Smart Contract Audit of dForce USDx (v1.3)

PeckShield Inc.

2019/08/01

Contents

1. Introduction

We (**PeckShield** [1]) were entrusted to review the smart contract source code of **dForce USDx** project. In this report, we outline our systematic method 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 contract code is secure, as no serious vulnerabilities had been discovered so far. However, there still exists several security-sensitive issues that need to be fixed in the revision. This document outlines our audit results.

1.1 About USDx

The basic information of USDx is as follows:

Name	Description
Issuer	dForce Network
DApp Name	USDx
Audit Completion Date	2019-08-01

Table 1: Basic Information of USDx

In the following, we show the list of reviewed files and their checksum or version:

- GitHub: https://github.com/dforce-network/USDXProtocol.git
- commit: 7a75d7b4a3aae72974c20c88f4d6b4061fb9002b

1.2 About PeckShield

PeckShield Inc. [1] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at: telegram, twitter, or email[1].

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [2]:

- Likelihood: represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

[1] Telegram: https://t.me/peckshield;
Twitter: http://twitter.com/peckshield;
Email: contact@peckshield.com

- 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, as shown in Table 2.

Impact	Likelihood		
	H	M	L
H	Critical	High	Medium
M	High	Medium	Low
L	Medium	Low	Low

Table 2: Overall Severity

We perform the audit according to the following procedures:

- Checked Vulnerabilities: We first statically analyze given smart contracts with our proprietary static code analyzer, and then manually verify (reject or confirm) all the issues found by our tools.
- Attacking Tests
  - Known Vulnerabilities Attacking Test
  - Business Logic Attacking Test
- Source Code Security Auditing
  - Coding Style and Common Security Issues Check
  - Business Logic Review
  - Functionality Check
  - Authentication Management Review
  - Oracle Security Review
  - Digital Asset Security Review
  - Semantic Consistency Check
  - Deployment Consistency Check
  - Kill-Switch Implementation Review

- Additional Recommendations: Providing additional suggestions regarding the coding and development of smart contract from the perspective of proven programming practice.

1.4 Disclaimer

Please note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit 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 is not an investment advice.

2. Findings

2.1 Key Findings

As of this writing, we have identified 1 medium vulnerability, and 4 low severity vulnerabilities. Besides that, we also provide 14 informational recommendations, as shown in Table 3.

Severity	# of Findings	Type	Description	Status
Medium	1	Business Logic	Flawed Wrapper Mechanism	Fixed
Low	4	Business Logic	Compatibility Issue with non-ERC20 Tokens	Fixed
	8	Recommendation	Inconsistency in Function and Interface Declarations	Fixed
	1	Recommendation	Improvement	Fixed
	1	Recommendation	Lack of Interface Declaration	Fixed
	2	Recommendation	Improvement	Confirmed
	1	Auth Mgmt	Lack of Auth Setting	Fixed
	1	Kill Switch	Lack of Kill Switch Impl	Confirmed

Table 3: Key Findings

3. Detailed Results

3.1 Flawed Wrapper Mechanism

- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Type: Business Logic Flaw
- Description

In contracts/converter/DFEngine.sol:66, a user deposit() _srcAmount of _srcToken which is immediately transferred to dfPool when _srcAmount > 0 and _srcToken is valid. However, in contracts/converter/DFEngine.sol:67, _srcAmount is converted into _amount by the wrapper mechanism. Later on, the number _amount represents the amount that the user deposited into the USDx system. In some cases, the USDx system would count less _srcToken than the amount transferred to dfPool, which is unfair to users.

```
61    function deposit(address _depositor, address _srcToken, uint _feeTokenIdx, uint _srcAmount) public auth returns (uint) {
62        require(_srcAmount > 0, "Deposit: amount not allow.");
63        address _tokenID = dfStore.getWrappedToken(_srcToken);
64        require(dfStore.getMintingToken(_tokenID), "Deposit: asset not allow.");
65
66        dfPool.transferFromSender(_srcToken, _depositor, _srcAmount);
67        uint _amount = IDSWrappedToken(_tokenID).wrap(address(dfPool), _srcAmount);
68        _unifiedCommission(ProcessType.CT_DEPOSIT, _feeTokenIdx, _depositor, _amount);
```

For example, if the decimal of _srcToken is 19 and 1009 is deposited into the USDx system. According to contracts/token/DSWrappedToken.sol, the _amount would be calculated as following:

_amount = 1009 / 10**(19-18) = 1009 / 10 = 100

However, 1009 of *_srcToken* has been transferred to dfPool but the USDx system only takes 100 wrapped *_srcToken* into account, which is equivalent to 1,000 *_srcToken*, making 9 *_srcToken* shortage from the user side.

Similar logic is implemented in withdraw(), we suggest the developers fix both of them.

