SMART CONTRACT AUDIT REPORT

for

Crossroad.gold

Prepared By: Yiqun Chen

PeckShield

June 21, 2021

Document Properties

Client	Djinn Gold
Title	Smart Contract Audit Report
Target	Crossroad.gold
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 21, 2021	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of Crossroad.gold, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Crossroad.gold

`Crossroad` is a `DEX` aggregator on the `Binance Smart Chain`. By connecting the liquidity of multiple exchanges, `Crossroad` is able to provide the best possible exchange rates. `Crossroad`'s current features include limit trades, OTC (over-the-counter) trades, and an unified liquidity aggregator.

The basic information of the Crossroad.gold is as follows:

Table 1.1: Basic Information of Crossroad.gold

Item	Description
Issuer	Djinn Gold
Website	https://www.crossroad.gold
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 21, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/donutcrypto/crossroad.git (550a8c3)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/donutcrypto/crossroad.git (1f4b533)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

		Likelihood		
Impact		*High*	*Medium*	*Low*
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of Crossroad.gold. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 1 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key Audit Findings of Crossroad.gold

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Refund of The Excess BNB	Time and State	Fixed
PVE-002	Medium	Possible Sandwich/MEV Attacks For Reduced Return	Time and State	Fixed
PVE-003	Informational	Redundant State/Code Removal	Coding Practice	Confirmed
PVE-004	Low	Gas Efficient Replacement of amountIn() and amountOut()	Coding Practice	Confirmed
PVE-005	High	Inconsistent Fee Rate Between PancakeSwap And DjinnAutoBuyer	Time and State	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Refund of The Excess BNB

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LiquidityProxy
- Category: Time and State [5]
- CWE subcategory: CWE-682 [3]

Description

In Crossroad.gold, there is a handy contract LiquidityProxy which provides a number of convenience routines for liquidation addition/removal, e.g., addLiquidity(), addLiquidityBnb(), removeLiquidity (), and removeLiquidityBnb(). During the analysis of these convenience routines, we notice that addLiquidityBnb() does not refund the excess BNB properly.

To elaborate, we show below the implementation of addLiquidityBnb(). This routine receives BNB and tokens from the caller, provides them to the _tokenLp pool as liquidity, and then refunds the excess BNB to the caller. Typically, the excess BNB is calculated by subtracting the amount of BNB consumed in the liquidity addition from the amount of BNB received from the caller.

```
    function addLiquidityBnb(
        address _factory,
        address _tokenLp,
        address _token,
        uint _amountTokenDesired,
        uint _amountBnbDesired,
        uint _amountTokenMin,
        uint _amountBnbMin,
        address _to
        ) external payable returns (uint amountToken_, uint amountBnb_, uint liquidity_)
    {
        require(_amountBnbDesired <= msg.value, "LiquidityProxy: Insufficient BNB");

        factoryEnsurePairExistsInner(_factory,_token,WBNB);

```

```
100         (amountToken_, amountBnb_) = getAddLiquidityAmountsInner(
101             _tokenLp,
102             _token,
103             WBNB,
104             _amountTokenDesired,
105             _amountBnbDesired,
106             _amountTokenMin,
107             _amountBnbMin
108             );
109         IERC20(_token).safeTransferFrom(msg.sender, _tokenLp, amountToken_);
110         IWETH(WBNB).deposit{value: amountBnb_}();
111         IERC20(WBNB).safeTransfer(_tokenLp, amountBnb_);
112         liquidity_ = IUniswapPool(_tokenLp).mint(_to);
113         // refund excess bnb
114         if (_amountBnbDesired > amountBnb_) TransferHelper.safeTransferETH(msg.sender,
                _amountBnbDesired - amountBnb_);
115     }
```

Listing 3.1: `LiquidityProxy::addLiquidityBnb()`

However, it comes to our attention that the current implementation, `amountBnbDesired - amountBnb_` (line 114), is subtracting the amount of `BNB` consumed in the liquidity addition from the amount of `BNB` desired by the caller. This will cause `msg.value - _amountBnbDesired` amount of `BNB` left in the contract.

Recommendation Calculate the excess `BNB` by subtracting `amountBnb_` from `msg.value`.

Status This issue has been fixed in the commit: 1f4b533.

