QuuLab

Sense Finance
Point Tokenization Vault

Security Assessment &
Formal Verification
Nov 5th, 2025

Sense Finance Point Tokenization Vault - Security Assessment & Formal Verification

Project Overview

Project Summary

Project Name	Sense Finance Point Tokenization Vault
Language	Solidity
Codebase	https://github.com/sense-finance/point-tokenization-vault

Project Description

Sense Finance Point Tokenization Vault allows users to make deposits, claim `PToken` tokens and redeem them for rewards.

Audit Overview

Audit Summary

Delivery Date	Nov 5, 2025
Audit Methodology	Manual Review, Static Analysis, Formal Verification
Final Commit	2e607155b234dcb69d1fbbe126f930bed779c134
Formal Verification Report	PToken.sol, PointTokenVault.sol
Formal Verification CI Setup	PR URL

Audit Scope

Filename	URL
`PToken.sol`	https://github.com/sense-finance/point-tokenization-vault/blob/2e607155b234dcb69d1fbbe126f930bed779c134/contracts/PToken.sol
`PointTokenVault.sol`	https://github.com/sense-finance/point-tokenization-vault/blob/2e607155b234dcb69d1fbbe126f930bed779c134/contracts/PointTokenVault.sol

Severity Matrix

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

Impact

- **High** - results in a considerable risk that may jeopardize the protocol's overall integrity, impacting all or the majority of users.
- **Medium** - results in a non-critical risk for the protocol, impacting either all users or a specific subset, yet remaining unequivocally unacceptable.
- **Low** - losses incurred will be within acceptable limits, attack vectors can be fixed with relative ease.

Likelihood

- **High** - highly probable, presenting significant financial opportunities for exploitation by malicious actors.
- **Medium** - still relatively probable, although contingent upon certain conditions.
- **Low** - requires a unique set of conditions and presents a cost of execution that does not yield a favorable ratio of rewards for the individual involved.

Findings Summary

Severity	Discovered
Critical	-
High	-
Medium	4
Low	1
Total	5

Findings

ID	Title	Severity
M-01	Operator can brick `PointTokenVault` for a particular `PToken`	Medium
M-02	User may be unable to withdraw deposit if reward and deposit tokens are equal	Medium
M-03	Infinite `PToken` mint in a corner case	Medium
M-04	Some "weird" ERC20 tokens are not supported	Medium
L-01	ETH can be stuck in the contract	Low

Certora Formal Verification

Overview

Contract	Report URL
PToken.sol	Report URL
PointTokenVault.sol	Report URL

Properties (PToken.sol)

Property Description	Type	Passed
Total supply is sum of all balances	High	✅
Total supply never overflows	High	✅
Max number of balance changes in a single call is 2	High	✅
Only `approve()` and `transferFrom()` can change allowance	High	✅
User balance may be changed only by: `mint()` , `burn()` , `transfer()` , `transferFrom()`	High	✅
Only `mint()` and `burn()` can change total supply	High	✅
Account's balance can be reduced only by token holder or approved 3rd party	High	✅
Only token holder can increase allowance, spender can decrease it by using it	High	✅
`mint()` updates storage as expected	Unit	✅
`mint()` reverts when expected	Unit	✅
`mint()` does not affect 3rd party	Unit	✅
`burn()` updates storage as expected	Unit	✅
`burn()` reverts when expected	Unit	✅
`burn()` does not affect 3rd party	Unit	✅
`transfer()` updates storage as expected	Unit	✅
`transfer()` of a single huge amount works the same as 2 transfers of small amounts	Unit	✅
`transfer()` reverts when expected	Unit	✅
`transfer()` does not affect 3rd party	Unit	✅
`transferFrom()` updates storage as expected	Unit	✅
`transferFrom()` reverts when expected	Unit	✅
`transferFrom()` does not affect 3rd party	Unit	✅
`transferFrom()` of a single huge amount works the same as 2 transfers of small amounts	Unit	✅
`approve()` updates storage as expected	Unit	✅
`approve()` reverts when expected	Unit	✅

Property Description	Type	Passed
`approve()` does not affect 3rd party	Unit	✅
`pause()` updates storage as expected	Unit	✅
`pause()` reverts when expected	Unit	✅
`unpause()` updates storage as expected	Unit	✅
`unpause()` reverts when expected	Unit	✅

Properties (PointTokenVault.sol)

