Stellar-strkey

Stellar

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

DATE OF ENGAGEMENT
Feb 9, 2026 - Mar 7, 2026

SOURCE CODE
rs-stellar-strkey
rs-stellar-strkey

ASSESSED COMMIT ID
9c0cd13
04a5d08

SEVERITY BREAKDOWN

10	9	1	0
All Findings	Solved	Acknowledged	Risk Accepted

INTRODUCTION

This security assessment was commissioned by **Stellar**, a blockchain network designed to facilitate fast, low-cost cross-border payments and asset transfers.

The review was conducted by Halborn's experienced security team, focusing on the rs-stellar-strkey repository's Rust StrKey encoding/decoding library and optional CLI tooling corresponding to the v0.0.13 and v0.0.16 releases (commits 04a5d08 and 9c0cd13), from February 12th, 2026, to March 6th, 2026.

ASSESSMENT SUMMARY

The team at Halborn assigned a full-time security engineer to verify the security of the StrKey encoding/decoding library and CLI tooling. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Validate the correctness and security of StrKey encode/decode behavior (including CRC-16 verification and strict payload validation) under untrusted inputs.
- Identify and eliminate secret-handling risks (e.g., accidental disclosure via formatting, serialization, or CLI output) for private key material.
- Assess cross-version consistency between **rs-stellar-strkey v0.0.13** and **v0.0.16** to prevent host vs SDK behavioral drift and non-roundtrippable outputs.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were addressed by the `Stellar team`. The main recommendations were the following:

- `Redact private key material across all output channels, and require explicit opt-in for any secret exposure.`
- `Implement proper secret zeroization for PrivateKey and intermediate decode buffers to reduce memory-residency risk.`
- `Reduce build fragility by making the Git revision metadata optional and providing a safe fallback when the environment variable is missing. Ensure revision output is consistently normalized across build environments.`
- `Avoid large intermediate allocations during decoded signed-payload JSON deserialization by rejecting oversized hex inputs before allocating/decoding into bounded buffers.`

TEST APPROACH AND METHODOLOGY

A layered review approach was followed using the artifacts supplied in the repository and accompanying documentation. The sequence of work was as follows:

- Compare **rs-stellar-strkey v0.0.13** and **v0.0.16** to identify behavioral drift, then trace all changes affecting accepted/rejected inputs and round-trip behavior.
- Perform targeted manual code review of the high-risk parsing pipeline (base32 decode/encode, version dispatch, payload layout checks) and CRC-16 verification.
- Audit formatting/serialization/CLI surfaces for accidental secret disclosure, prioritizing any path that could emit private key material to stdout/logs/telemetry.
- Review robustness under untrusted inputs, focusing on panic conditions, unbounded allocations/OOM patterns, and lossy error handling that can hinder safe integrations.
- Validate findings with concrete exploit paths and minimal PoCs where applicable (e.g., encode→decode mismatches and CLI behaviors).

Risk Methodology

Halborn assesses the severity of findings using either the Common Vulnerability Scoring System (CVSS) framework or the Impact/Likelihood Risk scale, depending on the engagement. CVSS is an industry standard framework for communicating characteristics and severity of vulnerabilities in software. Details can be found in the CVSS Specification Document published by F.I.R.S.T.

Vulnerabilities or issues observed by Halborn scored on the Impact/Likelihood Risk scale are measured by the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

Scope

REPOSITORIES

(a) Repository: rs-stellar-strkey

(b) Assessed Commit ID: 9c0cd13dfda94a17b7a58498905016f2f31c7c88

(c) Items in scope:

- src 15 files
  - bin/stellar-strkey 1 file
    - main.rs Rust
  - cli 5 files
    - decode.rs Rust
    - encode.rs Rust
    - mod.rs Rust
    - version.rs Rust
    - zero.rs Rust
  - convert.rs Rust
  - crc.rs Rust
  - decoded_json_format.rs Rust
  - ed25519.rs Rust
  - error.rs Rust
  - lib.rs Rust
  - strkey.rs Rust
  - typ.rs Rust
  - version.rs Rust
- build.rs Rust

(a) Repository: rs-stellar-strkey

(b) Assessed Commit ID: 04a5d0856a01f0be3dd668abc6d25a0ce8923d36

(c) Items in scope:

- src 11 files
  - bin/stellar-strkey 3 files
    - decode.rs Rust
    - main.rs Rust
    - version.rs Rust
  - convert.rs Rust
  - crc.rs Rust
  - ed25519.rs Rust
  - error.rs Rust
  - lib.rs Rust
  - strkey.rs Rust
  - typ.rs Rust
  - version.rs Rust
- build.rs Rust

REMEDIATION COMMIT ID:

- d626f04146f4f3f1df52025728746149c59de0a0
- github.com/stellar/rs-stellar-strkey/commit/4e99b5f32014865d9583c921ea456cf16b7b816c
- e06b99771aefbd88ffa88306ae25fa488038fd53
- b83c2e3253f6f22be17085721bc46d5ee172a079
- cf1337bc62d78b07c13eb490e6df620ee3e4b37f
- b59a454d9e2df8807ebd3c837a9adfec7007bb7b
- 8bac7c8ef4e4cc4864f4557491281adb04a253c1
- d16561c672b950d7af05939817df19f3a1f63299
- 4b897e218b8f9a43cff0831d761d1e150e7cfb59
- e7eb20b95ea08672fa697289e8ba0041faef103a

Out-of-Scope: New features/implementations after the remediation commit IDs.

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Improper Exposure of Stellar Private Key Material	High	7.1	SOLVED 05/12/2026
HAL-002	Unbounded Allocation And Panic Conditions On Attacker-Controlled Input Sizes	Medium	6.2	SOLVED 05/07/2026
HAL-003	PrivateKey Secret Seed Persists in Stack Memory Due to Improper Zeroization	Medium	4.7	SOLVED 05/07/2026
HAL-004	SignedPayload Debug Output Mislabels Type	Low	2.8	SOLVED 05/07/2026
HAL-005	Large Intermediate Allocation Before Bounds Check When Deserializing Decoded SignedPayload JSON	Low	2.5	SOLVED 05/07/2026
HAL-006	Build-Time Git Revision Integration Can Break Builds In Non-Git Environments	Informational	—	ACKNOWLEDGED 05/06/2026
HAL-007	Typos in Comments Reduce Clarity and Readability	Informational	—	SOLVED 05/06/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-008	Single DecodeError Variant Obscures Root Causes Of Decode Failures	Informational	—	SOLVED 05/14/2026
HAL-009	SignedPayload Encoding Can Produce Strkeys That The Decoder Rejects (Empty Inner Payload)	Informational	—	SOLVED 05/06/2026
HAL-010	Decoded Strkey JSON Deserialization Produces Misleading Errors When Multiple Keys Are Present	Informational	—	SOLVED 05/06/2026

Findings & Tech Details

HAL-001 SOLVED

Improper Exposure of Stellar Private Key Material

High · 7.1 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

In `rs-stellar-strkey` (v0.0.16 and v0.0.13), the `PrivateKey` and `Strkey` types provide no redaction of secret seed material across four independent output channels — `Debug` formatting, `Display` formatting, decoded JSON serialization, and CLI stdout — causing the full 32-byte Ed25519 secret seed to be emitted in plaintext wherever any of those channels is exercised.

