SMART CONTRACT AUDIT REPORT

for

RULER PROTOCOL

PeckShield

Prepared By: Shuxiao Wang

PeckShield

February 19, 2021

Document Properties

Client	Ruler Protocol
Title	Smart Contract Audit Report
Target	Ruler Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 19, 2021	Xuxian Jiang	Final Release
1.0-rc	February 18, 2021	Xuxian Jiang	Release Candidate #1
0.3	February 10, 2021	Xuxian Jiang	Additional Findings #2
0.2	February 7, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 5, 2021	Xuxian Jiang	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 design document and related smart contract source code of the `Ruler` protocol, we outline in the report 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 Ruler Protocol

The `Ruler` protocol is a lending platform where users can borrow their preferred cryptocurrency with any other cryptocurrency. It aims to fill the gap by enabling the following goals: 1) supply and demand decide interest rate; 2) no liquidation as long as borrowers payback on time; 3) fungible loans that can be traded anytime. At the core of Ruler Protocol is the notion of `ruler pairs` and each pair consists of four elements: `collateral token`, `paired token`, `expiry`, and `mint ratio`. In essence, the protocol enables a market driven lending platform that provides non-liquidatable and fungible loans.

The basic information of the `Ruler` protocol is as follows:

Table 1.1: Basic Information of The `Ruler` Protocol

Item	Description
Issuer	Ruler Protocol
Website	https://rulerprotocol.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 19, 2021

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

- https://github.com/Ruler-Protocol/ruler-core.git (b90e7a6)

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

- https://github.com/Ruler-Protocol/ruler-core.git (bd0ca68)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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.

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

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 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) [8], 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 Ruler Protocol 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	0	
High	0	
Medium	1	
Low	3	
Informational	1	
Total	5	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Ruler Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Business Logic	Fixed
PVE-003	Informational	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-004	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Confirmed
PVE-005	Low	Front-Running For Nonce Invalidation	Time and State	Confirmed

Besides recommending specific countermeasures to mitigate these issues, based on 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 0.8.0` instead of specifying a range, e.g., `pragma solidity ^0.8.0`.

In addition, 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 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

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. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts. In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
```

```
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address recipient, uint256 amount)external returns (bool)`. 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.

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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `collectDust()` routine in the `BonusRewards` contract. If the `USDT` token is given as the routine's argument, i.e., `_token`, the unsafe version of `IERC20(_token).transfer(owner(), balance)` (line 236) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
219   /// @notice collect bonus token dust to treasury
220   function collectDust(address _token, address _lpToken, uint256 _poolBonusId) external
          override onlyOwner {
221     require(pools[_token].lastUpdatedAt == 0, "BonusRewards: lpToken, not allowed");

223     if (_token == address(0)) { // token address(0) = ETH
224       payable(owner()).transfer(address(this).balance);
225     } else {
226       uint256 balance = IERC20(_token).balanceOf(address(this));
227       if (bonusTokenAddrMap[_token] == 1) {
228         // bonus token
229         Bonus memory bonus = pools[_lpToken].bonuses[_poolBonusId];
230         require(bonus.bonusTokenAddr == _token, "BonusRewards: wrong pool");
231         require(bonus.endTime + WEEK < block.timestamp, "BonusRewards: not ready");
232         balance = bonus.remBonus;
233         pools[_lpToken].bonuses[_poolBonusId].remBonus = 0;
234       }

236       IERC20(_token).transfer(owner(), balance);
237     }
```

Listing 3.2: BonusRewards::collectDust()

Note that the same issue exists in the `_depositAndAddLiquidity()` routine from the `RulerZap` contract, which reverts related liquidation additions. Also, the `_approve()` helper from the same contract shares the same issue, which may revert a number of calling routines.

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

Status The issue has been fixed by this commit: `4e753ce`.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: medium
- Impact: Low
- Target: `RulerZap`
- Category: Business Logic [5]
- CWE subcategory: N/A

Description

In Section 3.1, we have examined certain non-compliant ERC20 tokens that may exhibit specific idiosyncrasies in their `transfer()` and `transferFrom()` implementations. In this section, we examine the `approve()` routine and possible another idiosyncrasy from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use as an example the `RulerZap` contract that is designed to facilitate the interaction with the `Ruler Core` contract. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function _deposit(
        address _col,
        address _paired,
        uint48 _expiry,
        uint256 _mintRatio,
        uint256 _colAmt
    ) internal returns (address rcTokenAddr, uint256 rcTokenReceived, uint256 rcTokenBalBefore) {
        ( , , , IRERC20 rcToken, IRERC20 rrToken, , , ) = core.pairs(_col, _paired, _expiry, _mintRatio);
        // receive collateral from sender
        IERC20 collateral = IERC20(_col);
        uint256 colBalBefore = collateral.balanceOf(address(this));
        collateral.safeTransferFrom(msg.sender, address(this), _colAmt);
        uint256 received = collateral.balanceOf(address(this)) - colBalBefore;
        require(received > 0, "RulerZap: col transfer failed");

        // deposit collateral to Ruler
        rcTokenBalBefore = rcToken.balanceOf(address(this));
        uint256 rrTokenBalBefore = rrToken.balanceOf(address(this));
        _approve(collateral, address(core), received);
        core.deposit(_col, _paired, _expiry, _mintRatio, received);

        // send rrToken back to sender, and record received rcTokens
        rrToken.transfer(msg.sender, rrToken.balanceOf(address(this)) - rrTokenBalBefore);
        rcTokenReceived = rcToken.balanceOf(address(this)) - rcTokenBalBefore;
        rcTokenAddr = address(rcToken);
    }

    function _approve(IERC20 _token, address _spender, uint256 _amount) internal {
        if (_token.allowance(address(this), _spender) < _amount) {
            _token.approve(_spender, type(uint256).max);
        }
    }
```

