Dinari Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, 0xCiphky, merlinboii

December 7th - December 11th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **dinaricrypto/usdplus-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Dinari

Dinari allows users to trade tokens backed by US securities using stablecoins. USD+ is a stablecoin backed by short-term US treasuries offering a stable value and monthly yield distributions. The process involves minting USD+ with USDC and redeeming it through a smart contract, with liquidity ensuring redemptions are completed in up to 3 days.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 1ed01b15d1f69828d8320c5384175c30cdd14377

fixes review commit hash - 1de4ba609a7ef187c75b0e5be53da3fa73479feb

Scope

The following smart contracts were in scope of the audit:

- `ERC7281Min`
- `SelfPermit`
- `TransferRestrictor`
- `UsdPlus`
- `UsdPlusMinter`
- `UsdPlusRedeemer`
- `WrappedUsdPlus`

7. Executive Summary

Over the course of the security review, 0xunforgiven, 0xCiphky, merlinboii engaged with Dinari to review Dinari. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Dinari
Repository	https://github.com/dinaricrypto/usdplus-contracts
Date	December 7th - December 11th
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
High	1
Medium	5
Low	5
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect handling of token decimals	High	Resolved
[M-01]	rebaseMul() not working as intended	Medium	Resolved
[M-02]	Attacker can DOS privateMint()	Medium	Resolved
[M-03]	The deprecated privateMint() prone to many issues	Medium	Resolved
[M-04]	Lack of data validation for oracle price feed	Medium	Resolved
[M-05]	Users can frontrun rebase functions	Medium	Acknowledged
[L-01]	Code would behave unexpectedly if the price reaches 0	Low	Resolved
[L-02]	Overriding function is missing "override" specifier	Low	Resolved
[L-03]	Reducing the limit from the maximum takes the full duration to replenish	Low	Resolved
[L-04]	Functions do not have slippage protection	Low	Resolved
[L-05]	burn() should round up when calculating share amounts	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Incorrect handling of token decimals

Severity

Impact: High

Likelihood: Medium

Description

The `previewDeposit()`, `previewMint()`, `previewWithdraw()` and `previewRedeem()` calculate returned amounts without accounting for the decimals of the `paymentToken`.

This can result in precision discrepancies during operations, leading to incorrect amounts being processed.

Assume the following scenario:

- The `paymentToken` has `18` decimals (e.g., `DAI`).
- The `USD`+ token has `6` decimals.

Impact on using each functions:

- `previewDeposit()`:

The `usdPlusAmount` to mint is calculated using the `paymentToken` decimals (18), resulting in an overestimated USD+ amount for the user (scaled up by `1e12`).

- `previewMint()`

The `paymentTokenAmount` required to mint is calculated using the USD+ decimals (6), causing the user to pay less than intended to mint USD+.

- `previewWithdraw()`

  The transaction may revert due to insufficient funds, as the returned `usdPlusAmount` is calculated in 18 decimals, representing an unrealistically large amount.

- `previewRedeem()`

  The `paymentTokenAmount` is tracked with the smaller USD+ decimals (6), leading to users receiving less value than expected when their redemption is fulfilled.

Recommendations

Consider handling accepted payment token decimals across all calculations.

8.2. Medium Findings

[M-01] `rebaseMul()` not working as intended

Severity

Impact: Low

Likelihood: High

Description

Function `rebaseMul()` increases the balance per share value by a factor. The issue is that the code only accepts integer factors and it will not be possible to increase the balance per share by small percentages.

```
function rebaseMul(uint128 factor) external onlyRole(OPERATOR_ROLE) {
        uint128 _balancePerShare = balancePerShare() * factor;
        UsdPlusStorage storage $ = _getUsdPlusStorage();
        $._balancePerShare = _balancePerShare;
        emit BalancePerShareSet(_balancePerShare);
    }
```

Recommendations

Use the denominator and nominator to specify the increase factor.

