Security Audit Report

Structured Private Deposit Smart Contract

v1.0

June 12, 2025

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Droplet Labs Ltd to perform a security audit of the Structured Private Deposit Smart Contract.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/structured-org/spdbtc
Commit	`026687d45f53078fa388fe00ecec1f9716e09858`
Scope	The scope is restricted to the contract in the `contracts/spdBTC` directory.
Fixes verified at commit	`ac53b5481cf317f6b6a3254864b370a4bf89efe6` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base’s intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The Structured Private Deposit smart contract is an ERC-4626 compliant vault that enables users to deposit WBTC (Wrapped Bitcoin) and receive spdBTC tokens at a 1:1 ratio.

The contract features a custodial model where deposited WBTC is transferred to a designated custodian address. It includes essential security measures such as deposit limits, address blacklisting capabilities, and a pause mechanism.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Low-Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium**	The client shared a high-level overview of the contract.
Test coverage	**Low-Medium**	`hardhat coverage` reports a test coverage of `61.54%`.

Summary of Findings

No	Description	Severity	Status
1	Function signature mismatch bypasses custom custodian flow	Critical	Resolved
2	Non-standard ERC-4626 implementation with external custody breaks core vault functionality	Critical	Resolved
3	Missing withdrawal functionality and burn method	Major	Partially Resolved
4	Incomplete enforcement of blacklist and pause mechanisms allows bypass	Major	Resolved
5	Ineffective deposit limit validation	Minor	Resolved
6	Missing validation of `_minDeposit` and `_maxDeposit`	Minor	Resolved
7	Missing initialization check in the `deposit` method	Minor	Acknowledged
8	Centralization issues	Minor	Acknowledged
9	Inconsistent implementation of ERC-4626 maximum limits	Minor	Resolved
10	Use of inline strings instead of custom errors is inefficient	Informational	Resolved
11	Inconsistent use of `_msgSender` and `msg.sender`	Informational	Resolved
12	Redundant use of `notBlacklisted` modifier for `deposit` method	Informational	Resolved
13	Redundant allowance check in `_deposit` method	Informational	Resolved
14	The `_maxDeposit` state variable should be private	Informational	Resolved

Detailed Findings

1. Function signature mismatch bypasses custom custodian flow

Severity: Critical

In `contracts/spdBTC/spdBTC.sol:153-168`, the `deposit` function facilitates the deposit of WBTC tokens in exchange for spdBTC tokens.

During its execution, it internally calls the `_deposit` function with four parameters: `caller`, `receiver`, `amount`, and `shares`.

However, the contract's `_deposit` function is defined to accept only three parameters: `caller`, `receiver`, and `amount`. Due to this discrepancy, Solidity's function resolution mechanism defaults to invoking the parent contract's `_deposit` function from the ERC-4626 standard, which matches the four-parameter signature.

As a consequence, the custom deposit logic defined in the child contract, specifically, the transfer of deposited assets to the `_custodian` address, is completely bypassed.

This leads to deposited WBTC tokens remaining within the contract's balance, deviating from the intended asset flow and undermining custody guarantees.

Furthermore, the `deposit` function enforces a rigid 1:1 mapping between assets and shares by invoking `_deposit(_msgSender(), receiver, amount, amount)`, passing identical values for both asset and share amounts. While this creates a 1:1 exchange rate, it deviates from the dynamic ratio utilized by the parent ERC-4626 standard, which calculates redemptions based on pool state: `shares * totalAssets() / totalSupply()`.

This discrepancy introduces inconsistency between deposit and redemption operations, risking inaccurate accounting of user balances over time as the pool state changes. Notably, this fixed exchange rate approach is not applied in the `mint` function, compounding the inconsistency.

Recommendation

We recommend aligning the function signature in the child contract's `_deposit` implementation to match the four-parameter structure used in the internal call. This will ensure that the intended custom logic, particularly the transfer of deposited assets to the `_custodian`, is correctly executed.

Status: Resolved

2. Non-standard ERC-4626 implementation with external custody breaks core vault functionality

Severity: Critical

The contract implements ERC-4626 but is designed to significantly deviate from the standard pattern by transferring deposited assets to an external custodian rather than holding them within the vault contract itself.

However, the contract fails to override ERC-4626 functions such as `totalAssets`, `withdraw`, `redeem`, and `mint` to account for this custodial model. For instance, the current implementation allows users to bypass the custodian transfer functionality by calling `mint` instead of `deposit`. As a result, any assets transferred via `mint` would be sent to the vault contract itself instead of the custodian.

These inherited functions will operate incorrectly since they assume assets are held by the vault. This breaks the ERC-4626 interface guarantees, making the vault unable to report asset balances or process withdrawals properly.