Any actor with read access to the application's stdout, logs, error output, or debug strings is the attacker. The four exposure paths are:

- **Path 1 — `Display` formatting (`{}`):** `PrivateKey::Display` (ed25519.rs:66–70) calls `self.to_string()`, producing the full strkey-encoded secret (the `"S..."` string that is directly loadable into any Stellar wallet). `Strkey::Display` (strkey.rs:122–126) delegates to the same path. Any `format!("{key}")`, `log::info!("{key}")`, `println!("{key}")`, or error message containing a `Strkey::PrivateKeyEd25519` value emits the usable secret. This is confirmed by the crate's own test suite `tests/display.rs:17–27`).
- **Path 2 — `Debug` formatting (`{:?}`):** `PrivateKey` implements `Debug` (ed25519.rs:20–28) by printing all 32 raw key bytes as lowercase hex. `Strkey` *derives* `Debug` (strkey.rs:15), so the auto-generated implementation transitively delegates to `PrivateKey::Debug` for the `PrivateKeyEd25519` variant. Any code holding a `Strkey` value of unknown variant — including generic logging middleware, error chains, `assert_eq!` failure messages, and `unwrap()` panics — will emit the raw key bytes without any opt-in decision by the caller. This is confirmed by `tests/debug.rs:18–31`.
- **Path 3 — Decoded JSON serialization `serde-decoded` feature):** `Decoded<&PrivateKey>` serializes the 32 raw key bytes as a hex string under the field name `private_key_ed25519` (ed25519.rs:97–102, strkey.rs:153–155). Any call to `serde_json::to_string(&Decoded(&strkey))` for a private-key variant emits the raw bytes to a JSON string, which may then be written to logs, telemetry, or transmitted over the network.
- **Path 4 — CLI `decode` and `encode` stdout and process/shell exposure:** Both v0.0.16 CLI commands expose private key material from two directions. The `decode` command (cli/decode.rs:30–35) accepts an `S...` strkey as a positional argument and calls Path 3 directly, printing the raw hex key bytes as decoded JSON to stdout. The `encode` command (cli/encode.rs:28–34) accepts JSON containing raw `private_key_ed25519` hex bytes as a positional argument and emits the re-encoded `S...` strkey to stdout via `PrivateKey::Display` (Path 1). Both positional arguments are stored verbatim in shell

history and are visible in process listings (`ps aux` / `/proc/<pid>/cmdline`) for the lifetime of the process, constituting a passive zero-effort exposure to any local user on shared systems.

In v0.0.13 (which predates the `serde-decoded` format), the CLI `decode` command uses Path 2 directly, calling `println!("{strkey:?}")` (v0.0.13/src/bin/stellar-strkey/decode.rs:35).

Code Location

`PrivateKey::Display` outputs the strkey-encoded secret key — directly usable by any attacker:

RUST 66–71

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs (lines 66-70)
impl Display for PrivateKey {
    fn fmt(&self, f: &mut core::fmt::Formatter<'_>) -> core::fmt::Result {
        write!(f, "{}", self.to_string()) // emits "SBU2RRG..." — the full usable secret
    }
}
```

`PrivateKey::Debug` outputs all 32 raw key bytes as hex:

RUST 20–29

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs (lines 20-28)
impl Debug for PrivateKey {
    fn fmt(&self, f: &mut core::fmt::Formatter<'_>) -> core::fmt::Result {
        write!(f, "PrivateKey(")?;
        for b in &self.0 {
            write!(f, "{b:02x}")?;
        }
        write!(f, ")")
    }
}
```

`Strkey` derives `Debug`, causing transitive exposure for the `PrivateKeyEd25519` variant without any caller opt-in:

RUST 15–17

```rust
// rs-stellar-strkey-0.0.16/src/strkey.rs (line 15)
#[derive(Clone, Hash, PartialEq, Eq, PartialOrd, Ord, Debug)]
pub enum Strkey { ... }
```

`Decoded<&PrivateKey>` serializes raw key bytes as hex under `serde-decoded`:

RUST 97–103

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs (lines 97-101)
impl Serialize for Decoded<&PrivateKey> {
```

```
    fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
100         let Self(PrivateKey(bytes)) = self;
101         DecodedBorrowed(bytes).serialize(serializer) // raw 32 bytes as hex
    }
102 }
103
```

v0.0.16 CLI `decode` emits decoded JSON to stdout:

RUST 30–34

```
30  // rs-stellar-strkey-0.0.16/src/cli/decode.rs (lines 30-35)
    let strkey =
31      Strkey::from_str(&self.strkey).map_err(|e| Error::Decode(self.strkey.clone(),
32  e))?;
    let json = serde_json::to_string_pretty(&Decoded(&strkey)).unwrap();
33  println!("{json}"); // emits { "private_key_ed25519": "69a8c4cb..." } to stdout
```

v0.0.13 CLI `decode` emits `Debug` output directly to stdout:

RUST 34–35

```
34  // rs-stellar-strkey-0.0.13/src/bin/stellar-strkey/decode.rs (lines 34-35)
    println!("{strkey:?}"); // emits PrivateKeyEd25519(PrivateKey(69a8c4cb...))
35
```

v0.0.16 CLI `encode` takes raw hex key bytes as a positional JSON argument and re-emits the `S...` strkey via `Display` — private key material is exposed in both the input and output:

RUST 20–30

```
20  // rs-stellar-strkey-0.0.16/src/cli/encode.rs (lines 20-34)
    pub struct Cmd {
21      json: String, // positional arg — '{\"private_key_ed25519\":\"69a8c4cb...\"}'
22  stored in shell history and ps aux
    }

23  pub fn run(&self) -> Result<(), Error> {
24      let Decoded(strkey): Decoded<Strkey> =
25          serde_json::from_str(&self.json).map_err(Error::Json)?;
26      println!("{strkey}"); // Display — emits
27  "SBU2RRGLXH3E5CQHTD3ODLDF2BWDCYUSSBLLZ5GNW7JXHDIYKXZWHOKR" to stdout
        Ok(())
28  }
```

Impact

Recovery of the 32-byte secret seed gives the attacker complete signing authority over the corresponding Stellar account: they can sign arbitrary transactions, drain all assets, and issue account merges or configuration changes indistinguishable from the legitimate key holder. The most persistent risk is log and telemetry retention — once the key bytes appear in a log aggregator, they remain readable indefinitely after the application has terminated and the in-memory copy has been discarded.

PROOF OF CONCEPT

Code

The PoC code has been added to the `poc/poc_secret_seed_leak.sh` file:

SH

```sh
#!/usr/bin/env bash
set -euo pipefail

ROOT="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd)"

