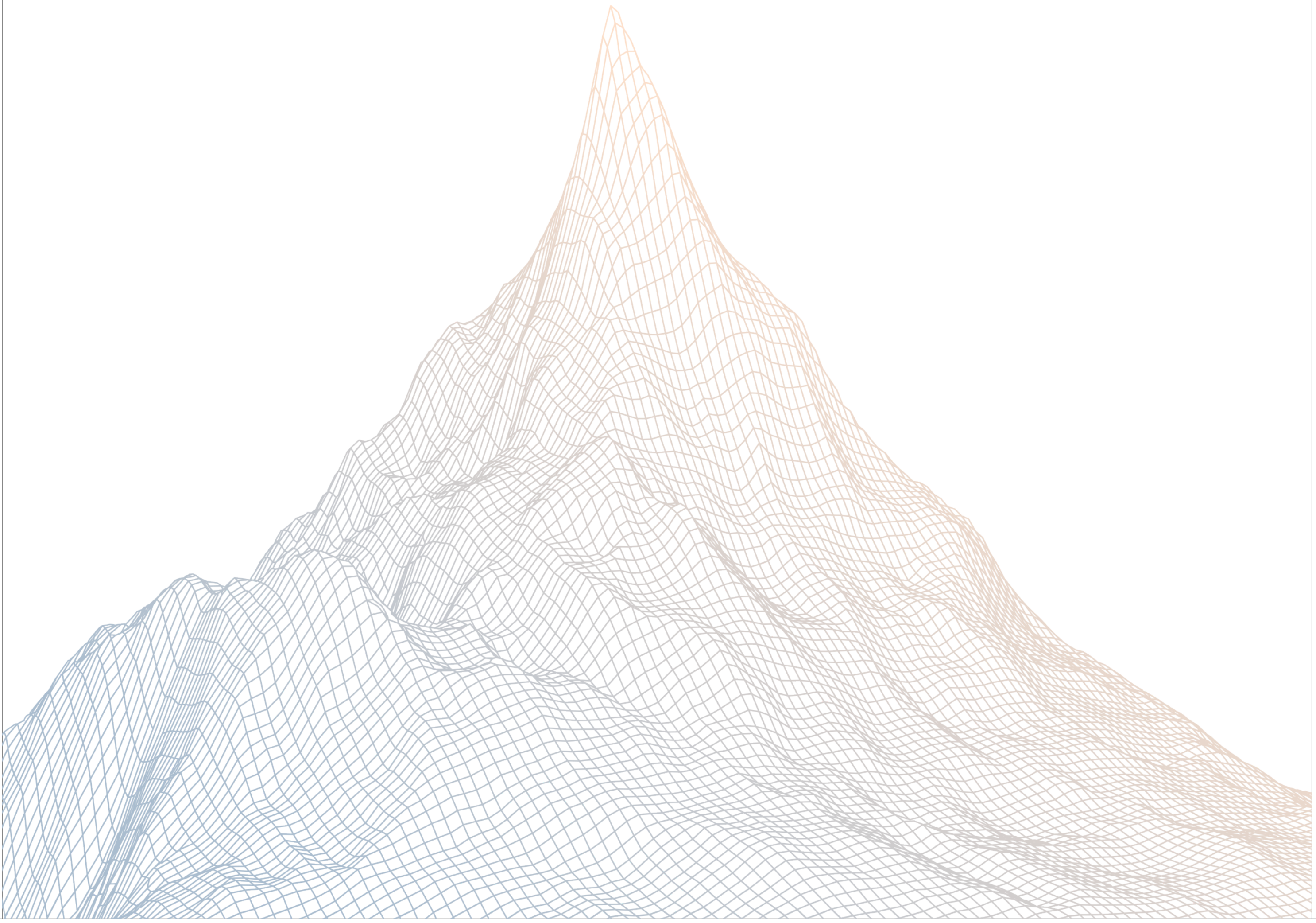
Zenith

Solv

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:	December 29th to December 31st, 2025
AUDITED BY:	Christian Vari oakcobalt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Solv

Solv Protocol is building the $1 trillion Bitcoin economy through a full-stack suite of financial services optimized for BTC holders. By enabling lending, liquid staking, yield generation, and fund management, Solv activates Bitcoin as a capital-efficient asset within a growing Bitcoin-native DeFi ecosystem. Supported by institutional partners and an on-chain BTC reserve, Solv bridges TradFi, CeFi, and DeFi, unlocking yield opportunities and expanding Bitcoin's role in programmable finance.