[M-02] Attacker can DOS `privateMint()`

Severity

Impact: Medium

Likelihood: Medium

Description

Function `privateMint()` calls `permit()` to set the allowance before token transfer:

```
IERC20Permit(address(paymentToken)).permit(permit.owner, address
          (this), permit.value, permit.deadline, v, r, s);
        usdPlusAmount = permit.value;

        _issue(
          paymentToken,
          permit.value,
          permit.value,
          permit.owner,
          permit.owner
        );
```

The issue is that the attacker can watch the mempool and use the signature to call permit and it would cause the original transaction to revert because of the used nonce. As a result, the attacker can DOS the calls to `privateMint()` function. The same issue exists in `selfPermit()` function too.

Recommendations

Use try/catch when calling permit and if the current allowance is bigger than the spending amount allow logic to be executed. link

[M-03] The deprecated `privateMint()` prone to many issues

Severity

Impact: High

Likelihood: Low

Description

The `privateMint()` is marked as a deprecated function. However, this function is prone to several issues:

1. The `splitSignature()`, which is only used for `privateMint()`, does not work with short signatures (64-length signature): link

2. The direct value for USD+ minting (`usdPlusAmount`) is used without performing oracle conversion to reflect the real amount especially when the paymentToken is not USD+ (e.g., USDC, which also has permit

functionalities). This can lead to over- or under-pricing of USD tokens at the time of minting.

Recommendations

Consider removing the deprecated function to ensure the protocol operates correctly and reduce attack vectors, or update the function to handle the work correctly:

1. For `splitSignature()`, refer to this for handling short signatures: link
2. Consider using the `previewDeposit()` to calculate the minting value along with the current oracle price of the `paymentToken`.

[M-04] Lack of data validation for oracle price feed

Severity

Impact: High

Likelihood: Low

Description

There are several issues with the validation of price feed data in the `USDPlusMinter` and `USDPlusRedeemer` (when using Chainlink's `latestRoundData()`):

- Missing Validation for Price > 0: There is no check to ensure the price is greater than 0, which could allow invalid price data.
- Price Staleness Not Checked: The contracts do not verify whether the price feed data is up-to-date, risking reliance on stale or outdated prices.
- Sequencer Downtime Handling: The `GRACE_PERIOD` following sequencer downtime is not enforced.

Recommendations

Implement a validation check to ensure the price is valid and fresh. Each token's oracle can have a different heartbeat, so the freshness validation should account for these differences separately. link

For sequencer downtime handling, follow the documentation: link

[M-05] Users can frontrun rebase functions

Severity

Impact: High

Likelihood: Low

Description

The UsdPlus Contract contains the `rebaseAdd`, `rebaseMul`, and `rebaseSub` functions, which allow an address with the `OPERATOR_ROLE` to perform a rebase. These functions adjust the value of `$._balancePerShare`, either increasing or decreasing it depending on the action executed.

```
function rebaseAdd(uint128 value) external onlyRole(OPERATOR_ROLE) {
        uint256 _supply = totalSupply();
        uint128 _balancePerShare = uint128(FixedPointMathLib.fullMulDiv
          (balancePerShare(), _supply + value, _supply));
        UsdPlusStorage storage $ = _getUsdPlusStorage();
        $._balancePerShare = _balancePerShare;
        emit BalancePerShareSet(_balancePerShare);
    }
```

This functionality can be exploited in the following ways:

Users can front-run a positive rebase to acquire tokens at a lower rate. Once the rebase is executed, they can submit a withdrawal request, locking in the higher rate. Although the redemption process can take up to three days, the `request` function locks the rate in the user's request ticket, allowing them to secure profits.

Users can also front-run a negative rebase by submitting a withdrawal request before the rebase is executed, locking in the rate prior to the update. This allows them to avoid losses associated with the negative rebase, effectively bypassing the intended impact of the rebase adjustment.

Recommendations

A small withdrawal fee could be implemented to make such exploit attempts less desirable. Additionally, a delay could also be introduced between deposit and withdrawal requests.

