SMART CONTRACT AUDIT REPORT

for

BSD DeFi

Prepared By: Shuxiao Wang

PeckShield

May 29, 2021

Document Properties

Client	BSD Finance
Title	Smart Contract Audit Report
Target	BSD DeFi
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 29, 2021	Yiqun Chen	Final Release
0.1	May 23, 2021	Yiqun Chen	First Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the BSD protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, 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 BSD DeFi

BSD DeFi is a financial ecosystem that unifies the current scattered DeFi (Decentralized Finance) landscape. It is a system of finance that utilizes protocols, digital assets, smart contracts, and decentralized applications (dApps) on Heco and Tron to build a financial platform that's open to everyone. The BSD team aims to provide a transparent, decentralized, and high-security platform for users to maximize their investment/loans returns with minimal efforts.

The basic information of the BSD DeFi is as follows:

Table 1.1: Basic Information of The BSD Protocol

Item	Description
Issuer	BSD Finance
Type	Heco and Tron Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 29, 2021

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

- https://github.com/BSD-DeFi/Stable-Coin (15240cb)

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

- https://github.com/BSD-DeFi/Stable-Coin (1438d05)

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

Impact \ Likelihood	*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 classified into four categories accordingly, 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 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

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

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.

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 BSD DeFi implementation. 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	1	■
High	0	
Medium	1	■
Low	1	■
Informational	1	■
Total	4	

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 critical-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational suggestion.

Table 2.1: Key BSD DeFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-002	Medium	Suggested Use Of Safemath For claim()	Coding Practices	Confirmed
PVE-003	Critical	Lack Of Authentication For Privilege Functions	Coding Practices	Fixed
PVE-004	Informational	Logic Error Of burn()	Coding Practices	Confirmed

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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `POLMain`
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. It is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function mint(uint256 amount) public{
            address owner = msg.sender;
            uint256 mintedPUSD = amount.mul(getPOLPrice()).div(getStakeRate()).div(1000000);
            // _minted[owner] = _minted[owner].add(mintedPUSD);
            setMinted(owner,getMinted(owner).add(mintedPUSD));
            uint256 bonusPNX = mintedPUSD.mul(1000000).mul(getBonusRate()).div(100).div(getPNXPrice());
            // _bonus[owner] = _bonus[owner].add(bonusPNX);
            setBonus(owner,getBonus(owner).add(bonusPNX));
            // _staked[owner] = _staked[owner].add(amount);
            setStaked(owner,getStaked(owner).add(amount));
            polToken.transferFrom(owner,polPoolAddr,amount);
            // pusdToken.transferFrom(pusdPoolAddr,owner,mintedPUSD);
            transferPUSDTo(owner,mintedPUSD);
            // pnxToken.transferFrom(pnxPoolAddr,pnxOfficialAddress,bonusPNX*15/100);
            transferPNXTo(pnxOfficialAddress,bonusPNX.mul(15).div(85));
```

```
    }
```

Listing 3.1: POLMain::mint()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()`/`transferFrom()`.

Status This issue has been confirmed.

3.2 Suggested Use Of nonReentrant In mint()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `POLMain`
- Category: Coding Practices [5]
- CWE subcategory: CWE-190 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once.

We notice there are several occasions the `checks-effects-interactions` principle is violated. For example, the `mint()` function (see the code snippet below) is provided to externally call several token contracts to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

```
    function mint(uint256 amount) public{
            address owner = msg.sender;
            uint256 mintedPUSD = amount.mul(getPOLPrice()).div(getStakeRate()).div
                (1000000);
            // _minted[owner] = _minted[owner].add(mintedPUSD);
            setMinted(owner,getMinted(owner).add(mintedPUSD));
            uint256 bonusPNX = mintedPUSD.mul(1000000).mul(getBonusRate()).div(100).div(
                getPNXPrice());
```

```
84            // _bonus[owner] = _bonus[owner].add(bonusPNX);
85            setBonus(owner, getBonus(owner).add(bonusPNX));
86            // _staked[owner] = _staked[owner].add(amount);
87            setStaked(owner, getStaked(owner).add(amount));
88            polToken.transferFrom(owner, polPoolAddr, amount);
89            // pusdToken.transferFrom(pusdPoolAddr,owner,mintedPUSD);
90            transferPUSDTo(owner, mintedPUSD);
91            // pnxToken.transferFrom(pnxPoolAddr,pnxOfficialAddress,bonusPNX*15/100);
92            transferPNXTo(pnxOfficialAddress, bonusPNX.mul(15).div(85));
93    }
```

Listing 3.2: POLMain::mint()

Apparently, the interaction with the external contract (line 88) starts before effecting the update on internal states (line 82), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `mint()` function.

Recommendation Add the `nonReentrant` modifier to prevent reentrancy.

Status This issue has been confirmed.

3.3 Lack Of Authentication For Privilege Functions

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `PolDataMain`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `BSD` protocol, the `owner` account plays a critical role in governing and regulating the entire operation and maintenance (e.g., price setting). It also has the privilege to transfer funds from the `polPoolAddr`, the `pusdPoolAddr`, and the `pnxPoolAddr` to any addresses.

To elaborate, we show below the related `setPOLPrice()` function.

