SMART CONTRACT AUDIT REPORT

for

DFORCE NETWORK

Prepared By: Shuxiao Wang

Hangzhou, China
Mar. 24, 2020

Document Properties

Client	dForce Network
Title	Smart Contract Audit Report
Target	USR
Version	1.0-rc1
Author	Chiachih Wu
Auditors	Chiachih Wu, Huaguo Shi
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	Mar. 24, 2020	Chiachih Wu	Status Update
0.2	Mar. 23, 2020	Huaguo Shi	Status Update
0.1	Mar. 20, 2020	Huaguo Shi	Initial 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 **USR** design document and related smart contract source code, 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 USR

USDx Savings Rate (USR) is an addition of dForce Protocol that allows any USDx holder to earn risk-free savings. The savings paid out to USDx holders are financed by DIP-001 protocol which deposits constituent stable coins into the decentralized lending market to earn interest.

The basic information of USR is as follows:

Table 1.1: Basic Information of USR

Item	Description
Issuer	dForce Network
Website	https://dforce.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Mar. 24, 2020

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

- https://github.com/dforce-network/USR/tree/audit (05cfe29)
- https://github.com/dforce-network/USR/tree/audit (b49ae71)

- https://github.com/dforce-network/USR/tree/audit (cf24e4d)

1.2 About PeckShield

PeckShield Inc. [24] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [19]:

- 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

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

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) [18], 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 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 should not be used as an 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 USR implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	0	
Medium	2	■■
Low	2	■■
Informational	7	■■■■■■■
Total	11	

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 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 2 medium-severity vulnerability, 2 low-severity vulnerabilities, and 7 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Missing Address Validation in changeAdmin()	Business Logics	Resolved
PVE-002	Info.	Wrong Function Types	Security Features	Confirmed
PVE-003	Info.	Missing takeout() Function in USR	Behavioral Issues	Resolved
PVE-004	Info.	Excessive Return Statement in transferOut()	Error Conditions	Resolved
PVE-005	Info.	Gas Optimization in drip()	Resource Management	Resolved
PVE-006	Low	approve()/transferFrom() Race Condition	Time and State	Confirmed
PVE-007	Info.	Wrong Variable Name	Bad Coding Practices	Resolved
PVE-008	Low	Flawed Fee Calculation	Business Logics	Confirmed
PVE-009	Medium	Missing Drip in setInterestRate()	Business Logics	Resolved
PVE-010	Info.	Missing Assertion Messages	Bad Coding Practices	Resolved
PVE-011	Info.	Missing Owner Check in transferOwnership()	Resource Management	Resolved

Please refer to Section 3 for details.

3 Detailed Results

3.1 Missing Address Validation in changeAdmin()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `contracts/USRProxy.sol`
- Category: Business Logics[15]
- CWE subcategory: CWE-754 [9]

Description

The `changeAdmin()` function in USRProxy allows the current admin to transfer her privileges to another address. However, inside `changeAdmin()`, the `newAdmin` is directly stored into the storage through `_setAdmin()` after validating the `newAdmin` is a non-zero address, which is not enough.

```
105     function changeAdmin(address newAdmin) external ifAdmin {
106         require(newAdmin != address(0), "Cannot change the admin of a proxy to the zero
                address");
107         emit AdminChanged(_admin(), newAdmin);
108         _setAdmin(newAdmin);
109     }
```

Listing 3.1: contracts/USRProxy.sol

As shown in the above code snippets, `newAdmin` is only validated against the zero address in line 106. However, if you enter a wrong address by mistake, you will never be able to take the management permissions back. Besides, if the `newAdmin` is the same as the current admin address stored in `ADMIN_SLOT`, it's a waste of gas.

