SatLayer

BVS Slashing

Smart Contract Security Assessment

June 20, 2025

ABSTRACT

Dedaub was commissioned to perform a security audit of parts of SatLayer's newly introduced slashing scheme.

BACKGROUND

As a restaking protocol, SatLayer aims to enforce financial incentives that discourage protocol operators from performing malicious actions against the validated services they support. This will be achieved by introducing functionality that allows Bitcoin Validated Services (BVS) opting into SatLayer's staking primitives to slash a portion of an operator's stake in cases of malicious behavior or malfunction.

SatLayer is built on the Babylon blockchain—a Cosmos SDK-based chain secured by Bitcoin staking. Babylon supports smart contract execution through the CosmWasm module of Cosmos, which enables the development of smart contracts in the Rust programming language.

SETTING & CAVEATS

This audit report mainly covers the contracts of the repository **https://github.com/satlayer/satlayer-bvs** at commit `260e45bb379e0a2cb6be1524de5b9a2be303d6e9`.

Considering this was not a full audit of the protocol's slashing functionality, the team also considered all the relevant diffs from the first audit of the protocol's core at commit `03650d141f8b2633b2573b7959df042d409ab22a`.

2 auditors worked on the following contracts for 7 days:

```
crates
├── bvs-vault-router
│   ├── Cargo.toml
│   ├── package.json
│   ├── README.md
│   └──src
│       ├── contract.rs
│       ├── error.rs
│       ├── lib.rs
│       ├── msg.rs
│       └── state.rs
│
├── bvs-vault-cw20-tokenized
│   ├── Cargo.toml
│   ├── package.json
│   ├── README.md
│   └──src
│       ├── contract.rs
│       ├── error.rs
│       ├── lib.rs
│       └── msg.rs
│
└── bvs-vault-bank-tokenized
    ├── Cargo.toml
    ├── package.json
    ├── README.md
    └── src
        ├── bin
        │   └── schema.rs
        ├── contract.rs
        ├── error.rs
        ├── lib.rs
        └── msg.rs
```

Although this scope does not cover the full implementation of SatLayer's slashing scheme, the auditors spent considerable time reviewing the remaining slashing contracts to build a solid understanding of the code's security context.

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.

Finally, the following components:

- The Cosmos SDK
- The CosmWasm runtime
- The CosmWasm SDK

are all considered secure for the purposes of this security audit. While the auditors reviewed the use of all involved libraries and considered potential attack vectors stemming from the Cosmos/CosmWasm runtime, a thorough audit of the above-mentioned dependencies was outside the scope.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	Queued withdrawals may still count towards an operator's voting power	**ACKNOWLEDGED**
Acknowledgement:		

If a BVS wishes to track power by the number of shares, they should add code to account for queued shares and not rely on the operator's vault balance. In terms of the protocol's accounting, an operator's voting power will not be tracked on-chain directly but an oracle SDK will be provided in the future.

From the protocol's whitepaper, we have that the intended flow is:

1. A vault is created for an operator.
2. The operator deposits funds in it
3. Other stakers deposit funds in it

The shares that get minted by the vault should be representative of the funds that have been delegated to the operator and thus determine their voting power.

With the introduction of slashing, operators should not be permitted to withdraw their stake atomically – since this would allow them to perform a slashable action and immediately withdraw funds within the same transaction before there's any time to submit a slashing request.

However, in the case of queued withdrawals the shares of the user are not burned immediately, but are rather transferred from the user to the vault itself.

bvs-vault-bank-tokenized::queue_withdrawal_to:322

```
...
// ill-liquidate the receipt token from the staker
// by moving the asset into this vault balance.
// We can't burn until the actual unstaking (redeem withdrawal) occurs.
// due to total supply mutation can impact the exchange rate to change
prematurely.

cw20_base::contract::execute_transfer(
    deps.branch(),
```

```
    env.clone(),
    info.clone(),
    env.contract.address.to_string(), //@Dedaub: token sender is info.sender
    msg.amount,
)?;
...
```

This means that for the outside world the vault maintains the same number of minted shares. A problem that could arise from this is that a malicious operator queues the withdrawal of their stake and while maintaining the same voting power within a BVS. Once the stake becomes withdrawable they would be free to perform any slashable action and atomically withdraw their stake.