- Exploit Scenario

As shown in the above description, when the USDx system wraps a token T which has decimal = 19, only 1000T of the 1009T would be taken into account, 9T would be taken by dfPool for no reason.

- Recommendation

Calculate _amount earlier and only transfer _amount of _srcToken to dfPool. This would also be an optimization chance to filter out _amount <= 0 case before anything.

3.2 Compatibility Issue with Non-ERC20 Tokens

- Severity: Low
- Likelihood: Low
- Impact: Low
- Type: Business Logic Flaw
- Description

In storage/DFCollateral.sol:16, the transferOut() function invokes the transfer() interface of the *_tokenID* token for transferring *_amount* of tokens to *_to*. In the meantime, assert() is used to check the return value of transfer().

```
 7 contract DFCollateral is DSAuth, Utils {
 8 
 9     function transferOut(address _tokenID, address _to, uint _amount)
10         public
11         validAddress(_to)
12         auth
13         returns (bool)
14     {
15         require(_to != address(0), "TransferOut: 0 address not allow.");
16         assert(IERC20Token(_tokenID).transfer(_to, _amount));
17         return true;
18     }
19 }
```

However, not all of the token contracts follow the ERC20 standard. A token contract may return nothing in transfer() or even has an opposite logic against the standard implementation.

- Exploit Scenario

USDx adopts a non-ERC20 token in which the transfer() returns nothing or wrong value.

- Recommendation

1) Strictly check the token contract and make sure that it complies to the ERC20 standard.
2) Remove the assert() and let the transfer() handler in the token contract to deal with the failed cases.

3.3 Compatibility Issue with Non-ERC20 Tokens

- Severity: Low
- Likelihood: Low
- Impact: Low
- Type: Business Logic Flaw
- Description

In storage/DFFunds.sol:16, the transferOut() function invokes the transfer() interface of the *_tokenID* token for transferring *_amount* of tokens to *_to*. In the meantime, assert() is used to check the return value of transfer().

```
 9     function transferOut(address _tokenID, address _to, uint _amount)
10         public
11         validAddress(_to)
12         auth
13         returns (bool)
14     {
15         require(_to != address(0), "TransferOut: 0 address not allow.");
16         assert(IERC20Token(_tokenID).transfer(_to, _amount));
17         return true;
18     }
19 }
```

However, not all of the token contracts follow the ERC20 standard. A token contract may return nothing in transfer() or even has an opposite logic against the standard implementation.

- Exploit Scenario

USDx adopts a non-ERC20 token in which the transfer() returns nothing or wrong value.

- Recommendation

1) Strictly check the token contract and make sure that it complies to the ERC20 standard.
2) Remove the assert() and let the transfer() handler in the token contract to deal with the failed cases.

3.4 Compatibility Issue with Non-ERC20 Tokens

- Severity: Low
- Likelihood: Low
- Impact: Low
- Type: Business Logic Flaw
- Description

In storage/DFPool:20, the transferFromSender() function invokes the transferFrom() interface of the *_tokenID* token for transferring *_amount* of tokens. In the meantime, assert() is used to check the return value of transferFrom(). Besides, in storage/DFPool:30 and storage/DFPool:40, the transfer() interface of the *_tokenID* token is invoked respectively with assert() for checking the return values.

```
15      function transferFromSender(address _tokenID, address _from, uint _amount)
16          public
17          auth
18          returns (bool)
19      {
20          assert(IERC20Token(_tokenID).transferFrom(_from, address(this), _amount));
21          return true;
22      }
23
24      function transferOut(address _tokenID, address _to, uint _amount)
25          public
26          validAddress(_to)
27          auth
28          returns (bool)
29      {
30          assert(IERC20Token(_tokenID).transfer(_to, _amount));
31          return true;
32      }
33
34      function transferToCol(address _tokenID, uint _amount)
35          public
36          auth
37          returns (bool)
38      {
39          require(dfcol != address(0), "TransferToCol: collateral address empty.");
40          assert(IERC20Token(_tokenID).transfer(dfcol, _amount));
41          return true;
42      }
```

However, not all of the token contracts follow the ERC20 standard. A token contract may return nothing in transfer() or even has an opposite logic against the standard implementation.

- Exploit Scenario

USDx adopts a non-ERC20 token in which the transfer() returns nothing or wrong value.

- Recommendation

1) Strictly check the token contract and make sure that it complies to the ERC20 standard.
2) Remove the assert() and let the transfer() and transferFrom() handlers in the token contract to deal with the failed cases.