Recommendation The transition should be managed by the implementation with a two-step approach: `changeAdmin()` and `updateAdmin()`. Specifically, the `changeAdmin()` function keeps the new address in the storage, `newAdmin`, instead of modifying the `ADMIN_SLOT` through `_setAdmin()`. The `updateAdmin()` function checks whether `newAdmin` is `msg.sender`, which means `newAdmin` signs the transaction and verifies itself as the new admin. After that, `newAdmin` could be `_setAdmin()` into `ADMIN_SLOT`. This had been addressed in the patched `contracts/USRProxy.sol`.

```
    function changeAdmin(address _newAdmin) external ifAdmin {
        require(_newAdmin != address(0), "Cannot change the admin of a proxy to the zero address");
        require(_newAdmin != _admin(), "The current and new admin cannot be the same.");
        require(_newAdmin != newAdmin, "Cannot set the newAdmin of a proxy to the same address.");
        newAdmin = _newAdmin;
        emit AdminChanged(_admin(), newAdmin);
    }

    function updateAdmin() external {
        require(newAdmin != address(0), "Cannot change the newAdmin of a proxy to the zero address");
        require(msg.sender == newAdmin, "msg.sender and newAdmin must be the same.");
        _setAdmin(newAdmin);
        emit AdminUpdated(_admin());
    }
```

Listing 3.2: Revised contracts/USRProxy.sol

3.2 Wrong Function Types

- ID: PVE-002
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USR.sol`
- Category: Security Features[11]
- CWE subcategory: CWE-269 [5]

Description

The `initialize()` function in USR is not necessary a public function due to the fact that it is only called once in the `constructor()`.

```
    /**
     * The constructor is used here to ensure that the implementation
     * contract is initialized. An uncontrolled implementation
     * contract might lead to misleading state
     * for users who accidentally interact with it.
     */
    constructor(string memory _name, string memory _symbol, uint8 _decimals, address _interestModel, address _usdx, uint _originationFee, uint _maxDebtAmount) public {
        initialize(_name, _symbol, _decimals, _interestModel, _usdx, _originationFee, _maxDebtAmount);
    }

    // --- Init ---
```

```
    function initialize(string memory _name, string memory _symbol, uint8 _decimals,
        address _interestModel, address _usdx, uint _originationFee, uint _maxDebtAmount
        ) public {
        require(!initialized, "initialize: already initialized.");
        require(_originationFee < BASE / 10, "initialize: fee should be less than ten
            percent.");
        name = _name;
```

Listing 3.3: contracts/USR.sol

Recommendation Change the function type of `initialize()` from `public` to `internal`. Since it's an internal function now, we also suggest to rename it to `_initialize()`.

```
    /**
     * The constructor is used here to ensure that the implementation
     * contract is initialized. An uncontrolled implementation
     * contract might lead to misleading state
     * for users who accidentally interact with it.
     */
    constructor(string memory _name, string memory _symbol, uint8 _decimals, address
        _interestModel, address _usdx, uint _originationFee, uint _maxDebtAmount) public
        {
        _initialize(_name, _symbol, _decimals, _interestModel, _usdx, _originationFee,
            _maxDebtAmount);
    }

    // --- Init ---
    function _initialize(string memory _name, string memory _symbol, uint8 _decimals,
        address _interestModel, address _usdx, uint _originationFee, uint _maxDebtAmount
        ) internal {
        require(!initialized, "initialize: already initialized.");
        require(_originationFee < BASE / 10, "initialize: fee should be less than ten
            percent.");
        name = _name;
```

Listing 3.4: Revised contracts/USR.sol

3.3 Missing takeOut() Function

- ID: PVE-003
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USR.sol`
- Category:Behavioral Issues [14]
- CWE subcategory: CWE-431 [8]

Description

In the USR design document, the USR contract provides the `takeOut()` function for the manager to take out some tokens and transfer them to the receiver. However, `takeOut()` is not implemented in the code.

Recommendation Implement `takeOut()` or fix the design document. This had been addressed in the patched `contracts/USR.sol` by renaming `transferOut()` to `takeOut()`.

