Satlayer
Deposit Contract

Security Assessment &
Formal Verification
Oct 13th, 2025

ryzhak.com

Satlayer Deposit Contract - Security Assessment & Formal Verification

Project Overview

Project Summary

Project Name	Satlayer Deposit Contract
Language	Solidity
Codebase	https://github.com/satlayer/deposit-contract-public

Project Description

Satlayer Deposit Contract allows users to deposit wrapped bitcoin assets in exchange for receipt tokens. Users with active deposits can later migrate their staked assets to the Satlayer mainnet network.

Audit Overview

Audit Summary

Delivery Date	Oct 13, 2025
Audit Methodology	Manual Review, Static Analysis, Formal Verification
Author(s)	Vladimir Ryzhak
Final Commit	1dd233d6dda3991234bc5ed6e0477b22a2570e04
Formal Verification Report	ReceiptToken.sol, SatlayerPool.sol
Formal Verification CI Setup	PR URL

Audit Scope

Filename	URL
`ReceiptToken.sol`	https://github.com/satlayer/deposit-contract-public/blob/1dd233d6dda3991234bc5ed6e0477b22a2570e04/src/ReceiptToken.sol
`SatlayerPool.sol`	https://github.com/satlayer/deposit-contract-public/blob/1dd233d6dda3991234bc5ed6e0477b22a2570e04/src/SatlayerPool.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	-
Low	6
Total	6

Findings

ID	Title	Severity
L-01	Griefing by frontrunning `withdraw()` and `migrate()` methods	Low
L-02	Receipt tokens can be stuck in the `SatlayerPool` contract	Low
L-03	Some "weird" ERC20 tokens are not supported	Low
L-04	When `approve()` returns false the contract should revert	Low
L-05	The `_decimals` property should be immutable	Low
L-06	Error not used	Low

Certora Formal Verification

Overview

Contract	Report URL
ReceiptToken.sol	Report URL
SatlayerPool.sol	Report URL

Properties (ReceiptToken.sol)

Property Description	Type	Passed
Total supply is sum of all balances	High	✅
Balance of `address(0)` is 0	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	✅

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

Properties (SatlayerPool.sol)

Property Description	Type	Passed
`tokenAllowlist` updates are restricted to certain roles and methods	High	✅
`tokenMap` updates are restricted to certain roles and methods	High	✅
`capsEnabled` updates are restricted to certain roles and methods	High	✅
`caps` updates are restricted to certain roles and methods	High	✅
`migrator` updates are restricted to certain roles and methods	High	✅
`eventId` only increases & updates are restricted to certain roles and methods	High	✅
`onlyOwner` protected methods can be called only by contract owner	High	✅
`whenNotPaused` protected methods always revert if contract is paused	High	✅
`depositFor()` updates storage as expected	Unit	✅
`depositFor()` reverts when expected	Unit	✅
`depositFor()` does not affect other entities	Unit	✅
`withdraw()` updates storage as expected	Unit	✅
`withdraw()` reverts when expected	Unit	✅
`withdraw()` does not affect other entities	Unit	✅
`migrate()` updates storage as expected	Unit	✅
`migrate()` reverts when expected	Unit	✅
`migrate()` does not affect other entities	Unit	✅
`setCapsEnabled()` updates storage as expected	Unit	✅
`setCapsEnabled()` reverts when expected	Unit	✅
`setMigrator()` updates storage as expected	Unit	✅
`setMigrator()` reverts when expected	Unit	✅
`addToken()` updates storage as expected	Unit	✅
`addToken()` reverts when expected	Unit	✅
`setTokenStakingParams()` updates storage as expected	Unit	✅
`setTokenStakingParams()` reverts when expected	Unit	✅

Property Description	Type	Passed
`pause()` updates storage as expected	Unit	✅
`pause()` reverts when expected	Unit	✅
`unpause()` updates storage as expected	Unit	✅
`unpause()` reverts when expected	Unit	✅
`recoverERC20()` updates storage as expected	Unit	✅
`recoverERC20()` reverts when expected	Unit	✅
`recoverERC20()` does not affect other entities	Unit	✅
`renounceOwnership()` reverts when expected	Unit	✅

Findings

[L-01] Griefing by frontrunning `withdraw()` and `migrate()` methods

Description

Malicious user can frontrun withdraw() and migrate() methods causing DoS for a user who wants to withdraw or migrate a stake.

Example:

1. User A stakes 100 stake tokens and gets 100 receipt tokens
2. User A tries to withdraw() or migrate() 100 stake tokens
3. User B (malicious) buys 100 receipt tokens on a secondary market, frontruns User A's transaction and withdraws or migrates stake tokens for himself
4. At his point SatlayerPool has 0 stake token balance hence User A's transaction will revert with "out of funds"

Recommendation

Refactor the code to forbid users without a stake to call withdraw() and migrate() methods.