Property Description	Type	Passed
Methods are called by expected roles	High	✅
`deposit()` updates storage as expected	Unit	✅
`deposit()` reverts when expected	Unit	✅
`deposit()` does not affect other entities	Unit	✅
`withdraw()` updates storage as expected	Unit	✅
`withdraw()` reverts when expected	Unit	✅
`withdraw()` does not affect other entities	Unit	✅
`claimPTokens()` updates storage as expected	Unit	✅
`claimPTokens()` reverts when expected	Unit	✅
`claimPTokens()` does not affect other entities	Unit	✅
`redeemRewards()` updates storage as expected	Unit	✅
`redeemRewards()` reverts when expected	Unit	✅
`redeemRewards()` does not affect other entities	Unit	✅
`convertRewardsToPTokens()` updates storage as expected	Unit	✅
`convertRewardsToPTokens()` reverts when expected	Unit	✅
`convertRewardsToPTokens()` does not affect other entities	Unit	✅
`trustReceiver()` updates storage as expected	Unit	✅
`trustReceiver()` reverts when expected	Unit	✅
`deployPToken()` reverts when expected	Unit	✅
`updateRoot()` updates storage as expected	Unit	✅
`updateRoot()` reverts when expected	Unit	✅
`setCap()` updates storage as expected	Unit	✅
`setCap()` reverts when expected	Unit	✅

Property Description	Type	Passed
`setRedemption()` updates storage as expected	Unit	✅
`setRedemption()` reverts when expected	Unit	✅
`setMintFee()` updates storage as expected	Unit	✅
`setMintFee()` reverts when expected	Unit	✅
`setRedemptionFee()` updates storage as expected	Unit	✅
`setRedemptionFee()` reverts when expected	Unit	✅
`pausePToken()` updates storage as expected	Unit	✅
`pausePToken()` reverts when expected	Unit	✅
`unpausePToken()` updates storage as expected	Unit	✅
`unpausePToken()` reverts when expected	Unit	✅
`renouncePauseRole()` updates storage as expected	Unit	✅
`renouncePauseRole()` reverts when expected	Unit	✅
`collectFees()` updates storage as expected	Unit	✅
`collectFees()` reverts when expected	Unit	✅
`collectFees()` does not affect other entities	Unit	✅
`setFeeCollector()` updates storage as expected	Unit	✅
`setFeeCollector()` reverts when expected	Unit	✅

Findings

[M-01] Operator can brick `PointTokenVault` for a particular `PToken`

Description

Operator is able to pause a particular `PToken` contract and renounce the `PAUSE_ROLE` from the `PointTokenVault` contract
which will cause DoS for most of the methods.

Example:

1. Operator pauses a particular `PToken`
2. Operator removes the `PAUSE_ROLE` from the `PointTokenVault` contract

At this point operator can no longer unpause the `PToken` since the `PointTokenVault` contract doesn't have the `PAUSE_ROLE` anymore.

Additionally the following methods would always revert since `PToken` is paused and `PointTokenVault` is missing the `PAUSE_ROLE` :

- PToken.mint()
- PToken.burn()
- PToken.transferFrom()
- PToken.transfer()
- PointTokenVault.claimPTokens()
- PointTokenVault.redeemRewards()
- PointTokenVault.convertRewardsToPTokens()
- PointTokenVault.pausePToken()
- PointTokenVault.unpausePToken()
- PointTokenVault.collectFees()

PoC

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity =0.8.24;

import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {Test, console, console2} from "forge-std/Test.sol";
import {MockERC20, ERC20} from "solmate/test/utils/mocks/MockERC20.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib.sol";
import {PToken} from "../../PToken.sol";
import {PointTokenVault} from "../../PointTokenVault.sol";

contract ProtocolTest is Test {
    PointTokenVault pointTokenVaultImplementation;
    ERC1967Proxy proxy;
    PointTokenVault pointTokenVault;