Listing 3.4: RulerZap::_deposit()

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been fixed by this commit: `4e753ce`.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BonusRewards`
- Category: Numeric Errors [7]
- CWE subcategory: CWE-190 [2]

Description

In the `Ruler` protocol, there is a `BonusRewards` contract that allows for rewarding multiple bonus tokens for participating users. The reward logic enforces pro-rata claims of disseminated rewards.

For illustration, we show below the `updateBonus()` routine that is used to update an active rewarding pool. It implements a rather straightforward logic by firstly performing necessary sanity checks on the input arguments, and then updating the related bonus entry with the specified `startTime` and `weeklyRewards` (lines 154 – 164).

```
136  /// @notice called by authorizers only, update weeklyRewards (if not ended), or update
         startTime (only if rewards not started, 0 is ignored)
137  function updateBonus(
138    address _lpToken,
139    address _bonusTokenAddr,
140    uint256 _weeklyRewards,
141    uint48 _startTime
142  ) external override nonReentrant notPaused {
143    require(_isAuthorized(allowedTokenAuthorizers[_lpToken][_bonusTokenAddr]), "
         BonusRewards: not authorized caller");
144    require(_startTime == 0  _startTime > block.timestamp, "BonusRewards: startTime in
         the past");

146    // make sure the pool is in the right state (exist with no active bonus at the
         moment) to add new bonus tokens
147    Pool memory pool = pools[_lpToken];
148    require(pool.lastUpdatedAt > 0, "BonusRewards: pool does not exist");
149    Bonus[] memory bonuses = pool.bonuses;
150    for (uint256 i = 0; i < bonuses.length; i++) {
151      if (bonuses[i].bonusTokenAddr == _bonusTokenAddr && bonuses[i].endTime > block.
           timestamp) {
152        Bonus storage bonus = pools[_lpToken].bonuses[i];
153        _updatePool(_lpToken); // update pool with old weeklyReward to this block
154        if (bonus.startTime >= block.timestamp) {
155          // only honor new start time, if program has not started
156          if (_startTime >= block.timestamp) {
157            bonus.startTime = _startTime;
158          }
```

```
159             bonus.endTime = uint48(bonus.remBonus * WEEK / _weeklyRewards + bonus.
                    startTime);
160           } else {
161             uint256 remBonusToDistribute = (bonus.endTime - block.timestamp) * bonus.
                    weeklyRewards / WEEK;
162             bonus.endTime = uint48(remBonusToDistribute * WEEK / _weeklyRewards + block.
                    timestamp);
163           }
164           bonus.weeklyRewards = _weeklyRewards;
165         }
166       }
167     }
```

Listing 3.5: BonusRewards::updateBonus()

The update of `startTime` and `weeklyRewards` naturally leads to the re-calculation of the `endTime`. It comes to our attention that the current `endTime` is computed as follows: `uint48(remBonusToDistribute * WEEK / _weeklyRewards + block.timestamp)`, where `remBonusToDistribute = (bonus.endTime - block.timestamp)* bonus.weeklyRewards / WEEK`.

It is important to emphasize that the lack of `float` support in `Solidity` may introduce subtle, but troublesome issue: precision loss. One possible precision loss stems from the computation when both multiplication (`mul`) and division (`div`) are involved. Specifically, the computation at lines 161 – 162 can be performed as follows: `uint48((bonus.endTime - block.timestamp)* bonus.weeklyRewards / _weeklyRewards + block.timestamp)`.

A better approach is the one that can always avoid or reduce any precision loss. In other words, the computation of the form `A / B * C` can be converted into `A * C / B` under the condition that `A * C` does not introduce any overflow.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in `Solidity`. An example revision to the above `endTime` is shown as: `remBonusToDistribute = (bonus.endTime - block.timestamp)* bonus.weeklyRewards / WEEK`.

Status The issue has been fixed by this commit: `40f65e4`.