Recommendation

We recommend either following the standard ERC-4626 implementation by having the vault hold assets directly. Alternatively, consider removing the ERC-4626 standard entirely, as the current implementation only partially follows the standard.

Status: Resolved

3. Missing withdrawal functionality and burn method

Severity: Major

In `contracts/spdBTC/spdBTC.sol:199-206`, the contract is designed to allow users to deposit WBTC tokens in exchange for spdBTC tokens (at a 1:1 ratio), but there is no functionality to withdraw or redeem these tokens back for the underlying WBTC. Additionally, no burn mechanism enables future withdrawal contracts to properly burn spdBTC tokens when users want to redeem them.

The absence of withdrawal functionality effectively means that all user deposits are permanently locked. This limitation is not documented in the code or comments, which could lead users to deposit funds without understanding that they cannot be retrieved.

Furthermore, since the contract is intended to be immutable, the lack of an external burn function now will make implementing proper withdrawals via a separate contract difficult in the future.

Recommendation

We recommend implementing an external burn method that can be called by authorized addresses (such as future withdrawal contracts). This function would allow spdBTC tokens to be properly burned when users redeem them for WBTC.

Additionally, clearly document in both code comments and user-facing documentation that:

1. The current implementation does not support withdrawals
2. User funds are effectively locked until withdrawal functionality is implemented
3. A timeline or process for when/how withdrawals will be enabled

If the contract is deployed without these changes, a new version should be deployed before accepting any user deposits.

Status: Partially Resolved

4. Incomplete enforcement of blacklist and pause mechanisms allows bypass

Severity: Major

In `contracts/spdBTC/spdBTC.sol:199-206`, the contract enforces blacklist and paused state checks exclusively within the `deposit` function. Specifically, the blacklist is used to prevent targeted addresses from depositing WBTC and receiving newly minted spdBTC tokens during deposit operations.

However, this control is incomplete, as it does not extend to other ERC-4626 and ERC-20 functions, such as `withdraw`, `redeem`, `transfer`, `transferFrom`, or `mint`.

As a result, once spdBTC tokens are minted and in circulation, blacklisted users can trivially bypass the restriction. For example, a non-blacklisted user can deposit WBTC, receive spdBTC tokens, and subsequently transfer those tokens to a blacklisted address using the standard ERC-20 `transfer` function, which lacks any blacklist enforcement. Similarly, blacklisted addresses can execute the `mint` function to directly bypass the guard in the `deposit` function.

We classify this issue as major since if the blacklist is implemented for regulatory compliance purposes, then the ability to trivially bypass it represents a significant issue.

Recommendation

We recommend extending the blacklist and pause mechanisms to cover all token operations by overriding the functions in the ERC20 and ERC-4626 implementations.

Status: Resolved

5. Ineffective deposit limit validation

Severity: Minor

In `contracts/spdBTC/spdBTC.sol:210-225`, the deposit function invokes `_isValidDeposit` to validate the deposit intent against the defined maximum deposit per receiver.

However, the `maxDeposit` function does not account for the cumulative amount already deposited by a receiver. As a result, this validation becomes ineffective, as malicious actors can easily bypass it by splitting their intended deposit across multiple transactions.

Additionally, an attacker can further circumvent this check by utilizing multiple accounts, thereby evading the per-receiver limitation entirely.

Recommendation

We recommend implementing a total cumulative deposit limit, which is the only check that cannot be effectively enforced.

Status: Resolved

6. Missing validation of `_minDeposit` and `_maxDeposit`

Severity: Minor

The `initializeProduct` function within `contracts/inspdBTC/spdBTC.sol:111-123` is responsible for initializing key contract parameters.

However, it lacks a validation check to ensure logical consistency between `_maxDeposit` and `_minDeposit`. Without enforcing that `_maxDeposit` must be greater than `_minDeposit`, the function may accept and set incorrect configurations.

Recommendation

We recommend implementing a validation check within the `initializeProduct` function to enforce that `_maxDeposit` is strictly greater than `_minDeposit`.

Status: Resolved

7. Missing initialization check in the `deposit` method

Severity: Minor

The `deposit` method in `contracts/spdBTC/spdBTC.sol:153-168` does not verify whether the contract has been initialized before allowing deposits.

This could lead to unintended behavior, such as deposits being processed before the contract is properly configured (e.g. custodian address or deposit limits not set).

Recommendation

We recommend adding the already defined `whenInitialized` modifier to the deposit method to ensure that deposits can only occur after the contract has been properly initialized.

Status: Acknowledged

This issue has been acknowledged as subsequent changes made during the implementation of the fixes have rendered it no longer applicable.