    MockERC20 usdtToken;
    MockERC20 rewardToken;
    address admin = makeAddr("admin");
    address feeCollector = makeAddr("feeCollector");
    address user = makeAddr("user");
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    function setUp() public {
        pointTokenVaultImplementation = new PointTokenVault();
        bytes memory initData = abi.encodeWithSignature("initialize(address,address)", admin,
feeCollector);
        proxy = new ERC1967Proxy(address(pointTokenVaultImplementation), initData);
        pointTokenVault = PointTokenVault(payable(address(proxy)));

        usdtToken = new MockERC20("USDT", "USDT", 18);
        usdtToken.mint(user, 100 ether);

        rewardToken = new MockERC20("RWD", "RWD", 18);
        rewardToken.mint(address(pointTokenVault), 100 ether);

        vm.startPrank(user);
        usdtToken.approve(address(pointTokenVault), type(uint256).max);
        rewardToken.approve(address(pointTokenVault), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(admin);
        pointTokenVault.grantRole(pointTokenVault.OPERATOR_ROLE(), admin);
        pointTokenVault.grantRole(pointTokenVault.MERKLE_UPDATER_ROLE(), admin);
        pointTokenVault.setCap(address(usdtToken), type(uint256).max);
        pointTokenVault.setMintFee(0.1e18); // 10%
        pointTokenVault.setRedemptionFee(0.1e18); // 10%
        pointTokenVault.setFeeCollector(feeCollector);
        vm.stopPrank();
    }

    /**
     * Scenario:
     * 1. `OPERATOR_ROLE` pauses pToken
     * 2. `OPERATOR_ROLE` calls `renouncePauseRole()` which removes `PAUSE_ROLE` from
`PointTokenVault`
     * 3. `OPERATOR_ROLE` tries to unpause pToken which reverts with
`AccessControlUnauthorizedAccount` since
     * `PointTokenVault` doesn't have the `PAUSE_ROLE` anymore thus causing DoS for:
     * - PToken.mint()
     * - PToken.burn()
     * - PToken.transferFrom()
     * - PToken.transfer()
     * - PointTokenVault.claimPTokens()
     * - PointTokenVault.redeemRewards()
     * - PointTokenVault.convertRewardsToPTokens()
     * - PointTokenVault.pausePToken()
     * - PointTokenVault.unpausePToken()
     * - PointTokenVault.collectFees()
     */
    function testRenouncePauseRole() public {
        // deploy PToken
        bytes32 pTokenId = keccak256("pointsId1");
        PToken pToken = pointTokenVault.deployPToken(pTokenId);

        vm.startPrank(admin);
        pointTokenVault.pausePToken(pTokenId);
        pointTokenVault.renouncePauseRole(pTokenId);
        // reverts with `AccessControlUnauthorizedAccount` causing DoS for most of `PToken` and
`PointTokenVault` methods
        pointTokenVault.unpausePToken(pTokenId);
        vm.stopPrank();
    }
}
```

Recommendation

In `PToken` constructor grant `DEFAULT_ADMIN_ROLE` to `msg.sender` . This way, in case of emergency, `PointTokenVault` could be upgraded to manage `PToken` roles.

[M-02] User may be unable to withdraw deposit if reward and deposit tokens are equal

Description

If deposit and reward tokens are equal user may be unable to call withdraw().

Example:

1. User1 deposits 100 USDT
2. Admins sets redemption params so that deposit and reward tokens are the same
3. User2 buys 100 PTokens on secondary market
4. User2 calls `redeemRewards()` burning 100 PTokens for 90 USDT (-10% redemption fee)
5. User1 is unable to call `withdraw()` since there's not enough funds

PoC

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity =0.8.24;

import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {Test, console, console2} from "forge-std/Test.sol";
import {MockERC20, ERC20} from "solmate/test/utils/mocks/MockERC20.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib.sol";
import {PToken} from "../../PToken.sol";
import {PointTokenVault} from "../../PointTokenVault.sol";

contract ProtocolTest is Test {
    PointTokenVault pointTokenVaultImplementation;
    ERC1967Proxy proxy;
    PointTokenVault pointTokenVault;

    MockERC20 usdtToken;
    MockERC20 rewardToken;
    address admin = makeAddr("admin");
    address feeCollector = makeAddr("feeCollector");
    address user = makeAddr("user");
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    function setUp() public {
        pointTokenVaultImplementation = new PointTokenVault();
        bytes memory initData = abi.encodeWithSignature("initialize(address,address)", admin,
feeCollector);
        proxy = new ERC1967Proxy(address(pointTokenVaultImplementation), initData);
        pointTokenVault = PointTokenVault(payable(address(proxy)));

        usdtToken = new MockERC20("USDT", "USDT", 18);
        usdtToken.mint(user, 100 ether);

        rewardToken = new MockERC20("RWD", "RWD", 18);
        rewardToken.mint(address(pointTokenVault), 100 ether);

        vm.startPrank(user);
        usdtToken.approve(address(pointTokenVault), type(uint256).max);
        rewardToken.approve(address(pointTokenVault), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(admin);
        pointTokenVault.grantRole(pointTokenVault.OPERATOR_ROLE(), admin);
        pointTokenVault.grantRole(pointTokenVault.MERKLE_UPDATER_ROLE(), admin);
        pointTokenVault.setCap(address(usdtToken), type(uint256).max);
        pointTokenVault.setMintFee(0.1e18); // 10%
        pointTokenVault.setRedemptionFee(0.1e18); // 10%
        pointTokenVault.setFeeCollector(feeCollector);
        vm.stopPrank();
    }

    /**
     * Scenario:
     * 1. User1 deposits 100 USDT
     * 2. Admins sets redemption params so that deposit and reward tokens are the same
     * 3. User2 buys 100 PTokens on secondary market
     * 4. User2 calls `redeemRewards()` burning 100 PTokens for 90 USDT (-10% redemption fee)
     *.5. User1 is unable to call `withdraw()` since there's not enough funds
     */
    function testRewardAndDepositTokensAreTheSame() public {
        // deployPToken
        bytes32 pTokenId = keccak256("pointsId1");
        PToken pToken = pointTokenVault.deployPToken(pTokenId);

        // deposit
        vm.prank(user);
        pointTokenVault.deposit(usdtToken, 100 ether, user);

        // setRedemption
        vm.prank(admin);
        pointTokenVault.setRedemption(pTokenId, usdtToken, 1 ether, false);

        // user2 buys 100 PTokens on secondary market
        deal(address(pToken), user2, 100 ether);

        // redeemRewards
        bytes32[] memory proof = new bytes32[](1);
        proof[0] = "";
        PointTokenVault.Claim memory claim = PointTokenVault.Claim({
            pointsId: pTokenId,
            totalClaimable: 100 ether,
            amountToClaim: 100 ether,
            proof: proof
        });
        vm.prank(user2);
        pointTokenVault.redeemRewards(claim, user2);

        // withdraw reverts with "not enough funds"
        vm.prank(user);
        vm.expectRevert();
        pointTokenVault.withdraw(usdtToken, 100 ether, user);
    }
}
```

