Opentrade

Audit

Presented by:

OtterSec contact@osec.io

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Opentrade engaged OtterSec to perform an assessment of the `ot-perimeter-protocol` program. This assessment was conducted between July 26th and September 4th, 2023. Follow-up reviews were performed up until August 29th, 2024. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 9 findings in total.

In particular, we found a critical vulnerability relating to the lack of address validation during the loan rollover process, where the argument containing the address for the prior loan is not adequately checked to ensure whether it indeed points to a valid loan contract or not. This may allow the pool admin to introduce a custom contract capable of altering specific loan parameters, impacting the integrity of the rollover process, and enabling the admin to benefit from manipulating the distribution of assets (OS-OTD-ADV-00).

Additionally, we identified another issue concerning the formula employed to calculate a particular timestamp, which specifies the deadline for requesting early redemption of assets from the loan (OS-OTD-ADV-01) and also highlighted the lack of access control mechanism in a certain functionality (OS-OTD-ADV-02).

We also provided recommendations concerning an inaccurate check in the withdraw Operations for ERC-20 tokens (OS-OTD-SUG-00) and proposed minor code adjustments to the code base (OS-OTD-SUG-01).

02 | Scope

The source code was delivered to us in a git repository at github.com/tomniermann/ot-perimeter-protocol. This audit was performed against branch OtterSecAudit and commit bf4f0fb. We did followup reviews up until 792cba6.

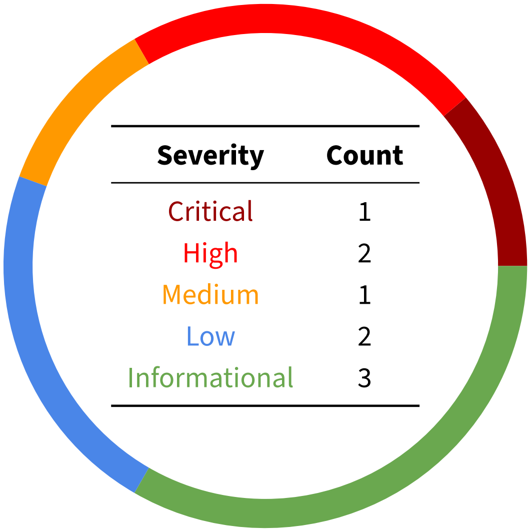
A brief description of the programs is as follows.

Name	Description
ot-perimeter-protocol	The protocol is comprised of an open-source collection of standards designed to facilitate the smooth transfer of stablecoin capital on secure, open, and permissionless networks.

03 | Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	1
High	2
Medium	1
Low	2
Informational	3

04 Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-OTD-ADV-00	Critical	Resolved	Lack of address validation for `priorLoan` contract address before initiating the rollover process.
OS-OTD-ADV-01	High	Resolved	Erroneous calculation within `Loan`.
OS-OTD-ADV-02	High	Resolved	Improper Access Control Modifiers in some functions of `WithdrawControllerFlex` and `PoolFlex`.
OS-OTD-ADV-03	Medium	Resolved	Generated Withdraw Event ID may experience collisions.
OS-OTD-ADV-04	Low	Resolved	Incorrect usage of `updateNonBusinessDays` in `PoolFlex` results in constant reversion.
OS-OTD-ADV-05	Low	Resolved	An oversight in `releaseWithdrawal` causes an invalid state to be stored in `IPoolLenderTotals`.

OS-OTD-ADV-00 [crit] | Loan Address Validation

Description

initiateRollover within pool is responsible for initiating the rollover process for a loan within the liquidity pool. As showcased in the code below, this method internally calls PoolLib::calculateRollover, which takes in priorLoan to calculate the assetsFromPriorToNextLoan value, which represents the assets that must be transferred from the prior loan to the new one.

contracts/Pool.sol SOLIDITY

```solidity
function initiateRollover(
        address loan,
        address priorLoan
    ) external onlyNotPaused onlyPoolController {
        uint256 _outstandingLoanPrincipals;
        uint256 assetsFromPool;
        uint256 assetsFromPriorToNextLoan;
        uint256 totalSupply;
        uint256 assetToAReturnToPool;
        (
            _outstandingLoanPrincipals,
            assetsFromPool,
            assetsFromPriorToNextLoan,
            totalSupply,
            assetToAReturnToPool
        ) = PoolLib.calculateRollover(
            priorLoan,
            address(_liquidityAsset),
            address(this),
            _accountings.outstandingLoanPrincipals
        );
    [...]
}
```

The vulnerability arises due to initiateRollover not performing validation to confirm whether the priorLoan address corresponds to a valid Loan contract. Consequently, this oversight may result in a scenario where a malicious PoolAdmin manipulates the priorLoan contract, returning incorrect values while calculating the rollover parameters.