P2	A BVS's `resolution_window` slashing parameter should reflect the BVS's tolerance in detecting slashable actions	**ACKNOWLEDGED**

Once a slash request is submitted, the BVS can configure a resolution window period that will give some time to the accused operator to resolve the slashing request.

`bvs-vault-router::request_slashing:370`

```
...
let request_resolution = env
    .block
    .time
    .plus_seconds(slashing_parameters.resolution_window);

// request_expiry will be using `resolution_window`
// value from the timestamp's slashing_parameters,
// instead of the most recent slashing param.
// This ensures that both parties agree upon all parameters used.
let request_expiry = env
    .block
    .time
    .plus_seconds(slashing_parameters.resolution_window)
    .plus_seconds(SLASHING_REQUEST_EXPIRY_WINDOW.u64());
```

```
let new_request = SlashingRequest {
    request: data.clone(),
    request_time: env.block.time,
    request_resolution, //@Dedaub: slashing cannot take place until resolution
window passes
    request_expiry,
    status: SlashingRequestStatus::Pending.into(),
    service: service.clone(),
};
...
```

However, for the slashing schema to be considered precise the timestamp of the slashable action should be as close as possible to when the action actually took place and be agreeable by the parties responsible for finalizing the slashing vote.

bvs-vault-router::request_slashing:264

```
...
// require active status between operator and service
let StatusResponse(operator_service_status) = deps.querier.query_wasm_smart(
    registry.to_string(),
    &bvs_registry::msg::QueryMsg::Status {
        service: service.to_string(),
        operator: operator.to_string(),
        timestamp: Some(data.timestamp.seconds()), //@Dedaub: Slashable action
timestamp
    },
)?;
...

// get slashing params of the service at the given timestamp, also checks if
slashing is enabled
let SlashingParametersResponse(slashing_parameters) =
deps.querier.query_wasm_smart(
    registry.clone(),
    &bvs_registry::msg::QueryMsg::SlashingParameters {
```

```
        service: service.to_string(),
        timestamp: Some(data.timestamp.seconds()),
    },
)?;
...
```

The BVS should choose an appropriate resolution window period such that even if the maximum delay in finding about the slashable action takes place, the operator cannot initiate a withdrawal at the time the slashable action takes place and still manage to withdraw their funds in time.

P3	Malicious operators can orchestrate really small partial slashing proposals to protect their stake	**ACKNOWLEDGED**

Acknowledgement:

This reflects a consideration of how slashing proposals are initiated in a BVS's code. Although this is not actionable at the protocol level, the documentation will be updated to warn about the permissionless initiation of slashing.

Because the percentage to be slashed can be any non-zero percentage as long as the maximum is not hit:

bvs-vault-router::request_slashing:270

```
// ensure that bips is greater than zero
if data.bips < 1 {
    return Err(InvalidSlashingRequest {
        msg: "Slashing bips must be greater than zero".to_string(),
    });
}
...
// ensure bips must not exceed max_slashing_bips set by service
if data.bips > slashing_parameters.max_slashing_bips {
    return Err(InvalidSlashingRequest {
```

```
        msg: "Slashing bips exceeds the maximum allowed by service".to_string(),
    });
}
```

And because an active proposal for a given operator and service pair will block new proposals from being created:

`bvs-vault-router::request_slashing:333`

```
...
match SlashingRequestStatus::try_from(prev_slashing_request.status)? {
    SlashingRequestStatus::Pending => {
        // slashing is pending within the expiry date
        if prev_slashing_request.request_expiry > env.block.time {
            return Err(ContractError::InvalidSlashingRequest {
                msg: "Cannot process new request while previous slashing request
is pending".to_string(),
            });
        }
...
```

Although this is not a direct issue on the router contract itself, it implicitly places a restriction on how slashing proposals can be initiated **through the smart contract of a service**. A fully permissionless and unconditional proposal initiation scheme could create a situation where:

1. An operator performs a slashable action
2. The operator (or an affiliated malicious address) repeatedly initiates partial slash requests for `bip = 1` (that's 0.01%) that can even get accepted or left up to expiration
3. In the scenario of an expiration, another 1-bip slash proposal is requested

This creates a situation where no proper slashing proposal may be submitted.