3.5 Compatibility Issue with Non-ERC20 Tokens

- Severity: Low
- Likelihood: Low
- Impact: Low
- Type: Business Logic Flaw
- Description

In storage/DFPool:58, the transferFromSenderToCol() function invokes the transferFrom() interface of the *_tokenID* token for transferring *_amount* of tokens. In the meantime, assert() is used to check the return value of transferFrom().

```
51    function transferFromSenderToCol(address _tokenID, address _from, uint _amount)
52        public
53        auth
54        returns (bool)
55    {
56        require(dfcol != address(0), "TransferFromSenderToCol: collateral address empty.");
57        uint _balance = IERC20Token(_tokenID).balanceOf(dfcol);
58        assert(IERC20Token(_tokenID).transferFrom(_from, dfcol, _amount));
59        assert(sub(IERC20Token(_tokenID).balanceOf(dfcol), _balance) == _amount);
60        return true;
61    }
```

- Exploit Scenario

USDx adopts a non-ERC20 token in which the transfer() returns nothing or wrong value.

- Recommendation

1) Strictly check the token contract and make sure that it complies to the ERC20 standard.
2) Remove the assert() and let the transferFrom() handlers in the token contract to deal with the failed cases.

3.6 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getDepositMaxMint() in converter/interfaces/IDFEngine.sol:8 is not consistent to the function declaration in converter/DFEngine.sol:317.

converter/interfaces/IDFEngine.sol:8

```
function getDepositMaxMint(address _depositor, address _tokenID, uint _amount)
public returns (uint);
```

converter/DFEngine.sol:317

```
function getDepositMaxMint(address _depositor, address _tokenID, uint _amount)
public view returns (uint)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.7 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getMaxToClaim() in converter/interfaces/IDFEngine.sol:9 is not consistent to the function declaration in converter/DFEngine.sol:352.

converter/interfaces/IDFEngine.sol:9

```
function getMaxToClaim(address _depositor) public returns (uint);
```

converter/DFEngine.sol:352

```
function getMaxToClaim(address _depositor) public view returns (uint)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.8 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getCollateralMaxClaim() in converter/interfaces/IDFEngine.sol:10 is not consistent to the function declaration in converter/DFEngine.sol:370.

converter/interfaces/IDFEngine.sol:10

```
function getCollateralMaxClaim() public returns (address[] memory, uint[] memory);
```

converter/DFEngine.sol:370

```
function getCollateralMaxClaim() public view returns (address[] memory, uint[] memory)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.9 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getMintingSection() in converter/interfaces/IDFEngine.sol:11 is not consistent to the function declaration in converter/DFEngine.sol:383.

converter/interfaces/IDFEngine.sol:11

```
function getMintingSection() public returns(address[] memory, uint[] memory);
```

converter/DFEngine.sol:383

```
function getMintingSection() public view returns(address[] memory, uint[] memory)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.10 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getBurningSection() in converter/interfaces/IDFEngine.sol:12 is not consistent to the function declaration in converter/DFEngine.sol:391.

converter/interfaces/IDFEngine.sol:12

```
function getBurningSection() public returns(address[] memory, uint[] memory);
```

converter/DFEngine.sol:391

```
function getBurningSection() public view returns(address[] memory, uint[] memory)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.11 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getWithdrawBalances() in converter/interfaces/IDFEngine.sol:13 is not consistent to the function declaration in converter/DFEngine.sol:399.

converter/interfaces/IDFEngine.sol:13

```
function getWithdrawBalances(address _depositor) public returns(address[] memory, uint[] memory);
```

converter/DFEngine.sol:399

```
function getWithdrawBalances(address _depositor) public view returns(address[] memory, uint[] memory)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.12 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getPrices() in converter/interfaces/IDFEngine.sol:14 is not consistent to the function declaration in converter/DFEngine.sol:411.

converter/interfaces/IDFEngine.sol:14

```
function getPrices(uint typeID) public returns (uint);
```

converter/DFEngine.sol:411

```
function getPrices(uint typeID) public view returns (uint)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.13 Inconsistency in Function and Interface Declarations

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The interface declaration of getFeeRateByID() in converter/interfaces/IDFEngine.sol:15 is not consistent to the function declaration in converter/DFEngine.sol:419.

converter/interfaces/IDFEngine.sol:15

```
function getFeeRateByID(uint typeID) public returns (uint);
```

converter/DFEngine.sol:419

```
function getFeeRateByID(uint typeID) public view returns (uint)
```

- Exploit Scenario

N/A

- Recommendation

Make them consistent by adding view.

3.14 Suggested Improvements

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The following functions access storage for read only. The modifier *view* could be added as a hint for the compiler.

converter/DFProtocol.sol:35: function getUSDXForDeposit()

converter/DFProtocol.sol:39: function getUserMaxToClaim()

converter/DFProtocol.sol:43: function getColMaxClaim()

converter/DFProtocol.sol:47: function getMintingSection()

converter/DFProtocol.sol:51: function getBurningSection()

converter/DFProtocol.sol:55: function getUserWithdrawBalance()

converter/DFProtocol.sol:59: function getPrice(uint typeID)

converter/DFProtocol.sol:63: function getFeeRate(uint typeID)

- Exploit Scenario

N/A

- Recommendation

Add *view* in the function declarations.

3.15 Suggested Improvements

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

In the implementations of *approveToEngine*(), contracts/storage/DFPool.sol:54 and contracts/storage/DFCollateral.sol:20, the ERC20 standard function, *allowance*() can be used to check if the *approve*() operation is succeeded.