2.2 Scope

The engagement involved a review of the following targets:

Target	SolvBTC-Stellar-Contract
Repository	https://github.com/solv-finance/SolvBTC-Stellar-Contract
Commit Hash	`65769082938c1c0f9296b3c39359c4216976ac73`
Files	`Diff from 303dcb9f2f01b2e4dc78541a3b2bba36f4e0f693`

Target	SolvBTC-Stellar-Contract Mitigation Review
Repository	https://github.com/solv-finance/SolvBTC-Stellar-Contract
Commit Hash	`6b3ad21ccf1b2587f57bfded74022fb5bbdbfdbd`
Files	`Diff from 303dcb9f2f01b2e4dc78541a3b2bba36f4e0f693`

2.3 Audit Timeline

December 29, 2025	Audit start
December 31, 2025	Audit end
January 9, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	0
Low Risk	4
Informational	3
Total Issues	**8**

3 Findings Summary

ID	Description	Status
H-1	User withdrawals can be locked in the vault	Resolved
L-1	Signer cap still allows unlimited Minting in key-compromised scenarios	Resolved
L-2	i128_to_ascii_bytes has an overflow edge case	Acknowledged
L-3	Division before multiplication can cause fund loss with vulnerable configuration	Resolved
L-4	Missing recovery Id normalization	Resolved
I-1	Bridge missing direct burn path for EOA users	Acknowledged
I-2	Unused admin storage key in Bridge	Resolved
I-3	Misleading error handling on withdraw	Resolved

4 Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] User withdrawals can be locked in the vault

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- vault.rs#L813-L815

Description:

The vault doesn't check the length of `request_hash` in `withdraw_request`. However, in `withdraw`, it panics when `request_hash` exceeds 32 bytes. Since a user's share tokens would have already been burned after a withdrawal request, withdrawal tx failure causes permanent loss of funds.

`withdraw_request_internal()` accepts `request_hash` of any length without validation, and burns tokens without length check:

```
fn withdraw_request_internal(
    env: &Env,
    from: &Address,
    shares: i128,
    request_hash: &Bytes, //@audit no length validation
    use_burn_from: bool,
) {
...
// Burn user's shares using the appropriate method
if use_burn_from {
    token_client.burn_from(&env.current_contract_address(), from, &shares);
} else {
    token_client.burn(from, &shares);
}
```

When `withdraw()` is called, it attempts to convert `request_hash` to hex string, but panics if the length exceeds 32 bytes:

```
fn bytes_to_hex_string_bytes(env: &Env, data: &Bytes) → Bytes {
    let len = data.len() as usize;
    let mut buf = [0u8; 32];
    if len > buf.len() {
        panic_with_error!(env, VaultError::InvalidVerifierKey);
    }
...
```

This can affect both new withdrawal requests and existing requests before vault upgrades. And there is no recovery process for burned tokens from stuck withdrawal requests.

Recommendations:

1. Add the same request_hash length validation in `withdraw_request_internal()`.
2. Or for backward compatibility, consider allowing `bytes_to_hex_string_bytes()` to process arbitrary-length data.

Solv: Resolved with `@0f2d90b434 ...`

Zenith: Verified.

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] Signer cap still allows unlimited Minting in key-compromised scenarios

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- bridge.rs#L181

Description:

Current design assumption is that a per-mint cap would significantly limits the impact of a key compromise scenario.

> Per-Signer Minting Caps: We implemented granular access control using SignerCap. Each authorized signing key is assigned a strict per-transaction minting limit (e.g., Key A can sign up to 1 BTC, Key B up to 10 BTC). This design significantly limits the blast radius in the event of a key compromise—an attacker cannot drain the system or mint unlimited tokens, but is constrained by the predefined cap of that specific key.

```
let cap_key = BridgeDataKey::SignerCap(recovered_key);
let cap: i128 = env.storage().instance().get(&cap_key).unwrap_or(0);

if mint_amount > cap {
    panic_with_error!(&env, BridgeError::SignerCapExceeded);
}
```

However, this security assumption doesn't hold. When a signer key is compromised, the attacker can generate `btc_tx_hash` with valid formats and sign messages immediately without waiting for Bitcoin confirmations. Repeated `mint()` calls with fake hashes, allowing unlimited minting until the admin manually intervenes.

Recommendations:

Consider adding maximum call count per time window (e.g., max N `mint()` calls per hour per signer). Implementation would track (last_reset_time, call_count) per signer and reset the count when the time window expires.

Solv: Resolved with `@e039c6bd7f ...`

Zenith: Verified.

[L-2] `i128_to_ascii_bytes` has an overflow edge case

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- bridge.rs#L636
- vault.rs#L878

Description:

The `i128_to_ascii_bytes()` method contains an integer overflow vulnerability when attempting to negate `i128::MIN`.

```
fn i128_to_ascii_bytes(env: &Env, mut n: i128) → Bytes {
    if n == 0 {
        return Bytes::from_slice(env, b"0");
    }
    let mut buf = [0u8; 40];
    let mut i = 40;
    let is_neg = n < 0;

    if is_neg {
        n = -n;  //@audit negating i128::MIN will overflow
    }
...
```

This is not currently exploitable because input validations will reject negative values first.