3.2 Possible Sandwich/MEV Attack For Reduced Returns

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `DjinnAutoBuyer`
- Category: Time and State [5]
- CWE subcategory: CWE-682 [3]

Description

The `DjinnAutoBuyer` contract has a helper routine, i.e., `buyTokenFromBnb()`, that is designed to swap fee token `DJINN` by `BNB`. We show below the implementation of `buyTokenFromBnb()` routine from the `DjinnAutoBuyer` contract.

```
    function buyTokenFromBnb(
        address _outTarget
        ) external payable
    {
        uint256 _amountOutBusd = amountOut(lpWbnbBusd, true, msg.value,
            SWAP_PERMILLION_PCS2);
        uint256 _amountOutDjinn = amountOut(lpDjinnBusd, false, _amountOutBusd,
            SWAP_PERMILLION_PCS1);

        // execute swap and transfer djinn to sender
        IWETH(wbnbToken).deposit{value: msg.value}();
        IWETH(wbnbToken).transfer(lpWbnbBusd, msg.value);
        IUniswapPool(lpWbnbBusd).swap(0, _amountOutBusd, lpDjinnBusd, new bytes(0));
        IUniswapPool(lpDjinnBusd).swap(_amountOutDjinn, 0, _outTarget, new bytes(0));

        emit BoughtToken(_amountOutDjinn);
    }
```

Listing 3.2: `DjinnAutoBuyer::buyTokenFromBnb()`

We notice the token swap is routed to `pancakeSwap` and the actual swap operation `swap()` does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Improve the above function by adding necessary slippage control with the user-specified `amountOutMin`.

Status This issue has been fixed in the commit: 1f4b533.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

In Crossroad.gold, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. For example, the UNIT_ONE variable in Crossroad and LiquidityProxy, the wbnbToken variable in Crossroad, the MAX_VALUE variable in LiquidityProxy, and the payable modifier appended on addLiquidity. For better gas efficiency, we suggest removing the redundant state and the payable modifier.

```
    contract Crossroad
    {
        ...
        /* ======== CONSTANT VARIABLES ======== */

        // unit
        uint256 constant UNIT_ONE = 1e18;

        // tokens
        address public constant wbnbToken = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c;
        ...
```

Listing 3.3: Crossroad.sol

```
    contract LiquidityProxy
    {
        using SafeERC20 for IERC20;
        using Address for address;
        using SafeMath for uint256;

        /* ======== DATA STRUCTURES ======== */

        /* ======== CONSTANT VARIABLES ======== */

        // unit
        uint256 constant UNIT_ONE = 1e18;
        uint256 constant MAX_VALUE = 0
            xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;
        ...
```

Listing 3.4: LiquidityProxy.sol

```
    function addLiquidity(
        address _factory,
        address _tokenLp,
        address _tokenA,
        address _tokenB,
        uint _amountADesired,
        uint _amountBDesired,
        uint _amountAMin,
        uint _amountBMin,
        address _to
        ) external payable returns (uint amountA_, uint amountB_, uint liquidity_)
    {
        factoryEnsurePairExistsInner(_factory, _tokenA, _tokenB);

        (amountA_, amountB_) = getAddLiquidityAmountsInner(
            _tokenLp,
            _tokenA,
            _tokenB,
            _amountADesired,
            _amountBDesired,
            _amountAMin,
            _amountBMin
            );
        IERC20(_tokenA).safeTransferFrom(msg.sender, _tokenLp, amountA_);
        IERC20(_tokenB).safeTransferFrom(msg.sender, _tokenLp, amountB_);
        liquidity_ = IUniswapPool(_tokenLp).mint(_to);
    }
```

Listing 3.5: LiquidityProxy :: addLiquidity ()

Recommendation Remove the redundant state and the `payable` modifier.

Status This issue has been confirmed. Considering them part of convention, the team decides not to remove the redundant constant state. The redundant `payable` issue has been fixed in the commit: 1f4b533.

3.4 Gas Efficient Replacement of amountIn() and amountOut()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DjinnAutoBuyer
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