8.3. Low Findings

[L-01] Code would behave unexpectedly if the price reaches 0

The operator can update the share price value by calling `rebaseSub()`. If for some reason protocol loses all its deposits (for example in early deployments) then after the operator calls to `rebaseSub()` the `_balancePerShare` value will be 0 while some users have a positive balance. Then if somebody makes a deposit and mints USD+ token, the code would use _INITIAL_BALANCE_PER_SHARE as the price:

```
function balancePerShare() public view override returns (uint128) {
        // Override with default if not set due to upgrade
        if (_balancePerShare == 0) return _INITIAL_BALANCE_PER_SHARE;
        return _balancePerShare;
    }
```

As a result, the depositor's balance would be less than what he deposited and those old depositors who had 0 balance would get a higher balance value. The issue is that the code assumes if `_balancePerShare` is 0 then the code is in the beginning state. To avoid this issue it's better to let the share price always be higher than 0 when calling `rebaseSub()`.

[L-02] Overriding function is missing "override" specifier

The functions `mintingMaxLimitOf()`, `burningMaxLimitOf()`, `mintingCurrentLimitOf()`, and `burningCurrentLimitOf()` do not include the override specifier, even though they are inherited from the `IERC7281Min` interface.

[L-03] Reducing the limit from the maximum takes the full duration to

replenish

The `_changeMintingLimit()` and `_changeBurningLimit()` incorrectly wait for the full replenishment duration before the new limit can be fully used.

When the limit is reduced from `type(uint256).max` to any new limit, the function uses `type(uint256).max` as the `oldMaxLimit` to calculate the current and new limits. This approach results in the new current limit being returned as 0, 1-day replenishment before the limit is available to use.

The limit `type(uint256).max` is a special case that bypasses the checks and allows the limit to be used immediately. In contrast, when the limit is set to any other value for the first time, it can be used immediately without waiting for the full 1-day replenishment duration.

```
function _calculateNewCurrentLimit
  (uint256 newMaxLimit, uint256 oldMaxLimit, uint256 currentLimit)
    internal
    pure
    returns (uint256)
{
    if (newMaxLimit > oldMaxLimit) {
        return currentLimit + (newMaxLimit - oldMaxLimit);
    }
    uint256 difference = oldMaxLimit - newMaxLimit;
@>  return currentLimit > difference ? currentLimit - difference : 0;
}
```

Recommendations

Consider treating the `oldMaxLimit` separately when it changes from `type(uint256).max` to any other limit. For example, treat it as a new limit being created.

[L-04] Functions do not have slippage protection

During the mint and withdraw process, users specify the token or USD+ amount and the code calculates the receiving or sending amount based on oracle price. The issue is that the deposit/withdraw asset price or USD+ share price may change while the transaction is in mempool and there's no way for the user to specify slippage to protect against price changes and as result user may receive unfair exchange.

```
function mint(IERC20 paymentToken, uint256 usdPlusAmount, address receiver)
 function requestRedeem
   (IERC20 paymentToken, uint256 usdplusAmount, address receiver, address owner)
```

Allow users to specify the minimum/maximum amounts during the deposit/withdraw process.

[L-05] `burn()` should round up when calculating share amounts

Function `burn()` in UsdPlus contract, burns specified USD+ tokens from the caller. It has been used during the withdrawal process.

```
/// @notice burn USD+ from msg.sender
    function burn(uint256 value) external {
        address from = _msgSender();
        _useBurningLimits(from, value);
        _burn(from, value);
    }
```

Because USD+ is a rebasing token to calculate the share burn amount code use `_balancePerShare` and divide the USD+ amount by this value.

```
function balanceToShares(uint256 balance) public view returns (uint256) {
        return mulDiv(balance, _INITIAL_BALANCE_PER_SHARE, balancePerShare
        //()); // floor
    }
```

The issue is that by doing this the share amount would be rounded down and as a result code would burn less than what the caller specified. This would cause small profit for the caller and in some circumstances, malicious users can extract value because of this.

Round up in favor of the contract when calculating the share amount in `burn()`.