dForce USDx
安全审计报告(v1.3)

PeckShield Inc.

2019/08/01

目录

1. 介绍

2. 检测结果概要

3. 检测结果详情

4. 结论

5. 参考文献

1. 介绍

我们 (PeckShield [1]) 对 dForce Network 的 USDx 项目合约代码进行了安全审计。根据我们的安全审计规范和客户需求，我们将在报告中列出用于检测潜在安全问题的系统性方法，并根据检测结果给出相应的建议或推荐以修复或改善代码质量。分析结果表明，该特定版本的智能合约代码存在拒绝服务漏洞，在升级时的代码注入风险，以及逻辑错误导致某功能无法使用，需要予以关注并及时修复。修复的方式请参考第3章检测结果详情部分的相关内容。本文档对审计结果作了分析和阐述。

1.1 关于USDx

USDx 是基于以太坊开发的 DeFi 项目，基本信息如下：

名称	描述
开发与运营	dForce Network
DApp 名称	USDx
审计完成时间	2019-08-01

表格 1 项目方的基本信息

以下是审计对象（即智能合约）相关信息:

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

1.2 关于 PeckShield

PeckShield (派盾) 是面向全球的业内顶尖区块链安全团队，以提升区块链生态整体的安全性、隐私性以及可用性为己任，通过发布行业趋势报告、实时监测生态安全风险，负责任曝光 0day 漏洞，以及提供相关的安全解决方案和服务等方式帮助社区抵御新兴的安全威胁。可以通过下列联系方式联络我们: telegram, twitter, or email[1].

1.3 评估方法和模型

为了检测评估的标准化，我们根据 OWASP Risk Rating Methodology[2] 定义下列术语:

- 可能性：表示某个特定的漏洞被发现和利用的可能性；
- 影响力：度量了（利用该漏洞的）一次成功的攻击行动造成的损失；
- 危害性：显示该漏洞的危害的严重程度。

可能性和影响力各自被分为三个等级: 高、中和低。危害性由可能性和影响力确定， 分为四个等级：严重、高危、中危、低危，如表格 2 所示。

影响力	可能性		
	高	中	低
高	严重	高危	中危
中	高危	中危	低危
低	中危	低危	低危

表格 2 危害性定义

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

我们整理了常见或具备一定危害性的检测项，并按照如下流程进行审计：

- 漏洞静态扫描：首先以自研的自动化静态检测工具侦测常见代码级别漏洞，而后针对代码实现以人工审查方式检测；
- 动态攻击测试：
  - 已知漏洞攻击测试：对可能存在的已知漏洞进行攻击验证；
  - 业务逻辑攻击测试：对代码分析中可能存在的风险进行攻击验证；
- 代码安全审计：
  - 白盒漏洞检测：分析代码，确认是否存在已知漏洞；
  - 业务逻辑分析：结合业务逻辑分析代码，寻找业务逻辑上是否存在安全风险；
  - 功能验证：验证合约代码基本功能是否正常；
  - 权限管理：对外功能及接口是否有权限管理及校验；
  - 预言机安全：验证预言机实现是否存在安全风险；
  - 资产安全：分析合约中资产流向是否存在安全风险；
  - 语义一致性检查：人工确认智能合约的实现与白皮书的描述或线上应用是否一致；
  - 合约部署一致性验证：验证智能合约代码与线上已部署合约是否一致；
  - 熔断机制：验证熔断机制实现是否存在安全风险；
  - 其它建议：从业已被证明的良好开发实践出发，针对智能合约代码的编写和部署给出建议或者意见。

1.4 免责声明

请注意该审计报告并不保证能够发现该智能合约中存在的一切安全问题，即评估结果并不能保证在未来不会发现新的安全问题。我们一向认为单次审计结果可能并不全面，因而推荐采取多个独立的审计和公开的漏洞奖赏计划相结合的方式来确保合约的安全性。最后必须要强调的是，审计结果仅针对智能合约代码的安全性，不构成任何投资建议。