In DjinnAutoBuyer, the function amountIn()/amountOut() is defined to calculate the required output/input amount of an asset when given an input/output amount of the other asset. During the analysis of this function, we notice that there are more calculation steps of subtraction, multiplication, and division than the getAmountIn()/getAmountsOut() function from UniswapV2. We suggest replacing the implementation of amountIn()/amountOut() by getAmountIn()/getAmountsOut() for gas efficiency.

```
    function amountIn(
        address _pool,
        bool _forward,
        uint256 _amountOut,
        uint256 _swapPerMillionRate
        ) public view returns (uint256)
    {
        uint256 _initReserveIn;
        uint256 _initReserveOut;
        if (_forward)
        {
            (_initReserveIn,_initReserveOut,) = IUniswapPool(_pool).getReserves();
        }
        else
        {
            (_initReserveOut,_initReserveIn,) = IUniswapPool(_pool).getReserves();
        }

        uint256 _initBalanceIn = _initReserveIn.mul(1000000);
        uint256 _initBalanceOut = _initReserveOut.mul(1000000);

        uint256 _initProduct = _initBalanceIn.mul(_initBalanceOut);

        uint256 _finiBalanceOut = _initBalanceOut.sub(_amountOut.mul(1000000));
        uint256 _finiBalanceIn = _initProduct.div(_finiBalanceOut);

        return _finiBalanceIn.sub(_initBalanceIn).div(_swapPerMillionRate).add(1); // add 1 to account for rounding
    }
```

Listing 3.6: DjinnAutoBuyer::amountIn()

```
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure
        returns (uint amountIn) {
        require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
            INSUFFICIENT_LIQUIDITY');
        uint numerator = reserveIn.mul(amountOut).mul(1000);
        uint denominator = reserveOut.sub(amountOut).mul(997);
        amountIn = (numerator / denominator).add(1);
    }
```

Listing 3.7: `UniswapV2Library::getAmountIn()`

Recommendation Replace the above `amountIn()`/`amountOut()` functions by the `UniswapV2` implementation of `getAmountIn()`/`getAmountsOut()`.

Status This issue has been confirmed. Considering current approach may only cost a small number of extra gas, the team decides to leave it as is.

3.5 Inconsistent Fee Rate Between PancakeSwap And DjinnAutoBuyer

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High

- Target: `DjinnAutoBuyer`
- Category: Time and State [5]
- CWE subcategory: CWE-682 [3]

Description

As mentioned in Section 3.2, the `DjinnAutoBuyer` contract has a helper routine, i.e., `buyTokenFixed()`, that is designed to swap tokens. It takes `BNB` and uses `PancakeSwap`'s `BUSD-WBNB` and `DJINN-BUSD` pairs to swap the fee token `DJINN`. However, it comes to our attention that the fee rate using in the `PancakeSwap` `BUSD-WBNB` pair (`lpWbnbBusd` : 0.2%) is different from the fee rate using in `DjinnAutoBuyer` (`SWAP_PERMILLION_PCS2` : 0.25%). The inconsistent fee rate between `PancakeSwap` and `DjinnAutoBuyer` will lead to more `BNB` spend for the same amount of fee tokens.

```
    contract DjinnAutoBuyer
    {
        using SafeERC20 for IERC20;
        using Address for address;
        using SafeMath for uint256;

        /* ======== DATA STRUCTURES ======== */
        /* ======== CONSTANT VARIABLES ======== */

        // swap contracts
        address public constant lpDjinnBusd = 0x03962E1907B0FA72768Bd865e8cA0C45C7De4937
            ;
        address public constant lpWbnbBusd = 0x1B96B92314C44b159149f7E0303511fB2Fc4774f;
        address public constant wbnbToken = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c;

        // factor for swap fees
        uint256 public constant SWAP_PERMILLION_PCS1 = 998000;
        uint256 public constant SWAP_PERMILLION_PCS2 = 997500;
        ...
```

Listing 3.8: `DjinnAutoBuyer.sol`

```
    function buyTokenFixed(
        uint256 _amountOut,
        address _outTarget,
        address _refundTarget
        ) external payable
        returns (uint256)
    {
        uint256 _amountInBusd = amountIn(lpDjinnBusd, false, _amountOut,
            SWAP_PERMILLION_PCS1);
        uint256 _amountInWbnb = amountIn(lpWbnbBusd, true, _amountInBusd,
            SWAP_PERMILLION_PCS2);
        ...
    }
```

Listing 3.9: `DjinnAutoBuyer::buyTokenFixed()`

Note another function `buyTokenFromBnb()` from the same contract shares the same issue.

Recommendation We suggest to make the `BUSD-WBNB` fee rate consistent between `PancakeSwap` and `DjinnAutoBuyer`.

Status This issue has been fixed in the commit: 1f4b533.

4 Conclusion

In this audit, we have analyzed the design and implementation of Crossroad.gold. Current features of `Crossroad` include limit trades, OTC (over-the-counter) trades, and an unified liquidity aggregator. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.