P4	Operators should be forced to opt into slashing after the `bvs-registry` contract is upgraded	**ACKNOWLEDGED**

With the introduction of the slashing scheme, an operator accepting the slashing parameters of a service is recorded in the `SLASHING_OPT_IN` map of the `bvs-registry` contract, which is read by `bvs-registry::is_operator_opted_in_to_slashing`:

`bvs-vault-router::request_slashing:333`

```
...
pub fn is_operator_opted_in_to_slashing(
    deps: Deps,
    service: Addr,
    operator: Addr,
    timestamp: Option<u64>,
) -> StdResult<IsOperatorOptedInToSlashingResponse> {
    let is_opted_in =
        state::is_operator_opted_in_to_slashing(deps.storage, &service,
&operator, timestamp)?;
    Ok(IsOperatorOptedInToSlashingResponse(is_opted_in))
}
...
```

After the upgrade, this map will not be populated on its own. There should be a clear mechanism (even off-chain) that will force the operators to submit the acceptance of the slashing parameters for the existing services.

Otherwise services might continue being validated by operators for which no slashing request might be initiated:

`bvs-vault-router::request_slashing:317`

```
...
if !is_operator_opted_in {
    return Err(InvalidSlashingRequest {
```

```
        msg: "Operator must be opted-in to slashing at the specified
timestamp".to_string(),
    });
}
...
```

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:

ID	Description	STATUS
H1	Queued withdrawals can be bypassed by malicious operators	**RESOLVED**

With the introduction of slashing, operators and stakers should not be able to atomically withdraw from vaults since vaults for operators that are actively validating a service must be slashable in the scenario a malicious operation takes place.

The above-mentioned property is enforced by the following code snippets:

bvs-vault-tokenized::withdraw_to:270

```
/// Withdraw assets from the vault by burning receipt token.
/// The resulting staked assets are now unstaked and transferred to
```

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

bvs-vault-base::assert_not_validating:77

```
pub fn assert_not_validating(deps: &Deps) -> Result<(), VaultError> {
    let router = get_router(deps.storage)?;
    let operator = get_operator(deps.storage)?;

    let is_delegated: bool = deps.querier.query_wasm_smart(
        router.to_string(),
        &QueryMsg::IsValidating {
            operator: operator.to_string(),
        },
    )?;
    if is_delegated {
        return Err(VaultError::Validating {});
    }
    Ok(())
}
```

bvs-vault-router::is_validating:862

```
/// Returns whether the operator is delegated or not.
/// Called by vaults to check if they are delegated.
/// Delegated vaults must queue withdrawals.
pub fn is_validating(deps: Deps, operator: Addr) -> StdResult<bool> {
    let registry = state::get_registry(deps.storage)?;
    let is_operator_active: bool = deps.querier.query_wasm_smart(
        registry.to_string(),
        &QueryMsg::IsOperatorActive(operator.to_string()),
    )?;
```

```
    Ok(is_operator_active)
}
```

bvs-registry::state::is_operator_active:178

```
/// Check if the operator is actively registered to any service
pub fn is_operator_active(store: &dyn Storage, operator: &Operator) ->
StdResult<bool> {
    let active_count = OPERATOR_ACTIVE_REGISTRATION_COUNT
        .may_load(store, operator)?
        .unwrap_or(0);

    Ok(active_count > 0)
}
```

Whether an operator is considered active depends on whether the number of BVS' to which they have registered is non-zero. However, an operator has the ability to permissionlessly de-register from a validated service by executing `bsv-registry::deregister_service_from_operator`:

bvs-registry::deregister_service_from_operator:364

```
/// Deregister a service from an operator (info.sender is the Operator)
/// Set the registration status to [`RegistrationStatus::Inactive`] (0)
pub fn deregister_service_from_operator(
    deps: DepsMut,
    info: MessageInfo,
    env: Env,
    service: Addr,
) -> Result<Response, ContractError> {
    let operator = info.sender.clone();

    let key = (&operator, &service);
    let status = get_registration_status(deps.storage, key)?;

    if status == RegistrationStatus::Inactive {
        Err(ContractError::InvalidRegistrationStatus {
            msg: "Service is not registered with this operator".to_string(),
```

```
        })
    } else {
        set_registration_status(deps.storage, &env, key,
RegistrationStatus::Inactive)?;
...
```