3.4 Excessive Return Statement in transferOut()

- ID: PVE-004
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USR.sol`
- Category: Error Conditions, Return Values, Status Codes [16]
- CWE subcategory: CWE-394 [7]

Description

In USR contract, `transferout()` is designed to provide the token transfer function for other modules (e.g., DIP001). However, the function always returns `true`. As shown in line 145, `true` is returned whether transfer succeed or not, which makes the return code meaningless.

```
135     /**
136      * @dev Manager function to transfer token out to earn extra savings
137            but only when the contract is not paused.
138      * @param _token reserve asset, generally spaking it should be USDx.
139      * @param _recipient account to receive asset.
140      * @param _amount transfer amount.
141      * @return bool true=success, otherwise a failure.
142      */
143     function transferOut(address _token, address _recipient, uint _amount) external
            onlyManager whenNotPaused returns (bool) {
144         require(doTransferOut(_token, _recipient, _amount));
145         return true;
146     }
```

Listing 3.5: contracts/USR.sol

Recommendation Modify the definition of `transferOut()` by removing the return statement. This had been addressed in the patched `contracts/USR.sol`.

```
    /**
     * @dev Manager function to transfer token out to earn extra savings
           but only when the contract is not paused.
     * @param _token reserve asset, generally spaking it should be USDx.
     * @param _recipient account to receive asset.
     * @param _amount transfer amount.
     */
    function transferOut(address _token, address _recipient, uint _amount) external
        onlyManager whenNotPaused {
        require(doTransferOut(_token, _recipient, _amount));
```

```
    }
```

Listing 3.6: contracts/USR.sol

3.5 Gas Optimization in drip()

- ID: PVE-005
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USR.sol`
- Category:Resource Management [17]
- CWE subcategory: CWE-920 [10]

Description

In USR contract, `drip()` updates `exchangeRate` and `lastTriggerTime` even in the condition of `now == lastTriggerTime`, which is a waste of gas. Specifically, the `drip()` function calls `rpow()` to recalculate the `exchangeRate` based on the interest rate and time. When `now = lastTriggerTime`, the result of the `rpow()` call would be $\_usr^0$ which is 1. Therefore, the recalculated `exchangeRate` would not change after some no-effect code which consumes gas.

```
    /**
     * @dev Savings Rate Accumulation.
     * @return the most recent exchange rate, scaled by 1e27.
     */
    function drip() public note returns (uint _tmp) {
        require(now >= lastTriggerTime, "drip: invalid now.");
        uint _usr = InterestModel(interestModel).getInterestRate();
        _tmp = rmul(rpow(_usr, now - lastTriggerTime, ONE), exchangeRate);
        exchangeRate = _tmp;
        lastTriggerTime = now;
    }
```

Listing 3.7: contracts/USR.sol

```
    // --- Math ---
    function rpow(uint x, uint n, uint base) internal pure returns (uint z) {
        assembly {
            switch x case 0 {switch n case 0 {z := base} default {z := 0}}
            default {
                switch mod(n, 2) case 0 { z := base } default { z := x }
                let half := div(base, 2)  // for rounding.
                for { n := div(n, 2) } n { n := div(n,2) } {
                    let xx := mul(x, x)
                    if iszero(eq(div(xx, x), x)) { revert(0,0) }
                    let xxRound := add(xx, half)
                    if lt(xxRound, xx) { revert(0,0) }
                    x := div(xxRound, base)
```

```
                    if mod(n,2) {
                        let zx := mul(z, x)
                        if and(iszero(iszero(x)), iszero(eq(div(zx, x), z))) { revert(0,0) }
                        let zxRound := add(zx, half)
                        if lt(zxRound, zx) { revert(0,0) }
                        z := div(zxRound, base)
                    }
                }
            }
        }
    }
```

Listing 3.8: contracts/USR.sol

Recommendation Change `now >= lastTriggerTime` to `now > lastTriggerTime`. In addition, we noticed all callers check `now > lastTriggerTime` before calling `drip()` due to the fact that `drip()` may revert when now < lastTriggerTime which breaks the business logic. Therefore, we suggest to replace the `require()` statement into an `if-else` check. This way, the callers no longer need to know the logic inside `drip()` and the function would not revert when it is called at a wrong time. According to the patched `USR.sol`, this had been addressed.

```
    /**
     * @dev Savings Rate Accumulation.
     * @return the most recent exchange rate, scaled by 1e27.
     */
     function drip() public note returns (uint) {
        if(now > lastTriggerTime){
            _tmp = rmul(rpow(interestRate, now - lastTriggerTime, ONE), exchangeRate);
            exchangeRate = _tmp;
            lastTriggerTime = now;
            return _tmp;
        }
        return exchangeRate;
    }
```