Recommendation

Don't allow deposit and reward tokens to be equal.

[M-03] Infinite `PToken` mint in a corner case

Description

If reward token is a `PToken` then, on certrain conditions, user can infinitely mint `PTokens` .

Example:

1. Admin sets `PToken` as a reward token with `0.5 ether` as a reward per `PToken`
2. User calls `convertRewardsToPTokens()` , transfers 10 `PTokens` and gets 20 `PTokens` minted
3. At his point user can repeat steps 1 and 2 which is basically an infinite mint

PoC

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity =0.8.24;

import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {Test, console, console2} from "forge-std/Test.sol";
import {MockERC20, ERC20} from "solmate/test/utils/mocks/MockERC20.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib.sol";
import {PToken} from "../../PToken.sol";
import {PointTokenVault} from "../../PointTokenVault.sol";

contract ProtocolTest is Test {
    PointTokenVault pointTokenVaultImplementation;
    ERC1967Proxy proxy;
    PointTokenVault pointTokenVault;
    MockERC20 usdtToken;
    MockERC20 rewardToken;

    address admin = makeAddr("admin");
    address feeCollector = makeAddr("feeCollector");
    address user = makeAddr("user");
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");

    function setUp() public {
        pointTokenVaultImplementation = new PointTokenVault();
        bytes memory initData = abi.encodeWithSignature("initialize(address,address)", admin,
feeCollector);
        proxy = new ERC1967Proxy(address(pointTokenVaultImplementation), initData);
        pointTokenVault = PointTokenVault(payable(address(proxy)));

        usdtToken = new MockERC20("USDT", "USDT", 18);
        usdtToken.mint(user, 100 ether);

        rewardToken = new MockERC20("RWD", "RWD", 18);
        rewardToken.mint(address(pointTokenVault), 100 ether);

        vm.startPrank(user);
        usdtToken.approve(address(pointTokenVault), type(uint256).max);
        rewardToken.approve(address(pointTokenVault), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(admin);
        pointTokenVault.grantRole(pointTokenVault.OPERATOR_ROLE(), admin);
        pointTokenVault.grantRole(pointTokenVault.MERKLE_UPDATER_ROLE(), admin);
        pointTokenVault.setCap(address(usdtToken), type(uint256).max);
        pointTokenVault.setMintFee(0.1e18); // 10%
        pointTokenVault.setRedemptionFee(0.1e18); // 10%
        pointTokenVault.setFeeCollector(feeCollector);
        vm.stopPrank();
    }