# Force a toolchain without changing global defaults.
# This avoids `rustup default ...` requirements on systems with no configured default.
export RUSTUP_TOOLCHAIN="${RUSTUP_TOOLCHAIN:-stable}"

echo "[*] Building CLI (features: cli)"
(
  cd "$ROOT"
  cargo build --quiet --features cli --bin stellar-strkey
)

echo "[*] Using a known test secret seed StrKey from the crate's own test vectors"
# Source of this constant: tests/tests.rs and tests/display.rs
SEED_STRKEY="SBU2RRGLXH3E5CQHTD3ODLDF2BWDCYUSSBLLZ5GNW7JXHDIYKXZWHOKR"
echo "[*] Seed StrKey: ${SEED_STRKEY}"
echo "[*] Decoding via CLI; output SHOULD contain private_key_ed25519 with 64 hex chars"

(
  cd "$ROOT"
  cargo run --quiet --features cli --bin stellar-strkey -- decode "$SEED_STRKEY"
)
```

Execution

BASH

```bash
./poc/poc_secret_seed_leak.sh
```

Result

```
{
  "private_key_ed25519": "69a8c4cbb9f64e8a0798f6e1ac65d06c3162929056bcf4cdb7d3738d1855f363"
}
```

Analysis

- Script uses a known test-vector secret seed StrKey (from the crate's own tests) and then runs `stellar-strkey decode`.
- The JSON output contains a `private_key_ed25519` field equal to 64 hex chars, proving the CLI emits secret-seed bytes to stdout when decoding a private-key StrKey.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- **Redact** `PrivateKey::Debug` — Replace the current implementation with a constant-output redacted form: `write!(f, "PrivateKey([REDACTED])")`. This eliminates Path 2 for both the inner type and any transitive `{:?}` usage on `Strkey`.
- **Redact** `PrivateKey::Display` — Replace `self.to_string()` (which outputs the full strkey-encoded secret) with a redacted form such as `write!(f, "[REDACTED]")`. Alternatively, remove the `Display` impl entirely and force callers to call `to_string()` explicitly, making the exposure opt-in. This eliminates Path 1 — the most operationally dangerous channel because the output is the directly usable secret string.
- **Implement a custom** `Debug` **for** `Strkey` — Remove `Debug` from `Strkey`'s `#[derive(...)]` list and implement it manually, redacting the `PrivateKeyEd25519` variant (e.g., `PrivateKeyEd25519([REDACTED])`) while allowing other variants to format normally. This prevents transitive leakage through generic code that holds a `Strkey` of an unknown variant.
- **Redact or gate decoded JSON serialization for private keys** — In `Decoded<&PrivateKey>`, either serialize a redacted placeholder by default, or gate raw-bytes serialization behind an explicit opt-in wrapper type (e.g., `UnsafeDecoded<&PrivateKey>`). The CLI `decode` command should refuse to emit decoded JSON for private-key inputs unless an explicit flag such as `--expose-secret` is provided, and must print a prominent warning when that flag is used.
- **Accept secret material via stdin or file, not as positional arguments** — Both `decode` (strkey argument) and `encode` (JSON argument containing raw hex bytes) pass private key material as positional CLI arguments, exposing it in shell history and process listings. Replace or supplement both with a `--from-stdin` or `--from-file <path>` mode to structurally eliminate Path 4's passive exposure. Document explicitly in CLI help text that passing secret key material as a positional argument is unsafe on shared or logged systems.
- **Audit** `serde` **feature** `SerializeDisplay` — When compiled with `features = ["serde"]` (distinct from `serde-decoded`), `PrivateKey` derives `serde_with::SerializeDisplay`, which serializes using the `Display` output (the full strkey-encoded secret string). Once `Display` is redacted per the above, this path becomes safe; confirm the redaction covers this channel before closing.
- **Add regression tests** — Add tests asserting that `format!("{:?}", strkey)`, `format!("{}", strkey)`, and `serde_json::to_string(&Decoded(&strkey))` for a `PrivateKeyEd25519` value do **not** contain the raw key bytes or the strkey-encoded secret string, to prevent redaction regressions.

REMEDIATION COMMENT

Addressed in: d626f04146f4f3f1df52025728746149c59de0a0

SOLVED: The **Stellar team** solved this finding across commits `d626f04` and `32108d1` by removing `PrivateKeyEd25519` from the `Strkey` enum, redacting `PrivateKey::Debug` to `[REDACTED]`, gating all encoding, display, and serialization paths behind an explicit `Unredacted<T>` wrapper (eliminating Paths 1, 2, and 3), and removing the positional CLI arguments from `decode` and `encode` entirely — making `pub struct Cmd {}` empty with stdin as the only input path — which structurally eliminates the shell history and process-listing exposure of Path 4.

HAL-002 SOLVED

Unbounded Allocation And Panic Conditions On Attacker-Controlled Input Sizes

Medium · 6.2 AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

DESCRIPTION

`stellar-strkey` **v0.0.13** contains multiple code paths where attacker-controlled input sizes can cause unbounded allocations (and in one case a panic), creating a denial-of-service risk.

Concrete instances:

- **Instance A (decode path)**: base32 decoding allocates a `Vec<u8>` sized to the full decoded length **before** any type/version/payload bounds validation.
- **Instance B (SignedPayload encode path)**: `SignedPayload::to_string()` allocates a buffer proportional to `payload.len()`, and uses `.expect(...)` on a `u32` conversion which can panic for extreme lengths; it also does not enforce the 64-byte inner payload maximum that the decoder requires.

Code Location

Instance A — `convert::decode` unbounded allocation before validation:

RUST 17–34

```rust
pub fn decode(s: &str) -> Result<(u8, Vec<u8>), DecodeError> {
    let data = data_encoding::BASE32_NOPAD.decode(s.as_bytes()).ok();
    if let Some(data) = data {
        if data.len() < 3 {
            return Err(DecodeError::Invalid);
        }
        let ver = data[0];
        let (data_without_crc, crc_actual) = data.split_at(data.len() - 2);
        let crc_expect = checksum(data_without_crc);
        if crc_actual != crc_expect {
            return Err(DecodeError::Invalid);
        }
        let payload = &data_without_crc[1..];
        Ok((ver, payload.to_vec()))
    } else {
        Err(DecodeError::Invalid)
    }
}

```

Instance B — `SignedPayload::to_string` unbounded allocation and panic condition:

RUST 243–258

```rust
impl SignedPayload {
  pub fn to_string(&self) -> String {
      let inner_payload_len = self.payload.len();
      let payload_len = 32 + 4 + inner_payload_len + (4 - inner_payload_len % 4) % 4;
```