bvs-registry::state::set_registration_status:150

```
pub fn set_registration_status(
    store: &mut dyn Storage,
    env: &Env,
    key: (&Operator, &Service),
    status: RegistrationStatus,
) -> StdResult<()> {
    let (operator, service) = key;
    match status {
        RegistrationStatus::Active => {
            increase_operator_active_registration_count(store, operator)?;
            // if service has enabled slashing, opt-in operator to slashing
            if is_slashing_enabled(store, service,
Some(env.block.time.seconds()))? {
                opt_in_to_slashing(store, env, service, operator)?;
            }
        }
        RegistrationStatus::Inactive => {
            decrease_operator_active_registration_count(store, operator)?;
        }
        _ => {}
    }
...
```

bvs-registry::state::decrease_operator_active_registration_count:203

```
/// Decrease the operator active registration count by 1
pub fn decrease_operator_active_registration_count(
    store: &mut dyn Storage,
    operator: &Operator,
) -> StdResult<u64> {
    OPERATOR_ACTIVE_REGISTRATION_COUNT.update(store, operator, |count| {
        let new_count = count.unwrap_or(0).checked_sub(1);
```

```
        new_count.ok_or_else(|| {
            StdError::generic_err("Decrease operator active registration count
failed")
        })
    })
}
```

The attack scenario that arises is therefore the following:

1. An operator performs any slashable action in any services they are participating
2. The operator de-registers from all services by invoking `bsv-registry::deregister_service_from_operator` multiple times (one for each registered BVS), setting their `OPERATOR_ACTIVE_REGISTRATION_COUNT` to 0.
3. The operator is considered to no longer be validating, so they will be able to atomically withdraw their stake before any slashing requests could take place

H2	Queued withdrawals can be DOS'ed	**RESOLVED**

Resolution:

Users can delegate the initiation and completeness of withdrawals to trusted "proxy" addresses. Because proxy addresses can initiate and complete queued withdrawals, they have to be fully trusted by the user.

In the scenario where multiple addresses are trusted as proxies, the user has to trust that they will not only behave in the user's best interest, but that they will also reach a consensus on the status of withdrawals. For instance, if one proxy address does not agree with a withdrawal, they can DOS a queued withdrawal by using the mechanism described in this issue.

When a withdrawal request is queued, a maturity date is calculated after which the recipient of the withdrawn funds will be able to successfully perform a withdrawal

`bvs-vault-bank-tokenized::queue_withdrawal_to:335`

```
...
let current_timestamp = env.block.time;
let unlock_timestamp = current_timestamp.plus_seconds(withdrawal_lock_period);
...
let new_queued_withdrawal_info = QueuedWithdrawalInfo {
    queued_shares: msg.amount,
    unlock_timestamp,
};

let result = shares::update_queued_withdrawal_info(
    deps.storage,
    &msg.recipient,
    new_queued_withdrawal_info,
)?;
...
```

`bvs-vault-bank-tokenized::redeem_withdrawal_to:375`

```
...
if unlock_timestamp.seconds() > env.block.time.seconds() {
    return Err(VaultError::locked("The shares are locked").into());
}
...
```

An issue arises if we assume that somebody (address `X`) is about to withdraw a queued withdrawal that has reached maturity.

Just before their message is executed, a griefer frontruns the withdrawal with a `queue_withdrawal_to` call, setting the receiver to `X`. This will increase `X`'s queued shares, but more importantly it will re-set the unlock timestamp timestamp for X to the latest one + withdrawal delay, causing the `unlock_timestamp` check in `redeem_withdrawal_to` to fail.

The DOS can be performed repeatedly even with small (non-zero) share amounts.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

[No low severity issues]

CENTRALIZATION CONSIDERATIONS:

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 considerations 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	Some entities are considered trusted	**INFO**

1. The owner of the SatLayer contracts can set important protocol parameters (e.g. the withdrawal lock period) and can whitelist vaults to be considered operable by operators.
2. The CosmWasm admin of the contracts may upgrade the implementation of the core contracts

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	The default decimal offset of OZ's ERC4626 implementation still leaves a window for DOSing deposits	**INFO**