Recommendations:

If this method is expected to handle negative values in the future, consider adding a special case handling for `i128::MIN`.

Solv: Acknowledged. Our business logic never uses negative values, so i128_to_ascii_bytes will only see positive inputs. As a result, the i128::MIN overflow case is unreachable in practice.

[L-3] Division before multiplication can cause fund loss with vulnerable configuration

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- bridge.rs#L525-L526
- vault.rs#L1073-L1075

Description:

The `calculate_mint_amount` function in both the bridge and vault contracts performs integer division before multiplication when `currency_decimals > shares_decimals`.

```
} else {
    let scale = 10_i128.pow(currency_decimals - common_factor);
    deposit_amount
        .checked_div(scale)
        .unwrap_or_else(|| panic_with_error!(env,
    BridgeError::InvalidAmount))
};

let minted = scaled_amount
    .checked_mul(nav_scale)
    .and_then(|x| x.checked_div(nav))
    .unwrap_or_else(|| panic_with_error!(env, BridgeError::InvalidAmount));
```

The impact depends on `currency_decimals` and `shares_decimals`. For common configurations such as currency_decimals = 8, shares_decimals = 6, the fund loss is trivial (the value of the last two digits). If shares_decimals can be set to lower decimals, the fund loss can be significant.

Recommendations:

1. Consider refactoring `calculate_mint_amount` to multiply before division.
2. Or adding checks for currency_decimal and shares_decimals.

Solv: Resolved with `@5039429002 ...`

Zenith: Verified.

[L-4] Missing recovery Id normalization

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault.rs#L385

Description:

The vault contract's `withdraw` does not normalize the `recovery_id` parameter before passing it to `secp256k1_recover`, unlike the bridge contract, which handles ethereum-style v value(27,28). This can cause legitimate signatures to revert withdrawal txs.

The vault contract accepts recovery_id and passes it directly to `secp256k1_recover`, which expects recovery key id of 0, 1.

```
// Recover public key and compare with stored 65-byte uncompressed key
let recovered = env
    .crypto()
    .secp256k1_recover(&digest, &signature, recovery_id);
```

In contrast, the bridge contract normalizes the recovery ID:

```
let v_byte = sig_array[64];
let recovery_id = if v_byte ≥ ETHEREUM_V_OFFSET {
    (v_byte - ETHEREUM_V_OFFSET) as u32
} else {
    v_byte as u32
};
```

Recommendations:

Consider adding recovery_id normalization before calling `secp256k1_recover`.

Solv: Resolved with `@01e53e95ed ...`

Zenith: Verified.

4.3 Informational

A total of 3 informational findings were identified.

[I-1] Bridge missing direct burn path for EOA users

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- bridge.rs#L235

Description:

The bridge contract's `redeem` only supports the allowance-based authorization path (`burn_from`), requiring users to approve the bridge contract in advance. Unlike the vault contract which provides both direct burn and allowance-based paths.

```
fn redeem(env: Env, from: Address, token_address: Address, amount: i128,
    receiver: Bytes) {
    from.require_auth();
...
    let token_client = Token**Solv:**:new(&env, &token_address);
    token_client.burn_from(&env.current_contract_address(), &from, &amount);

}
```

The vault's internal implementation switches between paths:

```
// Burn user's shares using the appropriate method
if use_burn_from {
    // Use burn_from with allowance (vault as spender)
    token_client.burn_from(&env.current_contract_address(), from, &shares);
} else {
    // Use direct burn (requires caller to be the token owner)
    token_client.burn(from, &shares);
}
```

Recommendations:

Consider adding a direct burn path for `redeem` method.

Solv: Acknowledged. This is by design, and the bridge intentionally only supports the allowance-based burn_from flow.

[I-2] Unused admin storage key in Bridge

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- bridge.rs#L57

Description:

In the bridge contract, `Admin` variant of `BridgeDataKey` is never used. The admin access control is implemented through `Ownable`, making this storage key redundant.

In addition, the bridge contract does not provide a `get_admin()` query method like other contracts (vault, fungible-token).

Recommendations:

Consider removing the unused `Admin` variant.

Solv: Resolved with `@41b2636607 ...`

Zenith: Verified.

[I-3] Misleading error handling on withdraw

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- vault/src/vault.rs

Description:

In the vault withdraw flow, the `request_hash` parameter is hex-encoded using a helper that implicitly assumes a maximum input length of 32 bytes. If a caller supplies a longer `request_hash`, this helper panics and surfaces a `VaultError::InvalidVerifierKey`.

The failure is caused by invalid user input (an oversized `request_hash`), yet the emitted error indicates a verifier public key misconfiguration. This misclassification obscures the true root cause of the failure, making operational debugging, alerting, and monitoring more difficult.

Recommendations:

We recommend using a specific error message.

Solv: Resolved with `@0f2d90b434 ...`

Zenith: Verified.