```
        let inner_payload_len_u32: u32 = inner_payload_len
            .try_into()
            .expect("payload length larger than u32::MAX");

        let mut payload = vec![0; payload_len];
        payload[..32].copy_from_slice(&self.ed25519);
        payload[32..32 + 4].copy_from_slice(&(inner_payload_len_u32).to_be_bytes());
        payload[32 + 4..32 + 4 + inner_payload_len].copy_from_slice(&self.payload);

        encode(version::SIGNED_PAYLOAD_ED25519, &payload)
    }
```

Impact

- **Availability**: potential OOM / process termination / severe slowdown on oversized inputs; `SignedPayload::to_string()` can also panic for extreme lengths.
- **Correctness**: `SignedPayload::to_string()` can emit signed-payload StrKeys that decoders reject (inner payload > 64).
- **Host vs SDK divergence**: **v0.0.16**'s bounded `heapless` decoding will reject oversize inputs without large allocations, while **v0.0.13** may allocate heavily and fail differently (including OOM). Additionally, **v0.0.16** bounds `SignedPayload` payloads to 64 bytes at the type level, preventing oversized encodings that **v0.0.13** can generate.

PROOF OF CONCEPT

Code

The PoC code has been added to the `rs-stellar-strkey-0.0.13/examples/poc_decode_unbounded_alloc.rs` file:

RUST

```rust
//! PoC for finding: unbounded allocation on oversized decode inputs.
//!
//! This example constructs a very large (but base32-valid) input string and
//! passes it to `Strkey::from_string()`. Internally, `convert::decode()` calls
//! `BASE32_NOPAD.decode(...)`, which allocates a `Vec<u8>` sized to the decoded
//! length *before* any version/CRC/payload bounds validation.
//!
//! The decode is expected to return `Err(DecodeError::Invalid)` (CRC mismatch),
//! but only after the proportional allocation and decode work are done.

use std::time::Instant;

fn main() {
    let nchars: usize = std::env::args()
        .nth(1)
        .as_deref()
        .unwrap_or("200000")
        .parse()
        .expect("nchars must be an integer");
```

```
    // Rough lower-bound estimate for decoded length (base32 is 5 bits/char).
    // Exact behavior depends on the base32 implementation and leftover bits.
    let approx_decoded = (nchars.saturating_mul(5)) / 8;

    eprintln!("poc_decode_unbounded_alloc:");
    eprintln!("- base32 input chars: {nchars}");
    eprintln!("- approx decoded bytes: ~{approx_decoded}");
    eprintln!("- note: wrap this command with '/usr/bin/time -l' on macOS to observe RSS");
    eprintln!();

    // Use 'A' (base32 value 0) to keep the input base32-decodable.
    // In a real attack, this oversized string would be attacker-supplied.
    let s = "A".repeat(nchars);

    let start = Instant::now();
    let res = stellar_strkey::Strkey::from_string(&s);
    let elapsed = start.elapsed();

    eprintln!("result: {res:?}");
    eprintln!("elapsed: {:.2?}", elapsed);

    // Ensure `s` is not trivially optimized away in release builds.
    eprintln!("input_len_check: {}", s.len());
}
```

Execution

SH

```
cargo build --example poc_decode_unbounded_alloc
for n in 2000000 5000000 10000000 20000000; do
  echo "=== nchars=$n ==="
  /usr/bin/time -l ./target/debug/examples/poc_decode_unbounded_alloc "$n"
done
```

Result

SH

```
=== nchars=2000000 ===
poc_decode_unbounded_alloc:
- base32 input chars: 2000000
- approx decoded bytes: ~1250000
- note: wrap this command with '/usr/bin/time -l' on macOS to observe RSS

result: Err(Invalid)
elapsed: 73.05ms
input_len_check: 2000000
        0.28 real         0.07 user         0.00 sys
             6225920  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 502  page reclaims
                   0  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                  31  involuntary context switches
          1109571686  instructions retired
           226592331  cycles elapsed
             5768256  peak memory footprint
```

```
=== nchars=5000000 ===
poc_decode_unbounded_alloc:
- base32 input chars: 5000000
- approx decoded bytes: ~3125000
- note: wrap this command with '/usr/bin/time -l' on macOS to observe RSS

result: Err(Invalid)
elapsed: 167.53ms
input_len_check: 5000000
        0.17 real         0.16 user         0.00 sys
            12910592  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 910  page reclaims
                   0  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                  10  involuntary context switches
          2755568295  instructions retired
           537791569  cycles elapsed
            12403584  peak memory footprint
=== nchars=10000000 ===
poc_decode_unbounded_alloc:
- base32 input chars: 10000000
- approx decoded bytes: ~6250000
- note: wrap this command with '/usr/bin/time -l' on macOS to observe RSS

result: Err(Invalid)
elapsed: 338.20ms
input_len_check: 10000000
        0.34 real         0.33 user         0.00 sys
            24166400  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                1597  page reclaims
                   0  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                   9  involuntary context switches
          5500853784  instructions retired
          1097924520  cycles elapsed
            23659392  peak memory footprint
=== nchars=20000000 ===
poc_decode_unbounded_alloc:
- base32 input chars: 20000000
- approx decoded bytes: ~12500000
- note: wrap this command with '/usr/bin/time -l' on macOS to observe RSS

result: Err(Invalid)
elapsed: 676.37ms
input_len_check: 20000000
        0.68 real         0.67 user         0.00 sys
            46612480  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                2967  page reclaims
                   0  page faults
                   0  swaps
```