Thus, the malicious PoolAdmin may provide their own address as the priorLoan contract, possessing custom behavior that is intentionally crafted to manipulate the values returned during the rollover calculation; this subsequently results in an inaccurate distribution of assets and shares between the old and new loans, ultimately favoring the admin's interests.

Proof of Concept

1. A malicious `PoolAdmin` calls `initiateRollover` with `priorLoan` set to an address which points to a custom contract made by them.
2. This address for `priorLoan` is passed to `PoolLib::calculateRollover` alongside other required parameters.
3. In `PoolLib::calculateRollover`, while calculating `assetsFromPriorToNextLoan`, it recieves the values for `principal` and `interest` from the `priorLoan` contract.
4. The `priorLoan` contract returns incorrect values of `principal` and `interest`, resulting in `assetsFromPriorToNextLoan` being significantly higher than expected, ultimately benefiting the `PoolAdmin`.

Remediation

Ensure `initiateRollover` includes adequate validation checks to confirm that the `priorLoan` address points to a valid `Loan` contract with the expected behavior.

contracts/Pool.sol SOLIDITY

```solidity
function initiateRollover(
        address loan,
        address priorLoan
    ) external onlyNotPaused onlyPoolController {
        require( loan == requestedLoan && priorLoan == activeLoan, "Pool: invalid
    ↪   loan");
        uint256 _outstandingLoanPrincipals;
        uint256 assetsFromPool;
        uint256 assetsFromPriorToNextLoan;
        uint256 totalSupply;
        uint256 assetToAReturnToPool;
        (
            _outstandingLoanPrincipals,
            assetsFromPool,
            assetsFromPriorToNextLoan,
            totalSupply,
            assetToAReturnToPool
        ) = PoolLib.calculateRollover(
            priorLoan,
            address(_liquidityAsset),
            address(this),
            _accountings.outstandingLoanPrincipals
        );
    [...]
}
```

Patch

Fixed in e20b41a.

OS-OTD-ADV-01 [high] | Incorrect Timestamp Calculation

Description

`calculateSchedule` within `Loan` is crucial in determining and configuring several critical timestamps that delineate different phases within the loan life cycle.

contracts/Loan.sol SOLIDITY

```solidity
function calculateSchedule() internal {
    [...]
    earlyRedeemRequestClosingTimestamp =
        accrualStartTimestamp +
        (settings.durationDays -
            settings.transferInWindowDurationDays -
            settings.transferInWindowDurationDays) *
        (1 days);
    [...]
}
```

The vulnerability emerges from a calculation of `earlyRedeemRequestClosingTimestamp` within `calculateSchedule`. This timestamp represents the deadline for requesting early redemption of assets from the loan. The issue is present in the flawed formula utilized for calculations. Specifically, it subtracts `transferInWindowDurationDays` from itself, as shown in the above code snippet, resulting in a redundant operation and assigning an incorrect value to the timestamp.

Remediation

Change the formula utilized for the calculation of `earlyRedeemRequestClosingTimestamp` such that `transferInWindowDurationDays` is added to `transferOutWindowDurationDays`.

contracts/Loan.sol SOLIDITY

```solidity
function calculateSchedule() internal {
    [...]
    earlyRedeemRequestClosingTimestamp =
        accrualStartTimestamp +
        (settings.durationDays -
            settings.transferInWindowDurationDays -
            settings.transferOutWindowDurationDays) *
        (1 days);
    [...]
}
```

Patch

Fixed in 6249748.