8. Centralization issues

Severity: Minor

In `contracts/spdBTC/spdBTC.sol:177-206`, the contract exposes administrative functions, `setContractPaused`, `setCustodian`, and `setBlacklisted`, which are restricted to the admin via the `onlyOwner` modifier.

This design grants the administrator unilateral authority to pause all contract operations and arbitrarily blacklist or unblacklist user addresses.

Additionally, the ability to reassign immediately the custodian address further centralizes control over asset custody.

Such centralized control mechanisms introduce significant trust assumptions, as a compromised administrator account could arbitrarily freeze user assets, disrupt contract availability, or deny service to specific participants.

Recommendation

We recommend implementing governance mechanisms or multi-signature controls to mitigate the risks of unilateral administrative actions.

Additionally, we recommend implementing a timelock mechanism for custodian changes that enforces a mandatory waiting period of at least 24 hours between proposing a new custodian and executing the change.

Status: Acknowledged

9. Inconsistent implementation of ERC-4626 maximum limits

Severity: Minor

In `contracts/spdBTC/spdBTC.sol:139-143`, the contract overrides the `maxDeposit` function from the ERC-4626 standard but fails to override the related functions `maxMint`, `maxWithdraw`, and `maxRedeem`.

This inconsistency creates a mismatch in behavior, as `maxDeposit` includes the `whenInitialized` modifier and returns a custom maximum value, while the other limit functions inherit their implementation from the ERC-4626 base contract without these customizations.

Recommendation

We recommend implementing all four maximum limit functions consistently. Alternatively, consider removing the ERC-4626 standard entirely, as the current implementation only partially follows the standard.

Status: Resolved

10. Use of inline strings instead of custom errors is inefficient

Severity: Informational

In `contracts/spdBTC/spdBTC.sol`, the contract relies on string-based error messages within `require` statements.

This practice leads to higher gas consumption, as dynamic strings require additional memory allocation and increase bytecode size.

Furthermore, it reduces the clarity and maintainability of the codebase, as error strings are prone to inconsistencies and typos. Utilizing custom errors offers a more gas-efficient and structured approach to error handling, enabling standardized identification of failure conditions.

Recommendation

We recommend refactoring the `require` statements to utilize custom errors instead of inline string messages.

Status: Resolved

11. Inconsistent use of `_msgSender` and `msg.sender`

Severity: Informational

The `spdBTC` contract uses both `_msgSender` and `msg.sender` interchangeably.

This inconsistency can lead to potential issues in scenarios involving meta-transactions or proxy contracts.

While `_msgSender` currently resolves to `msg.sender` via OpenZeppelin's `Context` contract, it is designed to be overridden in more complex setups (e.g. meta-transactions or proxies).

Mixing the two can cause unexpected behavior, such as bypassing access control or blacklist checks.

Recommendation

We recommend to replace the instance of `msg.sender` in the `notBlacklisted` modifier, see `contracts/spdBTC/spdBTC.sol:77-80`, with `_msgSender` for consistency and future-proofing.

Status: Resolved

12. Redundant use of `notBlacklisted` modifier for `deposit` method

Severity: Informational

The `deposit` method currently uses the `notBlacklisted` modifier in `contracts/spdBTC/spdBTC.sol:162` to ensure the caller is not blacklisted.

However, the `_isValidDeposit` internal function, which is called within `deposit`, already performs a similar blacklist check for both the caller (`_msgSender`) and the receiver. This makes the `notBlacklisted` modifier redundant for the `deposit` method.

Recommendation

We recommend to remove the `notBlacklisted` modifier from the `deposit` method to avoid redundancy and improve gas efficiency.

Status: Resolved

13. Redundant allowance check in `_deposit` method

Severity: Informational

The internal `_deposit` method includes an explicit allowance check in `contracts/spdBTC/spdBTC.sol:239-240`.

This check is redundant because the subsequent `safeTransferFrom` call will invoke the underlying `transferFrom` function of the ERC-20 token, which will revert if the allowance is insufficient. Therefore, the explicit allowance check is unnecessary.

Recommendation

We recommend removing the explicit allowance check and relying on the `safeTransferFrom` function to handle insufficient allowance errors via the underlying `transferFrom` call.

Status: Resolved

14. The `_maxDeposit` state variable should be private

Severity: Informational

In `contracts/spdBTC/spdBTC.sol:23`, the `_maxDeposit` state variable is declared as public, but there is already an explicit getter function `maxDeposit` that returns this value.

This creates duplicate access methods for the same state variable, which is redundant and potentially confusing for developers. Also, more gas is spent for public variables than private.

Recommendation

We recommend changing the visibility of `_maxDeposit` from public to private.

Status: Resolved