Listing 3.9: contracts/USR.sol

3.6 approve()/transferFrom() Race Condition

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `contracts/USR.sol`
- Category: Time and State [12]
- CWE subcategory: CWE-362 [6]

Description

There is a known race condition issue regarding approve()/transferFrom() [2]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 USR to 1 USR. The previously approved spender might race to transfer the amount you initially approved (the 10 USR) and then additionally spend the new amount you just approved (1 USR). This breaks the user's intention of restricting the spender to the new amount (1 USR), not the sum of old amount and new amount (11 USR).

```
    function approve(address _spender, uint _wad) external returns (bool) {
        allowance[msg.sender][_spender] = _wad;
        emit Approval(msg.sender, _spender, _wad);
        return true;
    }
```

Listing 3.10: contracts/USR.sol

Recommendation Add additional sanity checks in approve().

```
    function approve(address _spender, uint _wad) external returns (bool) {
        require((_wad == 0)  (allowed[msg.sender][_spender] == 0));
        allowance[msg.sender][_spender] = _wad;
        emit Approval(msg.sender, _spender, _wad);
        return true;
    }
```

Listing 3.11: contracts/USR.sol

3.7 Wrong Variable Name

- ID: PVE-007
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USR.sol`
- Category: Bad Coding Practices [13]
- CWE subcategory: CWE-1099 [3]

Description

In USR contract, `equity()` could be used to retrieve the current contract debts. However, the variable `_banance` for keeping the balance of USDx is spelled wrong.

```
    /**
     * @dev Get current contract debet.
     * @return int > 0 indicates no debts,
     *         otherwise in debt, and it indicates lossing amount, scaled by 1e18.
     */
    function equity() external view returns (int) {
```

```
        uint _totalAmount = rmul(totalSupply, getExchangeRate());
        uint _banance = IERC20(usdx).balanceOf(address(this));
        if (_totalAmount > _banance)
            return -1 * int(_totalAmount.sub(_banance));

        return int(_banance.sub(_totalAmount));
    }
```

Listing 3.12: contracts/USR.sol

Recommendation Rename the variable `_banance` to `_balance`. This had been addressed in the patched `contracts/USR.sol`.

```
    /**
     * @dev Get current contract debet.
     * @return int > 0 indicates no debts,
     *         otherwise in debt, and it indicates lossing amount, scaled by 1e18.
     */
    function equity() external view returns (int) {
        uint _totalAmount = rmul(totalSupply, getExchangeRate());
        uint _balance = IERC20(usdx).balanceOf(address(this));
        if (_totalAmount > _balance)
            return -1 * int(_totalAmount.sub(_balance));

        return int(_balance.sub(_totalAmount));
    }
```

Listing 3.13: contracts/USR.sol

3.8 Flawed Fee Calculation

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `contracts/USR.sol`
- Category: Business Logics[15]
- CWE subcategory:

Description

In `draw()` and exit(), we noticed that `originationFee` is used to calculate the amount that the user needs to pay for management fee. However, USR does not consider the time keeping the user's digital assets in fee calculation. Instead, only a portion of USDx or USR tokens are kept as an one-time payment when the user is withdrawing USDx out.

Recommendation It's common to implement a time-based management fee mechanism. People make profits based on time. They should pay more fee when they make more profits (i.e., putting money in USR for a longer time).

3.9 Missing Drip in setInterestRate()

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `contracts/InterestModel.sol`
- Category:
- CWE subcategory:

Description

In USR, the `exchangeRate` is the global variable which represents the amount of USDx required to exchange for one USR token. For example, when an user pays in 100 USDx, $\frac{100}{exchangeRate}$ of USR tokens would be minted and transferred to the user. Since the system pays interets to users based on time, the `exchangeRate` keeps increasing by $exchangeRate \times (interestRate)^t$ where t represents number of seconds since the USR smart contract is initialized (`drip()` function). However, the `interestRate` could be updated by `setInterestRate()`, which results in wrong `exchangeRate` calculation.