```
                0  block input operations
                0  block output operations
                0  messages sent
                0  messages received
                0  signals received
                0  voluntary context switches
               33  involuntary context switches
      10993017778  instructions retired
       2160999588  cycles elapsed
         46121920  peak memory footprint
```

Analysis

Even though the input is invalid `Err(Invalid)`), runtime cost scales roughly linearly with attacker-controlled input length because decoding allocates/works *before* CRC/version/payload validation.

Concretely:

- **Time scaling (CPU-bound decode work)**: elapsed time grows from ~**73ms** (2M chars) → **168ms** (5M) → **338ms** (10M) → **676ms** (20M), which is near-linear.
- **Memory scaling (allocation proportional to decoded length)**: max RSS grows from ~**6.2MB** → **12.9MB** → **24.2MB** → **46.6MB** as input grows, consistent with the base32 decode producing a `Vec<u8>` sized to the decoded output and additional copies `payload.to_vec()`).

This demonstrates the DoS pattern: an attacker can feed oversized base32 strings that are ultimately rejected, but still force **predictable CPU time and heap growth proportional to input size** prior to rejection.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Adopt a **bounded decode** strategy (as in **v0.0.16**): precompute `decode_len`, reject if over maximum binary length, decode into fixed-capacity buffers.
- If **v0.0.13** must remain allocation-based, impose a **hard maximum input length** (and/or maximum decoded length) before calling `data_encoding` decode.
- Enforce the signed-payload maximum by bounding `SignedPayload.payload` to **<= 64 bytes** (type-level, as in **v0.0.16**, or via explicit checks in `to_string()`).
- Remove panic conditions in encoding paths by using fallible conversions/returns (or by bounding lengths so conversions cannot fail).

REMEDIATION COMMENT

Addressed in: github.com/stellar/rs-stellar-strkey/commit/4e99b5f32014865d9583c921ea456cf16b7b816c

SOLVED: The **Stellar team** solved this finding in the specified commit by replacing all heap-allocated decode/encode buffers with heapless fixed-capacity types parameterized by compile-time constants, and bounding SignedPayload.payload to 64 bytes at the type level, making unbounded allocation and the panic condition structurally impossible.

Client Note:
The issues identified were resolved in the v0.0.16 release. Regarding Host vs SDK Divergence with the failure path for large inputs and SignedPayload. The SignedPayload is mentioned but strkey decoding with SignedPayloads is not used in the soroban-sdk. The soroban-sdk v25 only uses MuxedAddress stellar_strkey decoding that are already fixed length, so we do not believe any meaningful divergence occurs. The risk of divergence has also been eliminated in soroban-sdk v26 that no longer uses a different stellar-strkey version to do any decoding in contracts.

HAL-003 SOLVED

PrivateKey Secret Seed Persists in Stack Memory Due to Improper Zeroization

• Medium · 4.7 AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:N/A:N

DESCRIPTION

`PrivateKey` in `src/ed25519.rs` holds a raw 32-byte Ed25519 secret seed as an inline `pub [u8; 32]` with `Copy` semantics and no mechanism to clear that material when the value goes out of scope. Any attacker with access to the process's address space — through a core dump, `/proc/<pid>/mem`, a kernel-readable swap partition, or physical DRAM extraction — can recover the secret seed after the application has nominally "dropped" the key.

`PrivateKey` is stack-allocated (inline fixed array, no heap involvement). It derives `Clone` and `Copy`, which means every by-value function call silently stamps an independent 32-byte copy onto a new stack frame; none of those copies is explicitly cleared on scope exit.

The decode path in `convert.rs` compounds this:

1. `decode()` materializes a `Vec<u8, B>` (35-byte stack buffer: version byte + 32-byte seed + 2-byte CRC) and a separate `Vec<u8, P>` (32-byte payload slice). Both are `heapless` types — stack-only, no allocator — so the raw bytes persist in their stack frame until the frame is overwritten by unrelated code.
2. `from_payload()` copies those 32 bytes into the returned `PrivateKey` struct, creating a second live copy.
3. The caller's binding (and any intermediate copies produced by `Copy` elision or by-value passing) remain on prior stack frames that have not yet been reused.

No `Drop` implementation exists for `PrivateKey`, and neither `zeroize`, `secrecy`, nor any volatile-write mechanism appears anywhere in the dependency tree (`Cargo.toml` confirms this).

Code Location

`PrivateKey` struct — `Copy` derived, no `Drop`, public inner bytes:

RUST 13–15

```rust
//rs-stellar-strkey-0.0.16/src/ed25519.rs
#[derive(Clone, Copy, Hash, PartialEq, Eq, PartialOrd, Ord)]
pub struct PrivateKey(pub [u8; 32]);
```

`decode()` in `convert.rs` — two stack-resident `heapless::Vec` buffers holding raw key bytes, neither cleared on return:

RUST 96–104

```rust
//rs-stellar-strkey-0.0.16/src/convert.rs
pub fn decode<const P: usize, const B: usize>(s: &[u8]) -> Result<(u8, Vec<u8, P>), DecodeError> {
    // ...
    let mut data: Vec<u8, B> = Vec::new();          // 35 bytes: ver + seed + CRC — not zeroed
    // ... base32 decode into data ...
    let payload_data = &data_without_crc[1..];
    let payload: Vec<u8, P> = Vec::from_slice(payload_data).unwrap();  // 32-byte copy — not zeroed
    Ok((ver, payload))
}

```

`Cargo.toml` — no `zeroize`, `secrecy`, or equivalent dependency:

JSON

```toml
// rs-stellar-strkey-0.0.16/Cargo.toml
[dependencies]
data-encoding = { version = "2.6.0", default-features = false }
heapless = { version = "0.8", default-features = false }
```

Impact

An attacker with read access to the process memory (core dump, `/proc/<pid>/mem`, swap page, or cold-boot acquisition) can scan for the 32-byte secret seed — or its strkey-encoded form — in stack pages that have not yet been reused, recovering full signing authority over the associated Stellar account. Because `Copy` is derived, each by-value use of `PrivateKey` in the decode and call chain leaves an additional residual copy that independently extends the window of exposure.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Add `zeroize` and enable the `heapless` zeroize feature to satisfy the `Zeroize` trait bound at compile time.
- Implement `Zeroize` and `ZeroizeOnDrop` on `PrivateKey` and remove `Copy`.
- Wrap intermediate `heapless::Vec` buffers in `Zeroizing<>` inside `convert::decode()` so both the 35-byte binary buffer and the 32-byte payload buffer are cleared when the function returns.
- Consider wrapping `PrivateKey` fields with `secrecy::Secret<[u8; 32]>` in higher-level consumers to prevent accidental logging or display of the inner bytes, as defense-in-depth on top of zeroization.

REMEDIATION COMMENT

Addressed in: e06b99771aefbd88ffa88306ae25fa488038fd53

SOLVED: The **Stellar team** solved this finding in the specified commit by deriving `Zeroize` / `ZeroizeOnDrop` on `PrivateKey`, removing `Copy`, adding the **zeroize** dependency, and routing `PrivateKey::from_slice` through the new `decode_zeroizing` out-parameter variant that zeroes all intermediate scratch buffers.

HAL-004 SOLVED

SignedPayload Debug Output Mislabels Type

Low · 2.8 AV:L/AC:L/PR:L/UI:R/S:U/C:N/I:L/A:N

DESCRIPTION

`stellar-strkey` **v0.0.13**'s `Debug` implementation for `ed25519::SignedPayload` writes the label `MuxedAccount(` instead of `SignedPayload(`, which can mislead operators and tooling that rely on logs/telemetry to identify which StrKey type is being handled.

The issue is triggered whenever a program logs/prints `SignedPayload` (or `Strkey::SignedPayloadEd25519`) using `{:?}`. In practice, this can cause incorrect log classification and increase time-to-diagnosis during debugging or incident response.

While the label is corrected to `SignedPayload(` in **v0.0.16**, the `Debug` output still prints the full inner payload bytes in hex, so the "payload-bytes-in-logs" exposure pattern remains unless explicitly redacted/truncated.

Code Location

`SignedPayload` `Debug` writes the wrong type label:

RUST 216–241