```
    function setPOLPrice(uint256 amount) public {
        _polPrice = amount;
    }
```

Listing 3.3: PolDataMain::setPOLPrice()

As we can see in the function above, this function is defined with a `public` modifier. The same issue is also present for the `setMinted()`, `setStaked()`, `setBonus()`, `setBonusAvailable()`, `setWaitingBurn()`, `setBonusRate()`, `setStakeRate()`, `setPUSDPrice()`, `setPNXPrice()`, `transferPOLTo()`, `transferPUSDTo()`, and `transferPNXTo()` functions in the `PolDataMain` contract. Besides, the `transferPOLTo()`, `transferPUSDTo()`, and `transferPNXTo()` functions in the `PolMain` contract also have the same issue. These privileged functions require proper authentication, which is currently missing. Malicious users are able to call some of these functions to change these important settings and may further transfer all the funds in `polPoolAddr`, `pusdPoolAddr`, and `pnxPoolAddr` to their own accounts.

Recommendation Add necessary authentication to the functions mentioned above, e.g., the `onlyOwner` modifier.

Status The issue has been addressed by the following commit: `1438d05`.

3.4 Logic Error Of burn()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `POLMain`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1188 [2]

Description

The `BSD` protocol allows users to `burn` their `pusdTokens` to gain back their `polTokens`. The logic behind this whole process is that the user transfers his/her `pusdTokens` to `pusdPoolAddr`, then transfers enough `pnxTokens` to `pnxTrashAddress`, and finally gets a certain amount of `polTokens` from the `PolDataMain` contract.

```
    function burn(uint256 amount) public {
        address owner = msg.sender;
        require(amount <= getMinted(owner), 'You don\'t have enough PUSD to burn');
            // if (_bonusAvailable[owner] >= amount/100*_bonusRate){
            // _minted[owner] = _minted[owner] - amount;
            // _staked[owner] = _staked[owner] - amount/(_stakeRate / price);
            // _bonusAvailable[owner] = _bonusAvailable[owner] - amount/100*_bonusRate;
            // // _waitingBurn[owner] = _waitingBurn[owner] - amount;
            // polToken.transferFrom(polPoolAddr,owner,amount / (_stakeRate / price));
            // }
            // else{
                pusdToken.transferFrom(owner,pusdPoolAddr,amount);
                // _waitingBurn[owner] = _waitingBurn[owner].add(amount);
                setWaitingBurn(owner,getWaitingBurn(owner).add(amount));
            // }
    }
```

Listing 3.4: POLMain::burn()

```
112     function receivePnxToBurn(uint256 amount) public {
113         // require(amount <= _minted[owner], 'You don\'t have enough PUSD to burn');
114             // _bonusAvailable[owner] = _bonusAvailable[owner] + amount;
115             address owner = msg.sender;
116             require(amount.mul(100).mul(100).div(getBonusRate()) >= getWaitingBurn(owner
                    ), 'You need transfer more PNX to burn');
117             pnxToken.transferFrom(owner,pnxTrashAddress,amount);
118             // uint256 amountToBurn = _waitingBurn[owner];
119             uint256 amountToBurn = getWaitingBurn(owner);
120             uint256 amountPOL = amountToBurn.mul(getStakeRate()).mul(1000000).div(
                    getPOLPrice());
121             // // _minted[owner] = _minted[owner].sub(amountToBurn);
122             setMinted(owner,getMinted(owner).sub(amountToBurn));
123             // // _staked[owner] = _staked[owner].sub(amountPOL);
124             setStaked(owner,getStaked(owner).sub(amountPOL));
125             // // _bonusAvailable[owner] = _bonusAvailable[owner] - amountToBurn/100*
                    _bonusRate;
126             // // _waitingBurn[owner] = 0;
127             setWaitingBurn(owner,0);
128             transferPOLTo(owner,amountPOL);
129     }
```

Listing 3.5: POLMain::receivePnxToBurn()

To elaborate, we show above the `burn()` and the `receivePnxToBurn()` functions. In the `burn()` function, it firstly validates whether the `msg.sender` has enough `pusdTokens` by calling the `getMinted()` function. However, after the `pusdTokens` sent to the `pusdPoolAddr` , the result of `getMinted()` does not change. The user can pass the check of `amount <= getMinted(owner)` (line 97) even he/she does not have enough tokens before he/she calls the `receivePnxToBurn()` function.

Recommendation Move the statements of `setMinted(owner,getMinted(owner).sub(amountToBurn))` and `uint256 amountToBurn = getWaitingBurn(owner)` to the `burn()` function.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `BSD` protocol. The audited contract allows the user to exchange between different tokens(`pusdToken`, `polToken`) through the `mint()` function, the `burn()` function, and the `receivePnxToBurn()` function. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based 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: Lack Of Authentication For Privilege Functions. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1188: Logic Error. https://cwe.mitre.org/data/definitions/1188.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[4] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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