For example, someone calls `drip()` at the $t = 100$ and the $t = 106$ respectively. If the $interestRate$ is not updated during $100 < t < 106$, $exchangeRate_{106}$ should be $exchangeRate_{100} \times (interestRate)^6$. If $interestRate$ is updated to $interestRate_{NEW}$ at $t = 105$, the current implementation generates $exchangeRate_{106} = exchangeRate_{100} \times (interestRate_{NEW})^6$. But, based on the business logic, $exchangeRate_{106}$ should be $exchangeRate_{100} \times (interestRate)^5 \times (interestRate_{NEW})$.

Recommendation Call `drip()` before updating `interestRate`. In the patches, this issue had be resolved and its remove the interface `InterestModel`.

3.10 Missing Assertion Messages

- ID: PVE-010
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/USRProxy.sol`
- Category: Bad Coding Practices [13]
- CWE subcategory: CWE-1113 [4]

Description

In `upgradeToAndCall()`, the `require` statement in line 118 does not provide enough information (i.e., no assertion message).

```
115    function upgradeToAndCall(address newImplementation, bytes calldata data) payable
           external ifAdmin {
116        _upgradeTo(newImplementation);
```

```
        (bool success,) = address(this).call.value(msg.value)(data);
        require(success);
        // require(address(this).call.value(msg.value)(data));
    }
```

Listing 3.14: contracts/USRProxy.sol

Recommendation Add assertion messages. This had been addressed in the patched `contracts/USRProxy.sol`.

```
    function upgradeToAndCall(address newImplementation, bytes calldata data) payable
        external ifAdmin {
        _upgradeTo(newImplementation);
        (bool success,) = address(this).call.value(msg.value)(data);
        require(success, "upgradeToAndCall-error");
    }
```

Listing 3.15: contracts/USRProxy.sol

3.11 Missing Owner Check in transferOwnership()

- ID: PVE-011
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/library/Ownable.sol`
- Category:Resource Management [17]
- CWE subcategory: CWE-920 [10]

Description

In Ownable contract, the `transferOwnership()` function does not validate the `_newOwner` against the `pendingOwner`, which is a waste of gas.

```
    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner_`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address _newOwner) external onlyOwner {
        require(_newOwner != owner, "transferOwnership: the same owner.");
        pendingOwner = _newOwner;
    }
```

Listing 3.16: contracts/library/Ownable.sol

Recommendation Ensure that the parameter, `_newOwner`, is not equal to `pendingOwner`. This had been addressed in the patched `contracts/Ownable.sol`.

```
     /**
      * @dev Transfers ownership of the contract to a new account (`newOwner_`).
```

```
     * Can only be called by the current owner.
    */
    function transferOwnership(address _newOwner) external onlyOwner {
        require(_newOwner != owner, "transferOwnership: the same owner.");
        require(_newOwner != pendingOwner, "transferOwnership: the same pendingOwner.")
            ;
        pendingOwner = _newOwner;
    }
```

Listing 3.17: contracts/library/Ownable.sol

3.12 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.12;` instead of `pragma solidity ^0.5.12;`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries. In case there is an absolute need of leveraging experimental features or integrating external libraries, make necessary contingency plans.

4 Conclusion

In this audit, we thoroughly analyzed the USR documentation and implementation. The audited system does involve various intricacies in both design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [20, 21, 22, 23, 25].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [26] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

[3] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[4] MITRE. CWE-1113: Inappropriate Comment Style. https://cwe.mitre.org/data/definitions/1113.html.

[5] MITRE. CWE-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

[6] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[7] MITRE. CWE-394: Unexpected Status Code or Return Value. https://cwe.mitre.org/data/definitions/394.html.

[8] MITRE. CWE-431: Missing Handler. https://cwe.mitre.org/data/definitions/431.html.

[9] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[10] MITRE. CWE-920: Improper Restriction of Power Consumption. https://cwe.mitre.org/data/definitions/920.html.

[11] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[12] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[14] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[15] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

[17] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

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

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

[20] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[21] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[22] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[23] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[25] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[26] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.