SatLayer Phase 1

Smart Contract Security Assessment

March 25, 2025

ABSTRACT

Dedaub was commissioned to perform a security audit of **SatLayer**, a restaking platform built on the Babylon blockchain—a Cosmos SDK-based chain secured by Bitcoin staking. SatLayer enables Bitcoin restakers to secure decentralized applications or protocols by providing Bitcoin Validated Services (BVS). Through this mechanism, Bitcoin serves as the primary security collateral, extending its use beyond traditional cryptocurrency applications.

BACKGROUND

This audit marks the first phase of the security review. The scope focused exclusively on contracts related to the platform's Total Value Locked (TVL), specifically the vault system. In this system, operators can own vaults that, in the future, may be subject to slashing or rewards based on their performance. At the time of the audit, the available functionality was limited to depositing, withdrawing, pausing, and managing a whitelist for the vaults.

SETTING & CAVEATS

This audit report mainly covers the contracts of the **at-the-time private** repository https://github.com/satlayer/satlayer-bvs of the Protocol **SatLayer** at commit `03650d141f8b2633b2573b7959df042d409ab22a`.

```
crates/
├── bvs-library/
│   └── src/
│       ├── addr.rs
│       ├── lib.rs
│       └── ownership.rs
├── bvs-pauser/
│   └── src/
│       ├── bin/
│       │   └── schema.rs
│       ├── contract.rs
│       ├── error.rs
│       ├── lib.rs
│       ├── msg.rs
│       └── state.rs
├── bvs-vault-bank/
│   └── src/
│       ├── bank.rs
│       ├── bin/
│       │   └── schema.rs
│       ├── contract.rs
│       ├── error.rs
│       ├── lib.rs
│       └── msg.rs
├── bvs-vault-base/
│   └── src/
│       ├── error.rs
│       ├── lib.rs
│       ├── msg.rs
│       ├── offset.rs
│       ├── router.rs
│       └── shares.rs
├── bvs-vault-cw20/
│   └── src/
│       ├── bin/
│       │   └── schema.rs
│       ├── contract.rs
│       ├── error.rs
│       ├── lib.rs
│       ├── msg.rs
│       └── token.rs
└── bvs-vault-router/
    └── src/
        ├── bin/
        │   └── schema.rs
        ├── contract.rs
        ├── error.rs
        ├── lib.rs
        ├── msg.rs
        └── state.rs
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

[No low severity issues]

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	All contracts are upgradeable	**OPEN**

All the contracts to be deployed in the system are upgradeable by a signer from SatLayer. Therefore future iterations of the contracts are not guaranteed to contain the same functionality they will contain at launch

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	`transfer_ownership` in `bvs-library::ownership` performs a redundant load of a storage variable	**RESOLVED**

```
/// Asserts that the sender of the message is the owner of the contract
pub fn assert_owner(storage: &dyn Storage, info: &MessageInfo) ->
Result<(), OwnershipError> {
    let owner = OWNER.load(storage)?;
    if info.sender != owner {
        return Err(OwnershipError::Unauthorized);
    }
    Ok(())
}

pub fn transfer_ownership(
    storage: &mut dyn Storage,
    info: MessageInfo,
    new_owner: Addr,
) -> Result<Response, OwnershipError> {
    assert_owner(storage, &info)?;

    // @dedaub - OWNER is loaded once in assert_owner then loaded again
here
    let old_owner = OWNER.load(storage)?;
    OWNER.save(storage, &new_owner)?;
    Ok(Response::new().add_event(
        Event::new("TransferredOwnership")
            .add_attribute("old_owner", old_owner.as_str())
            .add_attribute("new_owner", new_owner.as_str()),
    ))
}
```

As can be seen in the above code snippet the owner is loaded once inside the `assert_owner` function and then is later loaded again right before the response is sent out so that it can be included in the event. This can be optimised by removing the assert_owner function and reimplementing it such that a load only occurs once.

A2	Wrong comment in `bvs-vault-base::shares` for both `add_shares` and `sub_shares`	**RESOLVED**