```
    function approveToEngine(address _tokenIdx, address _engineAddress) public auth {
        IERC20Token(_tokenIdx).approve(_engineAddress, uint(-1));
    }
```

```
    function approveToEngine(address _tokenIdx, address _engineAddress) public auth {
        IERC20Token(_tokenIdx).approve(_engineAddress, uint(-1));
    }
```

- Exploit Scenario

N/A

- Recommendation

Add *allowance()* checks.

3.16 Suggested Improvements

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

In the implementation of oneClickMinting(), the input parameter, _amount, is not checked anywhere in the function.

```
225     function oneClickMinting(address _depositor, uint _feeTokenIdx, uint _amount) public auth {
226         address[] memory _tokens;
227         uint[] memory _mintCW;
228         uint _sumMintCW;
229         uint _srcAmount;
230
231         (, , , _tokens, _mintCW) = dfStore.getSectionData(dfStore.getMintPosition());
232         for (uint i = 0; i < _mintCW.length; i++) {
233             _sumMintCW = add(_sumMintCW, _mintCW[i]);
234         }
235         require(_sumMintCW != 0, "OneClickMinting: minting section is empty");
236         require(_amount % _sumMintCW == 0, "OneClickMinting: amount error");
237
238         _unifiedCommission(ProcessType.CT_DEPOSIT, _feeTokenIdx, _depositor, _amount);
239
240         for (uint i = 0; i < _mintCW.length; i++) {
241
242             _srcAmount = IDSWrappedToken(_tokens[i]).reverseByMultiple(div(mul(_amount, _mintCW[i]), _sumMintCW));
243             dfPool.transferFromSender(IDSWrappedToken(_tokens[i]).getSrcERC20(), _depositor, _srcAmount);
244             dfStore.setTotalCol(add(dfStore.getTotalCol(), div(mul(_amount, _mintCW[i]), _sumMintCW)));
245             IDSWrappedToken(_tokens[i]).wrap(dfCol, _srcAmount);
246         }
```

However, if the user who invokes oneClickMinting() does not have enough tokens to mint USDx, the function reverts in line 243 while transferring tokens to dfPool, which is a waste of gas.

An optimization choice is checking the balance of each token before performing the minting logic. The ERC20 standard function, balanceOf(), could be used for the preliminary checks.

- Exploit Scenario

N/A

- Recommendation

Add *balanceOf()* checks. If the caller does not have enough tokens, invokes *revert*() earlier.

3.17 Lack of Interface Declaration

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Improvement
- Description

The public function *oneClickMinting*() defined in contracts/converter/DFProtocol.sol:36 is missed in IDFProtocol.

```
contract IDFProtocol {
    function deposit(address _tokenID, uint _feeTokenIdx, uint _amount) public returns (uint _balance);
    function withdraw(address _tokenID, uint _feeTokenIdx, uint _amount) public returns (uint _balance);
    function destroy(uint _feeTokenIdx, uint _amount) public;
    function claim(uint _feeTokenIdx) public returns (uint _balance);
}
```

- Exploit Scenario

N/A

- Recommendation

Add *oneClickMinting*() interface declaration in IDFProtocol.

3.18 Lack of Authentication Setting

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Authentication Management
- Description

In migrations/3_after_deploy.js, compared to other contracts, the “Setting” contract is not setAuthority() to the guard.

- Exploit Scenario

N/A

- Recommendation

Apply setAuthority() to “Setting” contract.

3.19 Lack of Kill Switch Implementation

- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Type: Kill Switch
- Description

There is no kill switch implementation throughout the USDx system.

- Exploit Scenario

N/A

- Recommendation

Add kill switch mechanism into the design and implementation. For example, a *require(live == 1)* check could be added in the entries of critical functions with an interface for setting the *live* variable.

4. Conclusion

The USDx smart contact was analyzed in this audit. The wrapper mechanism logic error and the compatibility issues with non-ERC20 tokens were identified. Fortunately, all the outstanding vulnerabilities we identified were fixed.

As disclaimed in Section 1.4, this audit does not give any warranties on finding all possible security issues of the given smart contract, and cannot be regarded as an investment advice.

5. Reference

1. PeckShield. PeckShield Inc. https://www.peckshield.com

2. OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.