2. 检测结果概要

2.1 主要发现

在本次审计中，我们发现 1 个中危漏洞，以及 4 个低危漏洞。此外，我们也提出 14 个修改建议。情况如 表格 3 所示：

危害性	数量	类型	描述	状态
中危	1	业务逻辑风险	wrapper 机制实现问题造成用户损失少量成分币	已修复
低危	4	业务逻辑风险	不符合 ERC20 规范的成分币兼容问题	已修复
	8	代码建议与优化	接口与声明不一致问题	已修复
	1	代码建议与优化	优化建议	已修复
	1	代码建议与优化	接口声明缺失	已修复
	2	代码建议与优化	优化建议	已确认
	1	权限管理	某处权限未设置	已修复
	1	熔断机制	未实现熔断机制	已确认

表格 3 主要发现

3. 检测结果详情

3.1 WRAPPER 机制实现问题造成用户损失少量成分币

- 危害性：中
- 可能性：中
- 影响力：中
- 类型：业务逻辑风险
- 漏洞描述

在 converter/DFEngine.sol:66 的 *deposit*() 实现中，当调用者传入的 *_srcToken* 是成分币之一，并且 *_srcAmount* > 0 时，这些成分币会立即被转入 *dfPool*。然而，在 converter/DFEngine.sol:67，*_srcAmount* 被 wrapper 机制转换成 *_amount*，并且此一 *_amount* 用来代表调用者存入 USDx 系统的成分币总数。在一些极端的情况，USDx 系统会少算了调用者存入的成分币，造成一个对用户不公平的结果。

```
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);
```

例如，当 *_srcToken* 的 decimal 是 19 并且 *_srcAmount* 是 1,009 时，1,009 个 *_srcToken* 会立即被转入 USDx 系统。按照 token/DSWrappedToken.sol 的实现，*_amount* 会依照以下的方法计算：

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

此时，会造成一个对用户不公平的情况，那就是 1,009 个 *_srcToken* 被转入 *dfPool* 但是只有 100 个 *wrapped* 的 *_srcToken* 也就是 1,000 个 *_srcToken* 被计算，用户因此会损失 9 个 *_srcToken*。

相似的逻辑在 *withdraw*() 也被实现了，建议一并修改。

- 攻击场景

如前所述，当 USDx 系统使用了某种成分币 T，其 decimal 是 19 时，用户 *deposit*() 1,009 个 T 时，只有 1,000 个 T 会被计算，短缺的 9 个 T 会被 *dfPool* 所吞噬。

- 修复方法

提前计算 *_amount* ，并且按照 *_amount* 对应的数量将 *_srcToken* 转入 *dfPool*，这样的话，存入 USDx 系统的成分币就会跟记在合约里的数量一致。此外，还可以利用 *_amount* > 0 的判断，提前过滤掉不需要后续处理的情况（例如，*_srcAmount* = 9，*decimal* = 19 时，计算出来的 *_amount* 为 0，可以直接退出）。

3.2 不符合 ERC20 规范的成分币兼容问题

- 危害性：低
- 可能性：低
- 影响力：低
- 类型：业务逻辑风险
- 漏洞描述

在 storage/DFCollateral.sol:16，*transferOut*() 调用了成分币的 *transfer*() 接口作转账操作，并且使用了 *assert*() 确保此一 *transfer*() 正常返回 true。

```
 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 }
```

然而，并不是所有代币合约都严格依照 ERC20 规范实现，存在 *transfer*() 不返回的情况，也可能返回值跟规范相反。

- 攻击场景

USDx 接入不符合 ERC20 规范的代币，*transfer*() 不返回，或返回相反逻辑。

- 修复方法

审计接入的代币合约，确保符合规范。或者去掉 *assert*() 检查，把 *transfer*() 失败的处理交给对应的代币合约。

3.3 不符合 ERC20 规范的成分币兼容问题

- 危害性：低
- 可能性：低
- 影响力：低
- 类型：业务逻辑风险
- 漏洞描述

在 storage/DFFunds.sol:16，*transferOut*() 调用了成分币的 *transfer*() 接口作转账操作，并且使用了 *assert*() 确保此一 *transfer*() 正常返回 true。

```
 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 }
```