OS-OTD-ADV-02 [high] | Improper Access Control Modifier

Description

`WithdrawControllerFlex` and `PoolFlex` define public functions intended to manage the flow of the Flexible Term USD Vault. However, the implementation of some of these functions exhibits a similar oversight. Specifically, this oversight relates to the lack of appropriate access control modifiers on certain functions within these components. Below is the list of functions where this issue was identified:

- `PoolFlex.sol`
  - `feesPaidDown`
- `WithdrawControllerFlex.sol`
  - `drawDownToBorrowerWallet`
  - `deposit`
  - `repayLoans`
  - `releaseWithdrawal`

As a result, any user may call these functions at any time, regardless of their authentication status or permission levels. Unauthorized access to these functions may result in unauthorized actions and manipulation of the contracts' stored state.

Remediation

Add the correct modifiers to the functions.

Patch

Fixed in a6873ca and 58a65db.

OS-OTD-ADV-03 [med] | Event ID Collision

Description

When a lender wants to request the borrower to repay some of the loan, they will call `requestRedeem`, which eventually calls `performRequest` defined in `WithdrawControllerFlex`. This action will store the request event as `IPoolLenderWithdrawEvent` and save it in the contract's state. This event is identified with an ID called `eventId`, generated by `generateWithdrawEventId`.

contracts/controller/WithdrawDepositControllerFlex.sol SOLIDITY

```solidity
function generateWithdrawEventId() internal view returns (uint256) {
    return
        uint256(keccak256(abi.encodePacked(block.timestamp, msg.sender)));
}
```

However, this generation code does not guarantee a unique `eventId` each time it is called, especially when the same lender executes multiple `performRequest` actions within a single block. This may result in an ID collision.

Remediation

Change the generation parameters by adding a new parameter guaranteed to be unique for each call.

Patch

Fixed in 3cee443.

OS-OTD-ADV-04 [low] | Incorrect Function Usage

Description

BusinessDayRegistry contains a mapping called _isHoliday which stores non-business days of the protocol. Admins may update these by calling updateNonBusinessDays defined in PoolFlex.sol, which eventually will call the function in WithdrawControllerFlex.sol named updateNonBusinessDays.

contracts/PoolFlex.sol SOLIDITY

```solidity
function updateNonBusinessDays(uint64[] memory _timestamps) external {
    updatePoolData();
    return businessDayRegistry.updateNonBusinessDays(_timestamps);
}
```

contracts/BusinessDayRegistry.sol SOLIDITY

```solidity
function updateNonBusinessDays(
    uint64[] memory _timestamps
) public onlyPoolAdmin {
    [...]
}
```

However, updateNonBusinessDays in BusinessDayRegistry.sol has a modifier that restricts its execution to the pool's admin. This call will invariably revert since PoolFlex is not an admin.

Remediation

Change the modifier of updateNonBusinessDays in BusinessDayRegistry to onlyPool, and add the onlyPoolAdmin modifier to updateNonBusinessDays in PoolFlex.

Patch

Fixed in f360fe8.

OS-OTD-ADV-05 [low] | Invalid Account State

Description

When a lender requests the borrower to repay their loans, the contract updates certain states related to the request and stores them in the mapping of `IPoolLenderTotals`. These states should be cleared via `releaseWithdrawal` once the request is resolved.

contracts/controller/WithdrawDepositControllerFlex.sol SOLIDITY

```solidity
function releaseWithdrawal(
    uint256 eventId
) public returns (IPoolLenderWithdrawEvent memory ev) {
    ev = findEventAndRemove(eventId);

    _dailyWithdrawTotals[ev.transferOutDayTimestamp].requestedAssets -= ev
        .requestedAssets;
    _dailyWithdrawTotals[ev.transferOutDayTimestamp].requestedShares -= ev
        .requestedShares;

    _poolWithdrawTotals.requestedAssets -= ev.requestedAssets;

    _poolWithdrawTotals.requestedShares -= ev.requestedShares;

    _lenderTotals[ev.lender].requestedAssets -= ev.requestedAssets;

    _lenderTotals[ev.lender].requestedShares -= ev.requestedShares;
    _lenderTotals[ev.lender].assetsWithdrawn += ev.requestedAssets;
}
```

However, `releaseWithdrawal` currently omits to update the state variables `assetsDueForWithdraws` and `sharesDueForWithdraws`, resulting in inconsistency in the stored state.

