Crate-git-revision

Stellar

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

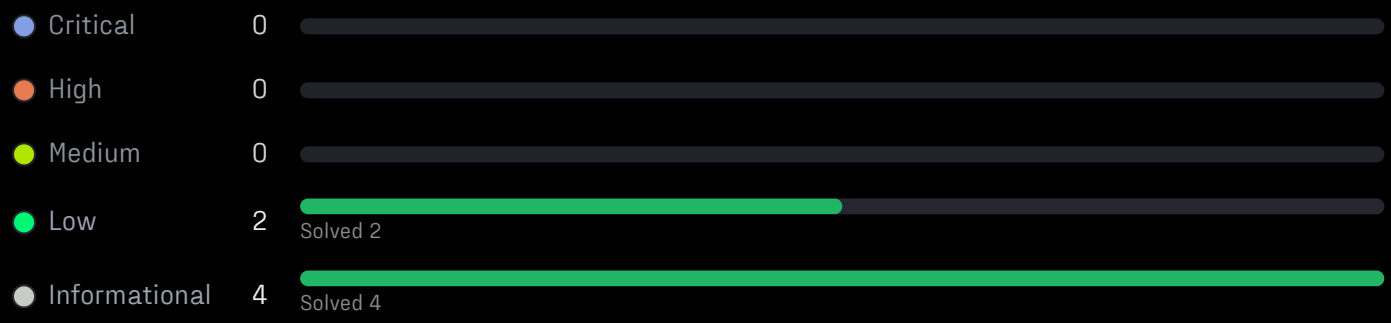
DATE OF ENGAGEMENT
Feb 9, 2026 - Mar 7, 2026

SOURCE CODE
crate-git-revision

ASSESSED COMMIT ID
38883b4

SEVERITY BREAKDOWN

6	6	0	0
All Findings	Solved	Risk Accepted	Acknowledged



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 build-script helper crate in the crate-git-revision repository—corresponding to the v0.0.6 release (commit 38883b4), from February 9th, 2026, to March 2nd, 2026.

ASSESSMENT SUMMARY

The team at Halborn assigned a full-time security engineer to verify the security of the Rust build-script helper 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:

- Assess build-script trust boundaries and directive injection risks when emitting Cargo build-script directives (e.g., `cargo:rustc-env=...`) and embedding `GIT_REVISION` into

downstream builds.

- Evaluate reliability and robustness failure modes across CI/CD, sandboxed, and non-git environments to prevent unexpected build breaks or missing build metadata.
- Review correctness and operational considerations (output normalization, reproducibility/hermetic build implications, and guidance on external exposure of revision metadata).

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

- `Always set a GIT_REVISION value and never silently discard errors; emit cargo:warning diagnostics when revision discovery fails to avoid downstream env!("GIT_REVISION") build failures.`
- `Normalize and validate all external strings before writing any cargo: directive: check git subprocess exit status, trim output, enforce an allowlist format, and reject/strip control characters to prevent directive injection and malformed metadata.`
- `Harden git subprocess execution in build scripts by sanitizing the inherited environment to reduce provenance spoofing and CI contamination.`
- `Avoid panics in build scripts by replacing current_dir().unwrap() with CARGO_MANIFEST_DIR (or graceful error handling) so environmental issues degrade safely instead of aborting builds.`

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 the crate's intended usage patterns (as a `build-dependency` invoked from `build.rs`) and how downstream consumers typically read `GIT_REVISION` via `env!` / `option_env!`.
- Perform manual, line-by-line review of `crate-git-revision-0.0.6/src/lib.rs` and supporting tests to map untrusted inputs (files, inherited environment, subprocess output) to build-script outputs.
- Analyze the build-time threat model for Cargo's build-script stdout protocol, prioritizing directive injection, provenance integrity, and unintended side effects during compilation.

- Evaluate correctness and reliability in common and non-standard environments (published crate metadata via `.cargo_vcs_info.json`, git checkouts, Windows CRLF handling, missing `git`, and sandboxed CI runners).
- Consolidate and prioritize recommendations by impact to build availability, traceability/provenance, and defense-in-depth hardening for CI/CD workflows.

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: crate-git-revision
(b) Assessed Commit ID: 38883b4
(c) Items in scope:

<> lib.rs Rust

REMEDIATION COMMIT ID:

7b29fabf6494fb4cec87f557211945515771bb24
87047f32585896472d347f59f34527aabed2898c
a3437f3da84b6f88747b52d7ff45df3252ad0a06
96f837802262663b8adad4b6b299b5c6e1a5bf1b
a90c7ea80f28659b691319e0a580f586589dd872
8c69d77337d6b96c25f890cedc7e14c771a908eb

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

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Git Subprocess Inherits Environment Allowing Repository Redirection During Build	Low	2.5	SOLVED 05/10/2026
HAL-002	Build-Script Directives Consume External Strings Without Normalization	Low	1.9	SOLVED 05/10/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-003	Improper Handling of GIT_REVISION Leads to Downstream Compile-Time Build Failures	Informational	—	SOLVED 05/25/2026
HAL-004	Unchecked current_dir() in init() Panics and Aborts the Build Permanently	Informational	—	SOLVED 05/25/2026
HAL-005	Incorrect Quoting of git describe Exclude Argument Produces Unexpected Revision Formats	Informational	—	SOLVED 05/01/2026
HAL-006	Non-Hermetic Dependency on git and Working Tree May Break Reproducible Build Requirements	Informational	—	SOLVED 05/25/2026

Findings & Tech Details

HAL-001 SOLVED

Git Subprocess Inherits Environment Allowing Repository Redirection During Build

Low · 2.5 AV:L/AC:H/PR:L/UI:N/S:U/C:N/I:L/A:N

DESCRIPTION

`crate_git_revision::__init()` shells out to `git` using `Command::new("git")` while inheriting the parent process environment, allowing Git-specific environment variables (e.g., `GIT_DIR`, `GIT_WORK_TREE`, `GIT_INDEX_FILE`, `GIT_OBJECT_DIRECTORY`) to redirect repository discovery and operations away from `current_dir`.