The vault contracts follow the advisory of OZ's ERC4626 implementation and include virtual shares in their accounting so as to prevent the known frontrunning issue when a vault gets initialized: https://docs.openzeppelin.com/contracts/4.x/erc4626

bvs-vault-bank-tokenized::redeem_withdrawal_to:3383

```
...
let underlying_token_balance = UnderlyingToken::query_balance(&deps.as_ref(),
&env)?;
let receipt_token_supply =
    cw20_base::contract::query_token_info(deps.as_ref())?.total_supply;
let vault = offset::VirtualOffset::new(receipt_token_supply,
underlying_token_balance)?;
...
```

bvs-vault-base::VirtualOffset::new:43

```
...
pub fn new(total_shares: Uint128, total_assets: Uint128) -> StdResult<Self> {
    let virtual_total_shares =
total_shares.checked_add(OFFSET).map_err(StdError::from)?; //@Dedaub: OFFSET =
10^0 = 1
    let virtual_total_assets =
total_assets.checked_add(OFFSET).map_err(StdError::from)?;
...
```

Implementing a decimal offset of 0 (`OFFSET = 1 = 10^0`) still leaves room for the classic ERC4626 inflation attack to take place.

The difference is that the attack is not profitable for the attacker since an amount analogous to the deposit that gets attacked will be donated to the virtual shares (attacker's loss).

However, the attacker can still consider performing the attack a grief to another user's deposit. Even though the attacker won't profit from the donation, the other party will receive 0 shares.

With a decimal offset of > 0 is set, it is shown that the attacker can potentially suffer losses (i.e., donations to the virtual shares) that is `10^(decimal offset)` larger than the attacked user's deposit.

A2	`x/Bank` native transfer may fail	**INFO**

Because Babylon implements a sender restriction hook in its network code:

`@github/babylon::keepers::keepers.go:861::VirtualOffset::new:43`

```
...
// bankSendRestrictionOnlyBondDenomToDistribution restricts that only the default
bond denom should be allowed to send to distribution mod acc.
func bankSendRestrictionOnlyBondDenomToDistribution(ctx context.Context,
fromAddr, toAddr sdk.AccAddress, amt sdk.Coins) (newToAddr sdk.AccAddress, err
error) {
    if toAddr.Equals(appparams.AccDistribution) {
        denoms := amt.Denoms()
        switch len(denoms) {
        case 0:
            return toAddr, nil
        case 1:
            denom := denoms[0]
```

```
            if !strings.EqualFold(denom, appparams.DefaultBondDenom) {
                return nil, errBankRestrictionDistribution
            }
        default: // more than one length
            return nil, errBankRestrictionDistribution
        }
    }

    return toAddr, nil
}
...
```

The following `x/Bank` transfer may fail if a BVS sets up the appropriate `destination` slashing parameter with an unexpected denomination.

`bvs-vault-router::finalize_slashing:683`

```
...
if slashing_parameters.destination.is_some() {
...
            let exec_msg = BankMsg::Send {
                to_address: destination,
                amount: vec![Coin {
                    denom: vault_info.asset_reference,
                    amount: locked_amount,
                }],
            };
...
```

However, this does not seem to offer any weaponization opportunities to a malicious party so we list it as an informatory item.

A3	Vaults cannot be de-whitelisted	**INFO**

The `bvs-vault-router` contract operates all slashing-related operations on whitelisted vaults. However there's no functionality to de-whitelist an initially whitelisted vault.

Both `bvs-vault-router::finalize_slashing` and `bvs-vault-router::lock_slashing` may potentially query the state of bad vaults if a vault is initially fine but then somehow becomes "broken" (assuming the protocol wishes to guard against such a scenario).

A4	Unreachable code path	**INFO**

The following snippet in `bsv-vault-router::request_slashing` is unreachable

`bvs-vault-router::request_slashing:364`

```
...
SlashingRequestStatus::Finalized => {
    // Previous slash has been finalized, eligible for new request
}
...
```

When a slashing request is canceled or finalized, the `bsv-vault-router::remove_slashing_request_id` will clear the `SLASHING_REQUEST_IDS` which means that upon attempting to create a new slashing request for the same operator/service pair as before will have `bsv-vault-router::state::get_pending_slashing_request` return `Ok(None)` and in the following match expression will not be hit:

`bvs-vault-router::request_slashing:329`

```
...
if let Some(prev_slashing_request) = prev_slashing_request {
    match SlashingRequestStatus::try_from(prev_slashing_request.status)? {
...
```

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.