```rust
impl Debug for SignedPayload {
    fn fmt(&self, f: &mut core::fmt::Formatter<'_>) -> core::fmt::Result {
        write!(f, "MuxedAccount(")?;
        write!(
            f,
            "{}",
            &self
                .ed25519
                .iter()
                .map(|b| format!("{b:02x}"))
                .collect::<String>()
        )?;
        write!(f, ", ")?;
        write!(
            f,
            "{}",
            &self
                .payload
                .iter()
                .map(|b| format!("{b:02x}"))
                .collect::<String>()
        )?;
        write!(f, ")")?;
        Ok(())
    }
}

```

Impact

- **Observability / correctness**: misleading logs/telemetry that can be misclassified as a muxed account instead of a signed payload.
- **Operational risk**: increased debugging and incident-response effort due to incorrect labeling.

RECOMMENDATION

To mitigate this issue, we recommend fixing the `Debug` label to print `SignedPayload(`.

REMEDIATION COMMENT

Addressed in: b83c2e3253f6f22be17085721bc46d5ee172a079

SOLVED: The **Stellar team** solved this finding in the specified commit by correcting the `Debug` label from `"MuxedAccount("` to `"SignedPayload("`.

HAL-005 SOLVED

Large Intermediate Allocation Before Bounds Check When Deserializing Decoded SignedPayload JSON

● Low · 2.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

In **rs-stellar-strkey v0.0.16**, when `serde-decoded` is enabled, deserializing `Decoded<SignedPayload>` uses an `alloc::vec::Vec<u8>` for the hex-decoded `payload` field. For oversized JSON inputs, the hex deserializer will allocate a vector proportional to the provided hex string *before* the code attempts to convert it into the bounded `heapless::Vec<u8, 64>`. This creates a “bounded target, potentially large intermediate allocation” denial-of-service pattern.

The attacker is **any local user (or untrusted pipeline input source)** that can cause a program to deserialize `Decoded<SignedPayload>` from untrusted JSON.

Exploit path:

- Provide JSON for the `signed_payload_ed25519` variant with a `payload` field containing an extremely large hex string.
- `serde_with::hex::Hex` decodes it into `alloc::vec::Vec<u8>` (allocation proportional to input).
- Only after allocation does the code attempt to downcast into `Vec<u8, 64>`, at which point it errors — but the memory cost has already been paid.

Code Location

v0.0.16 decoded signed payload deserializer uses an unbounded `alloc::vec::Vec<u8>`:

RUST 502-514

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs
struct DecodedOwned {
    #[serde_as(as = "serde_with::hex::Hex")]
    ed25519: [u8; 32],
    #[serde_as(as = "serde_with::hex::Hex")]
    payload: alloc::vec::Vec<u8>,
}

// ...
payload: payload
    .as_slice()
    .try_into()
    .map_err(|_| de::Error::custom("payload too large"))?,
```

v0.0.16 CLI `encode` command accepts arbitrary JSON from the user and deserializes it:

RUST 30–33

```rust
// rs-stellar-strkey-0.0.16/src/cli/encode.rs
let Decoded(strkey): Decoded<Strkey> =
    serde_json::from_str(&self.json).map_err(Error::Json)?;
println!("{strkey}");

```

Impact

Local denial-of-service (high memory use / OOM / crash) when parsing untrusted JSON. This is most relevant for **library consumers** deserializing decoded JSON from files/network/IPC where input sizes may be large; the CLI `encode` path takes JSON as a command-line argument and is typically constrained by OS argument size limits (reducing, but not eliminating, the worst-case impact).

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- In any application deserializing `Decoded<SignedPayload>` or `Decoded<Strkey>` from untrusted JSON, enforce a maximum input size before calling `serde_json::from_str`.
- In the CLI `encode` command, consider enforcing a reasonable maximum input size for the JSON string (defense-in-depth) before calling `serde_json::from_str`.

REMEDIATION COMMENT

Addressed in: cf1337bc62d78b07c13eb490e6df620ee3e4b37f

SOLVED: The **Stellar team** solved this finding across commits 4cc3199 and cf1337b by capping the CLI `encode` command's JSON input at 10 KiB before deserialization, and providing documentation to highlight the need for this in using applications.

HAL-006 ACKNOWLEDGED

Build-Time Git Revision Integration Can Break Builds In Non-Git Environments

● Informational · AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

DESCRIPTION

In **rs-stellar-strkey v0.0.16 and v0.0.13**, the crate invokes `crate_git_revision::init()` from its build script, then uses `env!("GIT_REVISION")` in `src/lib.rs` to populate `VERSION.rev`. This creates a hard build-time dependency on `GIT_REVISION` being set.

Per the confirmed findings in the in-scope dependency **crate-git-revision-0.0.6**, `init()` may fail to set `GIT_REVISION` in common situations (e.g., not a git checkout, git missing, metadata missing), causing downstream compilation using `env!("GIT_REVISION")` to fail.

In practice, `env!("GIT_REVISION")` fails the build only when the variable is **unset** at compile time. In some environments, the build script may successfully set `GIT_REVISION` to an **empty string**, in which case compilation succeeds but the embedded revision is empty/unstable. The practical risk is therefore that builds **may fail to build, or may embed an empty/incorrect revision**, depending on how `crate_git_revision::init()` behaves in the build environment.

Separately (also per confirmed dependency findings), the revision string may be derived from external sources (e.g., `git describe` stdout) without trimming/normalization. Because `VERSION.rev` is embedded into the artifact and printed by the CLI `version` command, a trailing newline can lead to confusing multi-line output and complicate downstream parsing/alerting. While "build scripts execute third-party code" is a broad supply-chain consideration in Rust/Cargo generally, the issue here is the **demonstrable, local build fragility** created by a hard compile-time requirement on `GIT_REVISION`.

Code Location

v0.0.16 build script unconditionally invokes the helper:

RUST 1–4

```rust
// rs-stellar-strkey-0.0.16/build.rs
pub fn main() {
    crate_git_revision::init();
}
```

v0.0.13 build script has the same pattern:

RUST 1–4

```rust
// rs-stellar-strkey-0.0.13/build.rs
pub fn main() {
    crate_git_revision::init();
}
```

v0.0.16 hard-requires `GIT_REVISION` at compile time:

RUST 11–15

```rust
// rs-stellar-strkey-0.0.16/src/lib.rs
pub const VERSION: Version = Version {
    pkg: env!("CARGO_PKG_VERSION"),
    rev: env!("GIT_REVISION"),
};
```

v0.0.13 has the same `env!("GIT_REVISION")` usage:

RUST 9–13

```rust
// rs-stellar-strkey-0.0.13/src/lib.rs
pub const VERSION: Version = Version {
    pkg: env!("CARGO_PKG_VERSION"),
    rev: env!("GIT_REVISION"),
};
```

v0.0.16 CLI prints the embedded revision:

RUST 10–12

```rust
// rs-stellar-strkey-0.0.16/src/cli/version.rs
let v = VERSION;
println!("stellar-strkey {} ({})", v.pkg, v.rev);
```

Impact

- **Build/release availability**: builds can fail hard in non-git / restricted environments if `GIT_REVISION` ends up **unset** at compile time (due to build script failure to emit it).
- **Observability**: if the embedded revision contains trailing newlines, `version` output can become multi-line and harder to parse reliably.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Replace `env!("GIT_REVISION")` with `option_env!("GIT_REVISION").unwrap_or("unknown")` so builds succeed when git metadata is unavailable.
- In the build integration, ensure the revision value is single-line and trimmed (defense-in-depth: strip `\r\n`).
- Consider surfacing a warning when revision derivation fails (either in `build.rs` or by configuring the dependency behavior) so users understand they are getting `"unknown"`.

REMEDIATION COMMENT

Addressed in: b59a454d9e2df8807ebd3c837a9adfec7007bb7b

ACKNOWLEDGED: The **Stellar team** partially solved this finding by trimming whitespace from git SHA values at source in **crate-git-revision** (eliminating the trailing-newline sub-issue), but made a business decision to not alter the **stellar-strkey** library, acknowledging the residual build-fail risk in non-git environments as intentional behavior.

