dual-signed proposals cannot be canceled	Denial of Service	Informational

1. Solidity compiler optimizations can be dangerous

Severity: Informational
Type: Undefined Behavior
Target: `truffle-config.js`

Difficulty: Low
Finding ID: Pishon-01

Description

Newpispool and pusdswap has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them, so it is unclear how well they are tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the Emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until late 2019. The fix for this bug was not reported in the Solidity `CHANGELOG`. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is “implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function.” Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in Pishon contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug. Also, consider using the binary version of `solc`. These steps will help prevent a bug from entering the codebase.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity. This will help prevent a potentially dangerous feature from becoming enabled until it is ready for production use.

Description
The function callOptionalreturn is called to set an ERC20 token field. Since anyone can call this function, anyone can set an account's `manager` field.

The entire `init` function appears in Figure 3.1. Note that the passed-in value `_manager` is only minimally validated: It must be non-null, and it must affirmatively respond to certain `isAuthorized` calls.

```
    function callOptionalReturn(ITRC20 token, bytes memory data) private {
    // We need to perform a low level call here, to bypass Solidity's return data size
checking mechanism, since
    // we're implementing it ourselves.

    // A Solidity high level call has three parts:
    //  1. The target address is checked to verify it contains contract code
    //  2. The call itself is made, and success asserted
    //  3. The return value is decoded, which in turn checks the size of the returned
data.
    // solhint-disable-next-line max-line-length
    require(address(token).isContract(), "SafeTRC20: call to non-contract");

    // solhint-disable-next-line avoid-low-level-calls
   (bool success, bytes memory returndata) = address(token).call(data);
    require(success, "SafeTRC20: low-level call failed");

    if (returndata.length > 0) { // Return data is optional
          // solhint-disable-next-line max-line-length
          require(abi.decode(returndata, (bool)), "SafeTRC20: TRC20 operation did not
succeed");
          }
        }
    }
```

Figure 3.1: `contracts/pusdswap.sol/514-533`

Any TRON network user can call this method through his own address to synthesize the TRX of the account into PUSD (see Figure 3.2). This method derives PUSDToTRX and PUSDtoUSDT. Any user can directly destroy the issued PUSD, and the destroyed PUSD enters the black hole address, and the user address can receive USDT or TRX.

```
function pusdTotrx(uint256 amount) public {
     require(amount > 0 , "Cannot TransferIn  0");
     pusd.transferFrom(msg.sender, address(this), amount);
     pusd.burn(amount);
     uint256 price = usdttokenexchange.getTrxToTokenOutputPrice(1);
     uint256 trxnub = amount.mul(price).mul(90).div(100);
     address(uint160(msg.sender)).transfer(trxnub);
     if (buypusd[msg.sender] > amount ) {
         buypusd[msg.sender]=buypusd[msg.sender].sub(amount);
     } else {
         buypusd[msg.sender]= 0;
     }
     burnpusd[msg.sender]= burnpusd[msg.sender].add(amount);
     emit PusdTotrx(msg.sender, amount, trxnub);
}
function pusdTousdt(uint256 amount) public {
     require(amount > 0 , "Cannot TransferIn  0");
     pusd.transferFrom(msg.sender, address(this), amount);
     pusd.burn(amount);
     usdt.transfer(msg.sender, amount.mul(95).div(100));
     if (buypusd[msg.sender] >amount ) {
         buypusd[msg.sender]=buypusd[msg.sender].sub(amount);
     } else {
         buypusd[msg.sender]= 0;
     }
     burnpusd[msg.sender]= burnpusd[msg.sender].add(amount);
     emit PusdTousdt( msg.sender, amount);
}
```

Figure 3.2: `contracts/pusdswap.sol/661-688`

After the user synthesizes PUSD through TRX or USDT, the user can query the data of PUSD destruction through the interface, and obtain the total amount of TRX and total USDT in the synthesis pool. The contract method in the figure below is mainly to synthesize PUSD and obtain the total number of PUSD burns, the total TRX and USDT of the synthetic pool

```
function trxInToPusdToken() public  payable{

     require(msg.value > 0 , "Cannot TransferIn  0");

     uint256 price = usdttokenexchange.getTrxToTokenOutputPrice(1);

     uint256 amount = msg.value;

     balancetrx.add(amount);

     uint256 usdtnub = amount.div(price);

     //pusd.transfer(msg.sender, usdtnub);

     pusd.mint(msg.sender, usdtnub);

     buypusd[msg.sender]=buypusd[msg.sender].add(usdtnub);

     emit TrxInToPusdToken( msg.sender, amount, usdtnub);
}


function getUserBurnPusd(address user ) public view returns(uint256) {
     return  burnpusd[user];

}

function getalltrx(uint256 amount) public {

     require(amount > 0 , "Cannot TransferIn  0");

     require(msg.sender == governance , "not governance");

     address(uint160(msg.sender)).transfer(amount);
 }

 function getallusdt(uint256 amount) public {

     require(amount > 0 , "Cannot TransferIn  0");

     require(msg.sender == governance , "not governance");

     usdt.transfer(msg.sender, amount);

}
```

Figure 3.3: `contracts/pusdswap.sol/612-633`

This finding is considered informational because, at present, it appears that `transFerFrom` calls `init` immediately after an `PUSDToken` is deployed (see Figure 3.3 and 3.4). In such a case, there is no risk, because internal transactions are not susceptible to race conditions.

```
function burn(address guy, uint wad) public  stoppable {

        require(_balances[guy] >= wad, "ds-token-insufficient-balance");

        _balances[guy] = sub(_balances[guy], wad);

        _supply = sub(_supply, wad);

        emit Burn(guy, wad);
}
```