```
// @dedaub - why does it say unchecked add here?
/// Unchecked add, you can add zero shares--accounting module won't check
this.
/// Adding zero shares is as good as not running this function.
pub fn add_shares(
    storage: &mut dyn Storage,
    recipient: &Addr,
    new_shares: Uint128,
) -> Result<Uint128, StdError> {
    SHARES.update(storage, recipient, |recipient_shares| -> StdResult<_> {
        recipient_shares
            .unwrap_or(Uint128::zero())
            .checked_add(new_shares)
            .map_err(StdError::from)
    })
}

// @dedaub - same here
/// Unchecked sub, you can remove zero shares--accounting module won't
check this.
/// Removing zero shares is as good as not running this function.
pub fn sub_shares(
    storage: &mut dyn Storage,
    recipient: &Addr,
    shares: Uint128,
) -> Result<Uint128, StdError> {
    SHARES.update(storage, recipient, |recipient_shares| -> StdResult<_> {
        recipient_shares
            .unwrap_or(Uint128::zero())
            .checked_sub(shares)
            .map_err(StdError::from)
    })
}
```

As is evidenced by the @dedaub comments the comments say that there is an unchecked sub/add however the code clearly shows that it's actually checked.

A3	Redundant check in `withdraw_to` for both the bank and cw20 vaults.	**RESOLVED**

```
pub fn withdraw_to(
        deps: DepsMut,
        env: Env,
        info: MessageInfo,
        msg: RecipientAmount,
    ) -> Result<Response, ContractError> {
        router::assert_not_validating(&deps.as_ref())?;

        // Remove shares from the info.sender
        shares::sub_shares(deps.storage, &info.sender, msg.amount)?;

        let (vault, claim_assets) = {
            let balance = token::query_balance(&deps.as_ref(), &env)?;
            let mut vault = offset::VirtualOffset::load(&deps.as_ref(),
balance)?;
```

```
            // @dedaub - not bad to have this here but realistically it
should never be reached.
            // and even if reached - vault.checked_sub_shares would
revert.
            if msg.amount > vault.total_shares() {
                return Err(VaultError::insufficient("Insufficient shares
to withdraw.").into());
            }

            let assets = vault.shares_to_assets(msg.amount)?;
            if assets.is_zero() {
                return Err(VaultError::zero("Withdraw assets cannot be
zero.").into());
            }

            // Remove shares from TOTAL_SHARES
            vault.checked_sub_shares(deps.storage, msg.amount)?;

            (vault, assets)
        };
}
```

If `shares::sub_shares` succeeds but then `msg.amount > vault.total_shares` fails then there are much more serious problems in the protocol as the sum of all shares in the `shares` mapping should always be equal to `vault.total_shares`, however even in this case if the statement is reached the code would fail regardless at the `vault.checked_sub_shares` statement.

A4	Considerations for Virtual Shares and Offset System	RESOLVED

The vault implementation employs a front-running mitigation strategy inspired by OpenZeppelin's ERC4626. This introduces two key considerations:

First, the provided implementation lacks an offset scaling factor, resulting in an approximately 1:1 pairing of shares and assets. We recommend including this scaling factor to improve the precision of deposit and withdrawal calculations and to reduce rounding errors. This will be especially important once slashing is introduced.

Second, the virtual shares system presents challenges when combined with slashing. As OpenZeppelin notes:

> The drawback of this approach is that the virtual shares do capture (a very small) part of the value being accrued to the vault. Also, if the vault experiences losses, the users try to exit the vault, the virtual shares and assets will cause the first user to exit to experience reduced losses in detriment to the last users that will experience bigger losses.

In the event of slashing, this creates an incentive for delegators to withdraw early to minimize their own losses. We believe this is an undesirable behavior for the system.

DISCLAIMER

ABOUT DEDAUB