然而，并非所有代币合约都严格依照 ERC20 规范实现，实际上存在 *transfer*() 不返回的情况，也可能返回值跟规范相反。

- 攻击场景

USDx 接入不符合 ERC20 规范的代币，*transfer()* 不返回，或返回相反逻辑。

- 修复方法

审计接入的代币合约，确保符合规范。或者去掉 *assert()* 检查，把 *transfer()* 失败的处理交给对应的代币合约。

3.4 不符合 ERC20 规范的成分币兼容问题

- 危害性：低
- 可能性：低
- 影响力：低
- 类型：业务逻辑风险
- 漏洞描述

在 storage/DFPool.sol:20，*transferFromSender()* 调用了成分币的 *transferFrom()* 接口作转账操作，并且使用了 *assert()* 确保此一 *transferFrom()* 正常返回 *true*。另外，在 storage/DFPool:30 及 storage/DFPool:40 也分别调用了成分币的 *transfer()* 接口并使用 *assert()* 检查返回值。

```
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      }
```

然而，并非所有代币合约都严格依照 ERC20 规范实现，存在 *transfer()* 或 *transferFrom()* 不返回的情况，也可能返回值跟规范相反。

- 攻击场景

USDx 接入不符合 ERC20 规范的代币，*transfer()* 或 *transferFrom()* 不返回，或返回相反逻辑。

- 修复方法

审计接入的代币合约，确保符合规范。或者去掉 *assert()* 检查，把 *transfer()* 及 *transferFrom()* 失败的处理交给对应的代币合约。

3.5 不符合 ERC20 规范的成分币兼容问题

- 危害性：低
- 可能性：低
- 影响力：低
- 类型：业务逻辑风险
- 漏洞描述

在 storage/DFPool.sol:58，*transferFromSenderToCol()* 调用了成分币的 *transferFrom()* 接口作转账操作，并且使用了 *assert()* 确保此一 *transferFrom()* 正常返回 *true*。

```
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      }
```

- 攻击场景

USDx 接入不符合 ERC20 规范的代币，*transferFrom()* 不返回，或返回相反逻辑。

- 修复方法

审计接入的代币合约，确保符合规范。或者去掉 *assert()* 检查，把 *transferFrom()* 失败的处理交给对应的代币合约。

3.6 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:8 的 *getDepositMaxMint()* 接口与 converter/DFEngine.sol:317 的函数声明不一致

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)
```

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.7 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:9 的 *getMaxToClaim()* 接口与 converter/DFEngine.sol:352 的函数声明不一致

converter/interfaces/IDFEngine.sol:9

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

converter/DFEngine.sol:352

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

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.8 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:10 的 *getCollateralMaxClaim()* 接口与 converter/DFEngine.sol:370 的函数声明不一致

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)
```

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.9 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:11 的 *getMintingSection()* 接口与 converter/DFEngine.sol:383 的函数声明不一致

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)
```

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.10 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:12 的 *getBurningSection()* 接口与 converter/DFEngine.sol:391 的函数声明不一致

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)
```

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.11 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:13 的 *getWithdrawBalances()* 接口与 converter/DFEngine.sol:399 的函数声明不一致