Figure 3.4: `contracts/pusd.sol#296-L301`.

```
function constructor () public TRC20Detailed("Pis", "PIS", 18) {
      governance = msg.sender;
      //_mint(governance,500*1e18);
  }

  function mint(address account, uint amount) public {
      require(minters[msg.sender], "!minter");
      _mint(account, amount);
  }

    function burn(uint256 amount) public {
        _burn(msg.sender, amount);
    }

  function setGovernance(address _governance) public {
      require(msg.sender == governance, "!governance");
      governance = _governance;
  }

  function addMinter(address _minter) public {
      require(msg.sender == governance, "!governance");
      minters[_minter] = true;
  }

  function removeMinter(address _minter) public {
      require(msg.sender == governance, "!governance");
      minters[_minter] = false;
  }
```

Figure 3.5: `contracts/Pistoken.sol/205-233`

3. Use of `abi.encodePacked` could result in collisions

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: pishon-004
Target: `contracts/newpispool.sol`

Description

The function `GetUserDownlinereferrerpower` uses `abi.encdePacked` with multiple dynamic arguments. Such uses could, in turn, cause calls to `create2` to try to create the user downline

The beginning of the function `GetUserDownlinereferrerpower` appears in Figure 3.1

```
    function GetUserDownlinereferrerpower(address userAddress, address
 downline) public view returns(uint256 ) {

          User storage user = Users[userAddress];
          return (user.downlineComm[downline].referrerpower);
      }
```

Figure 3.1: `contracts/newpispool.sol#L778-L781`.

Note that arguments `_keys` and `_backups` are dynamic arrays passed adjacently to `abi.encodePacked`. So, for example, the following values of `_keys` and `_backups` would cause the call to `abi.encodePacked` to produce the same value:

```
   function GetUserDatalistdownline(address userAddress) public view
returns(address []memory ) {
        User storage user = Users[userAddress];
        return (user.listdownline);
    }
```

The value that `abi.encodePacked` produces is passed to `create2`. If the same arguments are passed to `create2` twice, the second call will fail.

Note that a change to `createCounterfactualAccount` may also warrant a change to `getCounterfactualAccountAddress`.

The client points out that using a different salt in each call to `createCounterfactualAccount` mitigates this issue. We agree and have reduced the finding's severity to informational.

```
        function top49AdressCommpowerBonus(address currentAddress) public
onlyRewardDistribution    returns (bool){
            uint256 subnum = block.timestamp.sub(starttime);
            uint256 dayssub = subnum.div(ONEDAYS);
            require(!top49hadbonus[dayssub][currentAddress], "currentAddress this day had paied
bonus");
            top49hadbonus[dayssub][currentAddress]=true;
            uint256 alltoppower = gettop49Commpower();
            uint256 pissupply = ONEDAYSUPPLY.div(100).mul(15);
            if (alltoppower > 0) {
                  //uint256 uinitpis= pissupply.div(alltoppower);
                    uint256 one=(Users[currentAddress].maxdownlinuser.referrerpower).div( 5) +
                        (Users[currentAddress].referrerpower -
Users[currentAddress].maxdownlinuser.referrerpower).div(10);
                        uint256 bonus = one.mul(pissupply).div(alltoppower);
                        if (bonus >0) {
                              pis.mint(currentAddress,bonus);
                              emit Top49CommPowerBonus(currentAddress, bonus, one);
                        }
            }
              return true;
      }
```

Alice calls `createCounterfactualAccount` twice with values of `_keys` and `_backups` resembling the above bullets. Alice's second call fails.

The contract method is mainly to obtain the top 49 users' profit from the top of the network, rank according to the user's computing power, filter out the winning income of the former users according to the user's address, and distribute reward

Pishon is mainly divided into two parts, namely the contract factory and the token exchange part. Among them, the Pistoken contract and pusd are mainly responsible for creating an independent trading contract for the two tokens PIS and PUSD tokens. The newpispool contract is responsible for providing the liquidity of the superconducting pool, as well as the functions of the superconducting pool for token exchange, handling fee processing and custom capital pools. Each trading contract will be associated with a USDT token,
And there is a liquidity pool of USDT and this PIS token to realize the exchange function of USDT and PIS and the exchange of PUSD and
For the exchange function between USDT or TRX, the handling fee generated during the exchange process is deposited in the synthetic pool.

Conclusion

1. When the real pishon project was designed, some of the code was redundant, and there was no strong factory model.

2. The project involves two funding pools, one is a superconducting pool and the other is an asset synthesis pool. The superconducting pool is completely safe. No address has permission to obtain funds from the superconducting pool

3． The entire pishon protocol consists of two tokens, one token is。PIS and the other token is PUSD. PIS is produced through superconducting computing power mining. After actual contract code review, there is no additional issuance without PIS, and the destroyed PIS will also enter the contract black hole address. PUSD is a token issued by mortgage. PUSD is 100% of assets synthesized by collateralizing USDT or TRX. Any user who wants to obtain PUSD must exchange it from the secondary market. Contract security

Result

Audit conclusion: passed

Audit Number: 202011200918B

Audit time: August 25, 2020

Audit Team: Trail of Bits Team

Audit summary: No security issues were found in this audit. After communication and feedback, it was confirmed that the risks found during the audit were within the acceptable range.

Trail of Bits

Since 2012, Trail of Bits has helped secure some of the world's most targeted organizations and products.We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code.

Home: https://www.trailofbits.com

Contact Us: https://www.trailofbits.com/contact/

Twitter: https://twitter.com/trailofbits

Linkedin: https://www.linkedin.com/company/trail-of-bits

Github: https://github.com/trailofbits

Blog：https://blog.trailofbits.com/