Remediation

Update `releaseWithdrawal` to properly adjust the `assetsDueForWithdraws` and `sharesDueForWithdraws`.

Patch

Fixed in ea45c91.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-OTD-SUG-00	`withdrawERC20` implements an incorrect check while verifying the `receiver` address.
OS-OTD-SUG-01	Recommendations regarding minor modifications to the code.
OS-OTD-SUG-02	`LoanCreated` is set to `external`, and lacks any restrictions allowing anyone to set `createdLoan`.

OS-OTD-SUG-00 | Inconsistent Receiver Check

Description

In `withdrawERC20`, if the `vaultType` is equal to `BorrowerVault`, it checks if `receiver` matches the borrower vault specified in the associated pool's withdraw controller.

contracts/Vault.sol SOLIDITY

```solidity
function withdrawERC20(
    address asset,
    uint256 amount,
    address receiver
) external override onlyOwner onlyNotPaused {
    require(receiver != address(0), "Vault: 0 add");
    if (vaultType == IVaultType.BorrowerVault) {
        require(
            receiver == IWithdrawController(_owner).borrowerVault(),
            "Vault: Invalid receiver"
        );
    }
    IERC20Upgradeable(asset).safeTransfer(receiver, amount);
    emit WithdrewERC20(asset, amount, receiver);
}
```

The check compares with the borrower vault's address, but in the context of withdrawing ERC-20 tokens, `receiver` may be the vault itself (when withdrawing to the vault rather than the borrower wallet). This suggests that the current check considers the receiver valid if it is the vault itself instead of checking for `borrowerWallet`, resulting in unintended behavior.

Remediation

Ensure the `require` statement in `withdrawERC20` checks the `borrowerWallet` address.

OS-OTD-SUG-01 | Code Refactoring

Description

1. The _pool variable in Vault is declared, but not initialized. When isBorrowerVault is called, the attempt to access IPool(_pool) will result in an uninitialized variable, eventually reverting.
2. In PoolAccessControl, there is a typographical error in the name of the onlyTOSAccpeted modifier where *Accepted* is incorrectly spelled as *Accpeted*.
3. Reconsider the purpose of withdrawERC721 in Vault, as the code base does not utilize ERC721 tokens anywhere.
4. WithdrawDepositControllerFlex contains duplicated functions, namely repayLoans and repayLoan.
5. The transferOutDayTimestamp of dailyWithdrawTotal is assigned the same value twice in the function performRequest.

Remediation

1. Ensure that _pool is properly initialized, either through the constructor or a separate initialization function.
2. Correct the typographical error in PoolAccessControl.
3. Remove withdrawERC721 from Vault if it is unnecessary.
4. Remove one of the duplicated functions.
5. Remove the if block containing the redundant assignment.

OS-OTD-SUG-02 | Unauthorized Loan Assignment

Description

`LoanCreated` within `pool` lacks any access control mechanism and may be called by anyone, enabling them to set the `createdLoan` variable to any address. The only check performed in `LoanCreated` is `if (loan != msg.sender)`, which is insufficient to ensure the caller has permission to set the `createdLoan` variable.

contracts/Pool.sol SOLIDITY

```solidity
function loanCreated(address loan) external {
    if (loan != msg.sender) revert InvalidAccess();
    emit LoanCreated(loan);
    createdLoan = loan;
}
```

Thus, an attacker may abuse this vulnerability to manipulate the `createdLoan` state. The value of `createdLoan` is crucial for tracking the state of the contract, especially in the context of loans. Unauthorized modifications may result in a loss of data integrity and compromise the reliability of the contract's state.

Remediation

Implement an access restriction for `LoanCreated`, such that only the authorized addresses are able to invoke it.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical

Vulnerabilities that immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority or access control validation
- Improperly designed economic incentives leading to loss of funds

High

Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions
- Exploitation involving high capital requirement with respect to payout

Medium

Vulnerabilities that could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input that causes computational limit exhaustion
- Forced exceptions in normal user flow

Low

Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants
- Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.