HAL-007 SOLVED

Typos in Comments Reduce Clarity and Readability

● Informational · AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

DESCRIPTION

A spelling error ("implementated") is present in the `src/crc.rs` component comment in both reviewed versions (0.0.13 and 0.0.16). Although minor, the misspelling is considered to reduce clarity for maintainers and reviewers and to hinder automated or manual searches for standardized terminology during audits.

Code Location

A typographical error is present in the v0.0.16 source comment:

RUST 1-2

```rust
// rs-stellar-strkey-0.0.16/src/crc.rs
// Mod crc implementated according to CCITT standards.
```

The same typographical error is present in the v0.0.13 source comment:

RUST 1-2

```rust
// rs-stellar-strkey-0.0.13/src/crc.rs
// Mod crc implementated according to CCITT standards.
```

Impact

Minor effect on documentation clarity and terminology consistency.

RECOMMENDATION

To mitigate this issue, we recommend correcting spelling in comments and identifiers: “implementated” → “implemented”.

REMEDIATION COMMENT

Addressed in: 8bac7c8ef4e4cc4864f4557491281adb04a253c1

SOLVED: The **Stellar team** solved this finding in the specified commit by applying the mentioned recommendation.

HAL-008 SOLVED

Single DecodeError Variant Obscures Root Causes Of Decode Failures

● Informational · AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

DESCRIPTION

The **rs-stellar-strkey** component exposes only a single `DecodeError::Invalid` variant, and key decode paths collapse multiple distinct failure modes into it. This prevents callers from distinguishing between invalid base32 input, too-short decoded data, checksum mismatches, unsupported version bytes, and type/payload layout violations.

While decoding still fails safely, lossy error reporting reduces observability and can lead to poor UX or harder-to-debug integrations.

Code Location

v0.0.16 error type contains only one variant:

RUST 2-6

```rust
// rs-stellar-strkey-0.0.16/src/error.rs
pub enum DecodeError {
    // TODO: Add meaningful errors for each problem that can occur.
    Invalid,
}
```

v0.0.13 uses the same single-variant error type (so host-side decode failures are similarly lossy, and can mask host-vs-SDK divergence where one side returns `Invalid` and the other side may terminate due to resource exhaustion):

RUST 1-5

```rust
// rs-stellar-strkey-0.0.13/src/error.rs
pub enum DecodeError {
    // TODO: Add meaningful errors for each problem that can occur.
    Invalid,
}
```

v0.0.16 example of distinct failures being mapped to the same `Invalid` error:

RUST 103-117

```rust
// rs-stellar-strkey-0.0.16/src/convert.rs
let data_len = data_encoding::BASE32_NOPAD
    .decode_len(s.len())
    .map_err(|_| DecodeError::Invalid)?;
if data_len < 3 {
    return Err(DecodeError::Invalid);
```

```
110 }
111 // ...
112 data_encoding::BASE32_NOPAD
113     .decode_mut(s, &mut data)
114     .map_err(|_| DecodeError::Invalid)?;
115 // ...
116 if crc_actual != crc_expect {
117     return Err(DecodeError::Invalid);
    }
```

Impact

Reduced visibility into why decoding failed (e.g., corrupted input vs. wrong type vs. checksum mismatch), complicating debugging, support, and metrics/telemetry.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Add structured `DecodeError` variants (e.g., `InvalidBase32`, `TooShort`, `ChecksumMismatch`, `UnsupportedVersion`, `InvalidPayload`, `InvalidPadding`) and preserve root causes where helpful.
- Keep `Display` messages stable/user-friendly for CLI consumers while allowing library callers to branch on specific variants.

REMEDIATION COMMENT

Addressed in: d16561c672b950d7af05939817df19f3a1f63299

SOLVED: The **Stellar team** solved this finding in the specified commit by expanding `DecodeError` with eight structured variants (`InvalidBase32`, `TooShort`, `TooLong`, `ChecksumMismatch`, `UnsupportedVersion`, `UnsupportedClaimableBalanceVersion`, `InvalidPayloadLength`, `InvalidPadding`) and updating every call site across **convert.rs**, **ed25519.rs**, and **strkey.rs** to return the appropriate variant.

HAL-009 SOLVED

SignedPayload Encoding Can Produce Strkeys That The Decoder Rejects (Empty Inner Payload)

● Informational · AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

`ed25519::SignedPayload::to_string()` can encode a signed-payload StrKey even when `self.payload` is empty, producing a 36-byte payload layout (32-byte key + 4-byte length + 0 payload + 0 padding). However, `ed25519::SignedPayload::from_payload()` rejects any payload shorter than 40 bytes (it enforces `MIN_LENGTH = 32 + 4 + 4`), so the crate can generate an output that it will later refuse to decode.

This behavior is not theoretical: the CLI `zero` subcommand constructs a `SignedPayloadEd25519` value with an empty payload (`payload: Default::default()`), so `stellar-strkey zero signed_payload_ed25519` will print a StrKey that `stellar-strkey decode` (and library decoding) will treat as invalid.

The mismatch is caused by the decoder enforcing `MIN_LENGTH = 40` as a total-length floor rather than directly requiring a minimum inner payload length of four bytes. Inner payload lengths of **1..=3** can still be accepted because trailing padding (**0..=3** bytes) can bring the total payload length to 40. The non-roundtrippable case identified here is specifically `inner_payload_len == 0`, which yields a 36-byte encoding that the decoder rejects. Whether `inner_payload_len == 0` should be considered valid is a specification decision; regardless, the crate currently produces an encoding that is not decodable under its own decoding rules.

Code Location

v0.0.16 encoder includes an empty payload (and zero padding) as a valid encoding:

RUST 366-372

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs
let inner_payload_len = self.payload.len();
let payload_len = 32 + 4 + inner_payload_len + (4 - inner_payload_len % 4) % 4;
// ...
payload[32 + 4..32 + 4 + inner_payload_len].copy_from_slice(&self.payload);
// ...
&payload[..payload_len]

```

v0.0.16 decoder enforces a minimum total payload length of 40 bytes:

RUST 393-398

```rust
// rs-stellar-strkey-0.0.16/src/ed25519.rs
const MIN_LENGTH: usize = 32 + 4 + 4;
// ...

```

```rust
397 if !(MIN_LENGTH..=MAX_LENGTH).contains(&payload_len) {
398     return Err(DecodeError::Invalid);
    }
```

CLI `zero` constructs an empty signed payload:

RUST 49–53

```rust
49 // rs-stellar-strkey-0.0.16/src/cli/zero.rs
50 Strkey::SignedPayloadEd25519(ed25519::SignedPayload {
51     ed25519: [0; 32],
52     payload: Default::default(),
53 })
```

v0.0.13 contains the same encode/decode mismatch for `inner_payload_len == 0` (the decoder enforces `MIN_LENGTH = 40`) and can similarly produce non-roundtrippable signed-payload StrKeys. v0.0.13 `SignedPayload::to_string()` can emit a 36-byte encoding when `inner_payload_len == 0`:

RUST 243–251