3.4 Improved Sanity Checks For System/Function Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RulerCore`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Ruler Protocol protocol is no exception. Specifically, if we examine the `RulerCore` contract, it has defined a number of protocol-wide risk parameters: `flashLoanRate` and `minColRatioMap`. In the following, we show the corresponding routine that allows for their changes.

```
function setFlashLoanRate(uint256 _newRate) external override onlyOwner {
  emit FlashLoanRateUpdated(flashLoanRate, _newRate);
  flashLoanRate = _newRate;
}

function setMinColRatio(address _col, uint256 _minColRatio) external override onlyOwner {
  require(minColRatioMap[_col] > 0, "Ruler: collateral not listed");
  require(_minColRatio >= 0.5 ether, "Ruler: min colRatio < 50%");
  emit MinColRatioUpdated(_col, minColRatioMap[_col], _minColRatio);
  minColRatioMap[_col] = _minColRatio;
}
```

Listing 3.6: RulerCore::setFlashLoanRate() and RulerCore::setMinColRatio()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `flashLoanRate` may charge unreasonably high fee in the `flashLoan()` operation, hence incurring cost to participating users or hurting the adoption of the flashloans.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been confirmed and by design it is allowed to set any flashloan rate of choice.

3.5 Possible Front-Running For Nonce Invalidation

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [3]

Description

In the `Ruler` protocol, both `RulerCore` and `RulerZap` contracts support `permit`-style function variants that allow users to authorize actions using an off-chain signature. The intention is to support meta-transactions such that an user can simply sign an intended transaction offline and then send the signed transaction to a relayer. The replayer will take care of submitting the transaction for mining by paying required transaction fee. The signature correctness will be verified on chain via a helper routine, i.e., `permit()`.

In the following, we elaborate the related execution logic. Particularly, the `depositWithPermit()` function verifies the signature using the internal `_permit()` helper (line 126) before executing the intended `deposit()`.

```
/// @notice deposit collateral to a Ruler Pair, sender receives rTokens
function deposit(
  address _col,
  address _paired,
  uint48 _expiry,
  uint256 _mintRatio,
  uint256 _colAmt
) public override onlyNotPaused nonReentrant {
  Pair memory pair = pairs[_col][_paired][_expiry][_mintRatio];
  _validateDepositInputs(_col, pair);

  // receive collateral
  IERC20 collateral = IERC20(_col);
  uint256 colBalBefore = collateral.balanceOf(address(this));
  collateral.safeTransferFrom(msg.sender, address(this), _colAmt);
  uint256 received = collateral.balanceOf(address(this)) - colBalBefore;
  require(received > 0, "Ruler: transfer failed");
  pairs[_col][_paired][_expiry][_mintRatio].colTotal = pair.colTotal + received;

  // mint rTokens for reveiced collateral
  uint256 mintAmount = _getRTokenAmtFromColAmt(received, _col, _paired, pair.mintRatio
      );
  pair.rcToken.mint(msg.sender, mintAmount);
  pair.rrToken.mint(msg.sender, mintAmount);
  emit Deposit(msg.sender, _col, _paired, _mintRatio, received);
}
```

```
function depositWithPermit(
  address _col,
  address _paired,
  uint48 _expiry,
  uint256 _mintRatio,
  uint256 _colAmt,
  Permit calldata _colPermit
) external override {
  _permit(_col, _colPermit);
  deposit(_col, _paired, _expiry, _mintRatio, _colAmt);
}
```

Listing 3.7: RulerCore::depositWithPermit()

Within the `_permit()` helper, we observe that it basically invokes the built-in public `permit()` function in the token contract. This public function takes the normal procedure, i.e., `ecrecover()`, to retrieve the signer information. If validated, it advances the nonce by 1, i.e., `nonce_[owner]++`. Since this function is defined as `public`, any one could call this function to verify the signature but with the side-effect of advancing the nonce (if successfully verified)!

```
    function permit(address owner, address spender, uint256 amount, uint256 deadline,
        uint8 v, bytes32 r, bytes32 s) public virtual override {
        // solhint-disable-next-line not-rely-on-time
        require(block.timestamp <= deadline, "ERC20Permit: expired deadline");

        bytes32 structHash = keccak256(
            abi.encode(
                _PERMIT_TYPEHASH,
                owner,
                spender,
                amount,
                _nonces[owner],
                deadline
            )
        );

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _nonces[owner]++;
        _approve(owner, spender, amount);
    }
```

Listing 3.8: ERC20Permit::permit()

Here comes the problem: when an user invokes `depositWithPermit()` to perform specified actions by signing the transaction offline, but before the transaction is mined, it is possible for a malicious actor to observe it (by closely monitoring the transaction pool) and then possibly front-runs it by crafting a new transaction and offering a higher gas fee for block inclusion. The new transaction may perform a fresh `permit()` call. If the front-running is successful, the crafted transaction essentially advances the `nonce` by 1, effectively invalidating the user transaction that is being front-run.

Recommendation This is a common issue inherent in current blockchain infrastructure. However, its impact is rather limited in causing frictions without actual damages.

Status This issue has been confirmed.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Ruler` protocol that is a market driven lending platform that provides non-liquidatable and fungible loans. During the audit, we notice that 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.

PeckShield

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[7] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

PeckShield