[L-02] Receipt tokens can be stuck in the `SatlayerPool` contract

Description

Consider an example:

1. Owner adds a new stake token (`STK`) and a corresponding receipt token `RCT_0`
2. Owner adds `RCT_0` receipt token as a stake token
3. User stakes 100 `STK` tokens setting the _for parameter to the `SatlayerPool` contract which gets 100 `RCT_0`
4. Those 100 `RCT_0` are stuck in the contract since recoverERC20() reverts with the `TokenAlreadyAdded` error

PoC

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console2.sol";
import {IMigrator} from "../../src/interface/IMigrator.sol";
import {ReceiptToken} from "../../src/ReceiptToken.sol";
import {SatlayerPool} from "../../src/SatlayerPool.sol";

contract MockMigrator is IMigrator {
    function migrate(
        address _user,
        string calldata _destinationAddress,
        address[] calldata _tokens,
        uint256[] calldata _amounts
    ) external {}
```

```
}

contract ProtocolTest is Test {
    ReceiptToken stakeToken;
    ReceiptToken stakeToken2;
    SatlayerPool satlayerPool;
    MockMigrator mockMigrator;

    address owner = makeAddr("owner");
    address user = makeAddr("user");
    address user2 = makeAddr("user2");

    function setUp() public {
        vm.startPrank(owner);

        stakeToken = new ReceiptToken("STK", "STK", 18);
        stakeToken2 = new ReceiptToken("STK_2", "STK_2", 18);

        address[] memory tokensAllowed = new address[](1);
        tokensAllowed[0] = address(stakeToken);
        uint256[] memory caps = new uint256[](1);
        caps[0] = 1000 ether;
        string[] memory names = new string[](1); // receipt token names
        names[0] = "RCT_0";
        string[] memory symbols = new string[](1); // receipt token symbols
        symbols[0] = "RCT_0";
        satlayerPool = new SatlayerPool(tokensAllowed, caps, names, symbols);

        mockMigrator = new MockMigrator();

        vm.stopPrank();
    }

    /**
     * Funds are stuck:
     * 1. Owner adds a new stake token (`STK`) and a corresponding receipt token `RCT_0`
     * 2. Owner adds `RCT_0` token as a stake token
     * 3. User stakes 100 `STK` setting staking recipient to be `SatlayerPool` which gets 100 `RCT_0`
     * 4. Those 100 `RCT_0` are stuck in the contract since `recoverERC20()` reverts with
`TokenAlreadyAdded`
     */
    function test_ReceiptTokensStuck() public {
        vm.prank(owner);
        stakeToken.mint(user, 100 ether);

        // owner adds receipt token as a stake token
        vm.startPrank(owner);
        satlayerPool.addToken(satlayerPool.tokenMap(address(stakeToken)), 1000 ether, "RCT_0",
"RCT_0");
        vm.stopPrank();

        // user stakes 100 STK tokens setting receiver to be `SatlayerPool`
        vm.startPrank(user);
        stakeToken.approve(address(satlayerPool), type(uint256).max);
        satlayerPool.depositFor(address(stakeToken), address(satlayerPool), 100 ether);
```

```
        vm.stopPrank();

        // Owner tries to recover RCT_0 tokens mistakenly sent to `SatlayerPool`.
        // `recoverERC20()` reverts and RCT_0 tokens are stuck in the contract.
        vm.startPrank(owner);
        satlayerPool.recoverERC20(satlayerPool.tokenMap(address(stakeToken)), owner, 100 ether);
        vm.stopPrank();
    }
}
```

Recommendation

In the addToken() method validate that added token is not a receipt token.

[L-03] 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 should not be used as a deposit token in the SatlayerPool contract because it leads to DoS or unexpected behavior.

For example, ERC20 tokens with callbacks (like `ERC777`) will allow to circumvent the cap limit while pausable tokens and tokens with blacklist functionality will DoS users' withdrawals.

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

	Deposit 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
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	✅

And here is the list of compatibility of tokens expected to be used as deposit ones and their features which might not be compatible with the `SatlayerPool` contract:

	pumpBTC	WBTC	uniBTC	LBTC	solvBTC.BBN	waBTC	SBTC	stBTC	FBTC
Reentrant Calls	❎	❎	❎	❎	❎	❎	❎	❎	❎
Tokens with Blocklists	❎	❎	!	❎	!	❎	❎	❎	❎
Pausable Tokens	❎	!	❎	!	❎	!	!	❎	❎
Revert on Large Approvals & Transfers	❎	❎	❎	❎	❎	❎	❎	❎	❎

Recommendation

If you plan to add a new deposit token make sure it's compatible with the current `SatlayerPool` contract.

[L-04] When `approve()` returns false the contract should revert

Description

The interface of this approve() method returns `false` in case approval failed. Not all ERC20 tokens revert on failed approval so it makes sense to check the return value and revert in case it is `false` .

Recommendation

Revert in case the approve() method returned `false` .

[L-05] The `_decimals` property should be immutable

Description

The _decimals property should be marked as `immutable` in order to save gas.

Recommendation

Mark the _decimals property as `immutable` .

[L-06] Error not used

Description

The TokenAlreadyConfiguredWithState error is not used anywhere, remove it.

Recommendation

Remove the TokenAlreadyConfiguredWithState error.

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 the author 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 the author

Prepared by ryzhak.com.

Perform a comprehensive smart contract audit using advanced formal verification tools that identify even the most elusive and intricate bugs within smart contracts and mathematically prove the absence of security issues. All formal verification properties will be integrated into your standard deployment CI/CD pipelines, which helps reduce the number of bugs in already audited code, thereby lowering costs for future security assessments.