```rust
243 // rs-stellar-strkey-0.0.13/src/ed25519.rs
244 pub fn to_string(&self) -> String {
245     let inner_payload_len = self.payload.len();
246     let payload_len = 32 + 4 + inner_payload_len + (4 - inner_payload_len % 4) % 4;
        // ...
247     let mut payload = vec![0; payload_len];
248     // ...
249     encode(version::SIGNED_PAYLOAD_ED25519, &payload)
250 }
251
```

v0.0.13 `SignedPayload::from_payload()` rejects payloads shorter than 40 bytes:

RUST 265–273

```rust
265 // rs-stellar-strkey-0.0.13/src/ed25519.rs
266 pub fn from_payload(payload: &[u8]) -> Result<Self, DecodeError> {
        const MIN_LENGTH: usize = 32 + 4 + 4;
267     // ...
268     if !(MIN_LENGTH..=MAX_LENGTH).contains(&payload_len) {
269         return Err(DecodeError::Invalid);
270     }
271     // ...
272 }
273
```

Impact

Non-roundtrippable behavior (encode→decode mismatch) can break developer tooling and integration pipelines, and causes the CLI `zero signed_payload_ed25519` output to be misleading because the produced `StrKey` is not considered decodable or valid under this crate's own decoding rules.

PROOF OF CONCEPT

Execution

BASH

```
set -euo pipefail; SP=$(cargo run --quiet --features cli --bin stellar-strkey -- zero
signed_payload_ed25519); echo "zero output: $SP"; echo "decode:"; cargo run --quiet --
features cli --bin stellar-strkey -- decode "$SP";
```

Result

```
~/Doc/p/st/stellar-d/rs-stellar-strkey-0.0.16 ❯ set -euo pipefail; SP=$(cargo run --quiet --features cli --bin s
tellar-strkey -- zero signed_payload_ed25519); echo "zero output: $SP"; echo "decode:"; cargo run --quiet --feat
ures cli --bin stellar-strkey -- decode "$SP";
zero output: PAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAED2A
decode:
error: decoding "PAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAED2A": the strkey is invalid
[Process completed]
```

Analysis

This CLI sequence demonstrates a concrete encode→decode mismatch:

- `zero` produced: `PAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAED2A`
- `decode` then fails with: `the strkey is invalid`

RECOMMENDATION

To mitigate this issue, we recommend the following improvement:

- Align encoder and decoder invariants for signed payloads:
  - If the spec requires a minimum inner payload length (e.g., 4..=64), enforce it when constructing/encoding `SignedPayload` (and update `zero` to generate a minimally valid payload rather than empty), or
  - If zero-length inner payloads are allowed by the spec, update `from_payload()` to accept the 36-byte layout `inner_payload_len == 0`) and validate accordingly.

REMEDIATION COMMENT

Addressed in: 4b897e218b8f9a43cff0831d761d1e150e7cfb59

SOLVED: The **Stellar team** solved this finding in the specified commit by making `SignedPayload`'s fields private and introducing a validated `new()` constructor that enforces a minimum inner payload length of 1 byte, making the empty-payload state unrepresentable and routing all

construction paths (`from_payload`, serde deserialization, CLI zero) through this single enforcement point.

HAL-010 SOLVED

Decoded Strkey JSON Deserialization Produces Misleading Errors When Multiple Keys Are Present

● Informational · AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

DESCRIPTION

In **rs-stellar-strkey v0.0.16**, when `serde-decoded` is enabled, `Decoded<Strkey>` is expected to receive a JSON object containing exactly one variant key (for example, `{ "public_key_ed25519": "<hex>" }`). The deserializer's `visit_map` implementation, however, reads only the first key/value pair and returns without explicitly consuming or rejecting any remaining keys in the object.

In practice (for example, via the CLI `encode` command, which deserializes `Decoded<Strkey>` from a user-supplied JSON argument), supplying a JSON object with additional keys can produce misleading `serde_json` errors that suggest invalid JSON (such as "trailing comma"), even though the JSON is syntactically correct. This behavior obscures the actual requirement (that the object contain "exactly one variant key") and makes troubleshooting more difficult.

Code Location

In v0.0.16, the `Decoded<Strkey>` visitor reads a single map entry and returns `Ok` without verifying or rejecting any further keys:

RUST 182–198

```rust
// rs-stellar-strkey-0.0.16/src/strkey.rs
fn visit_map<M: MapAccess<'de>>(self, mut map: M) -> Result<Self::Value, M::Error> {
    let key: &str = map
        .next_key()?
        .ok_or_else(|| de::Error::custom("expected a variant key"))?;

    let strkey = match key {
        "public_key_ed25519" => {
            let Decoded(inner) = map.next_value()?;
            Strkey::PublicKeyEd25519(inner)
        }
        // ... other variants ...
        _ => return Err(de::Error::unknown_variant(key, &[/* ... */])),
    };

    Ok(Decoded(strkey))
}
```

In v0.0.16, the CLI `encode` command deserializes user-provided JSON into `Decoded<Strkey>` as shown:

RUST 30–33

```rust
// rs-stellar-strkey-0.0.16/src/cli/encode.rs
let Decoded(strkey): Decoded<Strkey> =
```

```
33 |     serde_json::from_str(&self.json).map_err(Error::Json)?;
   println!("{strkey}");
```

Impact

Confusing error messages and a brittle user experience are produced when decoded JSON is consumed; detection of extra keys is made more difficult and real JSON syntax errors are obscured. This is particularly problematic for CLI workflows and for downstream tooling that may augment JSON objects.

PROOF OF CONCEPT

Execution

BASH

```
cargo run --quiet --features cli --bin stellar-strkey -- encode
'{"public_key_ed25519":"0000000000000000000000000000000000000000000000000000000000000000","ha
sh_x":"0000000000000000000000000000000000000000000000000000000000000000"}'
```

Result

```
~/Doc/p/st/stellar-d/rs-stellar-strkey-0.0.16 ❯ cargo run --quiet --features cli --bin stellar-strkey -- encode
'{"public_key_ed25519":"0000000000000000000000000000000000000000000000000000000000000000","hash_x":"000000000000
0000000000000000000000000000000000000000000000000000000000000000"}'
error: trailing comma at line 1 column 89%                                                                    ~/
```

Analysis

The JSON is syntactically valid, but `visit_map` consumes only the first key/value and returns early; the remaining tokens then get surfaced by `serde_json` as a generic “trailing comma/trailing characters” parse-style error, obscuring the real issue (“object must contain exactly one variant key”).

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Enforce the “single variant key” invariant explicitly during deserialization:
  - after reading the first entry, attempt to read a second key and return a clear error if present (e.g., “expected exactly one variant key”).
- Alternatively, consume and ignore unknown keys deliberately (if you want permissive behavior), but do so explicitly and document the behavior.

REMEDIATION COMMENT

Addressed in: e7eb20b95ea08672fa697289e8ba0041faef103a

SOLVED: The **Stellar team** solved this finding in the specified commit by adding a post-match `next_key` check that explicitly rejects any additional keys and returns a clear *'expected exactly one variant key'* error.

Disclaimer

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.