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)
```

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.12 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:14 的 *getPrices()* 接口与 converter/DFEngine.sol:411 的函数声明不一致

converter/interfaces/IDFEngine.sol:14

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

converter/DFEngine.sol:411

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

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.13 接口与声明不一致问题

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

converter/interfaces/IDFEngine.sol:15 的 *getFeeRateByID()* 接口与 converter/DFEngine.sol:419 的函数声明不一致

converter/interfaces/IDFEngine.sol:15

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

converter/DFEngine.sol:419

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

- 攻击场景

  无

- 修复方法

  改为一致（接口加上 view）。

3.14 代码优化建议

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

  以下代码建议在声明时加上 view

  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)

- 攻击场景

  无

- 修复方法

  加上 view。

3.15 代码优化建议

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

在两处 *approveToEngine()* 的实现里（storage/DFPool.sol:54 以及 storage/DFCollateral.sol:20 ）， *approve()* 的成功或失败并没有被检查，我们建议可以利用 ERC-20 标准的 *allowance()* 接口检查 *approve()* 后的状态。

```
54     function approveToEngine(address _tokenIdx, address _engineAddress) public auth {
55         IERC20Token(_tokenIdx).approve(_engineAddress, uint(-1));
56     }
```

```
20     function approveToEngine(address _tokenIdx, address _engineAddress) public auth {
21         IERC20Token(_tokenIdx).approve(_engineAddress, uint(-1));
22     }
```

- 攻击场景

  无

- 修复方法

  加入 *allowance()* 的检查。

3.16 代码优化建议

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：代码建议及优化
- 漏洞描述

在 converter/DFEngine.sol:255 的 *oneClickMinting()* 实现中，*_amount* 这个输入参数没有在任何地方被检查。

```
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         }
```

如果调用者没有足够的成分币余额，会在第 243 行将成分币转给 dfPool 时发生错误，退出 *oneClickMinting()*，这可能会造成 gas 的浪费。

一个优化的思路是在真的开始运行第 240-246 的循环之前，先利用 ERC-20 标准的 *balanceOf()* 检查调用者的成分币余额，如果与 *_amount* 不匹配，可以直接退出 *oneClickMinting()*。

- 攻击场景

无

- 修复方法

加入 *balanceOf()* 检查，在调用者没有足够成分币时，直接退出。

3.17 接口声明缺失

- 危害性：低危
- 可能性：低
- 影响力：低
- 类型：代码建议及优化
- 漏洞描述

定义于 converter/DFProtocol.sol:36 的 *oneClickMinting()*，其对应的接口声明在 converter/interfaces/IDFProtocol.sol 缺失。

```
3 contract IDFProtocol {
4     function deposit(address _tokenID, uint _feeTokenIdx, uint _amount) public returns (uint _balance);
5     function withdraw(address _tokenID, uint _feeTokenIdx, uint _amount) public returns (uint _balance);
6     function destroy(uint _feeTokenIdx, uint _amount) public;
7     function claim(uint _feeTokenIdx) public returns (uint _balance);
8 }
```

- 攻击场景

  无

- 修复方法

  在 *IDFProtocol* 里加入 *oneClickMinting()* 接口声明。

3.18 某处权限未设置

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：权限管理
- 漏洞描述

在 migrations/3_after_deploy.js 的合约部署脚本中，相较于其他合约，Setting 合约的权限没有被用 *setAuthority()* 设置。

- 攻击场景

无

- 修复方法

使用 *setAuthority()* 对 Setting 合约设置权限。

3.19 未实现熔断机制

- 危害性：信息
- 可能性：无
- 影响力：无
- 类型：熔断机制
- 漏洞描述

  未实现熔断机制。

- 攻击场景

  无

- 修复方法

  加入熔断机制，例如在关键函数入口加入 *require(live == 1)*，并且实现一个函数接口，用来设置 *live* 变量。

4. 结论

我们对USDx的智能合约代码进行了安全审计，该特定版本的智能合约代码存在逻辑错误导致用户可能损失部分代币，以及不符合ERC20规范的成分币兼容问题等，目前皆已修复。

但正如在**免责声明** 1.4 中所述，该审计报告的结果并不意味着该智能合约不存在其它安全问题，亦不构成任何投资建议。

最后，对于报告内容和格式，我们欢迎任何建设性的反馈或建议。

5. 参考文献

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

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