    /**
     * Scenario:
     1. Admin sets PToken as a reward token with `0.5 ether` as a reward per PToken
     2. User calls `convertRewardsToPTokens()`, transfers 10 PTokens and gets 20 PTokens minted
     3. At his point user can repeat steps 1 and 2 which is basically an infinite mint
     */
    function testConvertRewardsToPTokens() public {
        // deployPToken
        bytes32 pTokenId = keccak256("pointsId1");
        PToken pToken = pointTokenVault.deployPToken(pTokenId);

        // setRedemption
        vm.prank(admin);
        pointTokenVault.setRedemption(pTokenId, pToken, 0.5 ether, false);

        deal(address(pToken), user, 10 ether);

        // convertRewardsToPTokens
        vm.startPrank(user);
        pToken.approve(address(pointTokenVault), type(uint256).max);
        pointTokenVault.convertRewardsToPTokens(user, pTokenId, 10 ether);
        vm.stopPrank();

        // user got 20 PTokens for transferring 10 PTokens
        assertEq(pToken.balanceOf(user), 20 ether);
    }
}
```

Recommendation

Don't allow reward token to be `PToken` .

[M-04] Some "weird" ERC20 tokens are not supported

Description

There are many "weird" ERC20 tokens which behave differently from "standard" ERC20 tokens. For example, some have fees on transfer while others are able to rebase token balances.

Some of those tokens must not be used as a deposit or reward token because it breaks the `PointTokenVault` contract leading to DoS or unexpected behavior.

For example, if there're fees on deposit token transfer then there will be a difference between the deposited amount in storage and the actual amount in contract here.

Here's the list of compatibility between the `PointTokenVault` contract and "weird" ERC20 tokens:

	Deposit Token	Reward Token
Reentrant Calls	✅	✅
Missing Return Values	✅	✅
Fee on Transfer	❌	❌
Rebasing	❌	❌
Upgradable Tokens	✅	✅
Flash Mintable Tokens	✅	✅
Tokens with Blocklists	❌	❌
Pausable Tokens	❌	❌
Approval Race Protections	✅	✅
Revert on Approval To Zero Address	✅	✅
Revert on Zero Value Approvals	✅	✅
Revert on Zero Value Transfers	✅	✅
Multiple Token Addresses	✅	✅

	Deposit Token	Reward Token
Low Decimals	✅	✅
High Decimals	✅	✅
transferFrom with src == msg.sender	✅	✅
Non string metadata	✅	✅
Revert on Transfer to the Zero Address	✅	✅
No Revert on Failure	✅	✅
Revert on Large Approvals & Transfers	✅	✅
Code Injection Via Token Name	✅	✅
Unusual Permit Function	✅	✅
Transfer of less than amount	❌	❌
ERC-20 Representation of Native Currency	✅	✅

Recommendation

If you plan to add a new reward token make sure it's compatible with the current `PointTokenVault` contract. Also you could whitelist deposit tokens and allow only the compatible ones.

[L-01] ETH can be stuck in the contract

Description

There's the receive() method which means that contract can accept `ETH`. The thing is that there's no way to withdraw it anyhow so all sent `ETH` will be stuck in the contract.

Recommendation

Remove the receive() method.

Disclaimer

This security review should not be interpreted as providing absolute assurance against potential hacks or exploits. Smart contracts represent a novel technological advancement, inherently associated with various known and unknown risks. The protocol for which this report is prepared indemnifies QuuLab from any liability concerning potential misbehavior, bugs, or exploits affecting the audited code throughout the entirety of the project's life cycle. It is also crucial to recognize that any modifications made to the audited code, including remedial measures for the issues outlined in this report, may inadvertently introduce new complications and necessitate further auditing.

About QuuLab

QuuLab is a Web3 security firm specializing in advanced formal verification tools and comprehensive smart contract audits. Using modern formal verification tools, we identify even the most elusive and intricate bugs within smart contracts and mathematically prove their absence. We integrate formal verification into the standard deployment pipelines of the audited protocols. It helps developers of the audited protocols to reduce the number of bugs in already audited pieces of code, thereby reducing costs for future security assessments.

Learn more about us at quulab.com.