An attacker with the ability to influence the build environment (e.g., CI misconfiguration, contaminated job environment from earlier steps, or a malicious build script executed earlier in the build graph) can set `GIT_DIR` / `GIT_WORK_TREE` to point at a different repository so that `git rev-parse --git-dir` and `git describe ...` return values from the attacker-chosen repository. This can cause the build output to embed an incorrect `GIT_REVISION` value and emit `cargo:rerun-if-changed` directives pointing at attacker-chosen paths.

Separately, some Git environment variables are not just “repo selection” knobs. For example, `GIT_EXEC_PATH` influences where Git looks for `git-<subcommand>` helper programs, and `GIT_ASKPASS` can point Git at a program to run when it needs to prompt for credentials. While the specific invocations used here `git rev-parse` / `git describe`) are typically implemented as built-in commands and should not require credential prompting under normal circumstances, inheriting these variables is still an avoidable build-host hardening gap and increases the chance of surprising behavior across Git versions/platforms or if this crate later invokes additional Git commands.

Code Location

`crate-git-revision-0.0.6/src/lib.rs` (git invocations inherit environment):

RUST 99-123

```rust
match Command::new("git")
    .current_dir(current_dir)
    .arg("rev-parse")
    .arg("--git-dir")
    .output()
    .map(|o| o.stdout)
{
    // ...
    Ok(git_dir) => {
        // ...
        match Command::new("git")
            .current_dir(current_dir)
            .arg("describe")
            .arg("--always")
```

```
                .arg("--exclude='*'")
                .arg("--long")
                .arg("--abbrev=1000")
                .arg("--dirty")
                .output()
                .map(|o| o.stdout)
        {
            // ...
        }
    }
}
```

Impact

An attacker with build-environment control can redirect git operations via `GIT_DIRGIT_WORK_TREE`, embedding an incorrect `GIT_REVISION` without modifying system binaries — a lighter forensic footprint than PATH hijacking. Any attacker at this privilege level can achieve equivalent impact via PATH substitution; this fix is a defense-in-depth hardening measure.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- **Explicitly remove Git control environment variables** for all `git` subprocess invocations, including at minimum: `GIT_DIR`, `GIT_WORK_TREE`, `GIT_INDEX_FILE`, `GIT_OBJECT_DIRECTORY`, `GIT_ALTERNATE_OBJECT_DIRECTORIES`, `GIT_CONFIG_GLOBAL`, `GIT_CONFIG_SYSTEM`, `GIT_EXEC_PATH`, `GIT_HOOKS_PATH`, `GIT_NAMESPACE`, and `GIT_ASKPASS`.
- **Prevent interactive hangs in CI**: explicitly set `GIT_TERMINAL_PROMPT=0` for all invocations (rather than relying on whatever the parent environment provides).
- **Prefer an allowlist model**: call `.env_clear()` and then set only the minimal required environment (including explicitly setting `GIT_CONFIG_NOSYSTEM=1` and `GIT_TERMINAL_PROMPT=0`) if compatibility with build environments permits.

REMEDIATION COMMENT

Addressed in: 7b29fabf6494fb4cec87f557211945515771bb24

SOLVED: The **Stellar team** solved this finding in the specified commit by introducing a `git()` helper that strips the five primary repository-redirect variables (`GIT_DIR`, `GIT_WORK_TREE`, `GIT_INDEX_FILE`, `GIT_OBJECT_DIRECTORY`, `GIT_ALTERNATE_OBJECT_DIRECTORIES`) and sets `GIT_TERMINAL_PROMPT=0`, applying it consistently to all `git` subprocess invocations in `__init()`. The narrower scope is a deliberate risk trade-off aligned with established toolchain precedent (cargo) and directly addresses the finding's stated attack surface.

HAL-002 SOLVED

Build-Script Directives Consume External Strings Without Normalization

Low · 1.9 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

`__init()` emits Cargo build-script directives that incorporate values derived from the following sources:

- a file: `.cargo_vcs_info.json` (JSON field `git.sha1`)
- subprocess stdout: `git rev-parse --git-dir`
- subprocess stdout: `git describe ...`

Under normal operation, the following assumptions are made:

- `.cargo_vcs_info.json` is produced by Cargo and contains a hexadecimal commit identifier.
- `git rev-parse --git-dir` produces a single-line path.
- `git describe` typically returns a single-line description ending with a trailing newline (and this output is currently not trimmed before embedding).

The implementation currently does not:

- validate the format of the `git.sha1` value in `.cargo_vcs_info.json`.
- verify the subprocess exit status before using stdout (for example, by checking `output.status.success()`).
- trim the `git describe` output prior to embedding it.
- enforce a single-line, control-character–free policy before writing `cargo:` directive lines.

Because subprocess exit status is ignored, a non-zero `git` execution can still yield `Ok(stdout)` and be consumed silently (including empty/partial output), which makes failures harder to detect and can lead to unreliable metadata.

Code Location

Reads Cargo's embedded VCS metadata (if present) and stores the JSON `git.sha1` string into `git_sha`:

RUST 57–62

```rust
57  if let Ok(vcs_info) = read_to_string(current_dir.join(".cargo_vcs_info.json")) {
        let vcs_info: Result<CargoVcsInfo, _> = serde_json::from_str(&vcs_info);
58      if let Ok(vcs_info) = vcs_info {
            git_sha = Some(vcs_info.git.sha1);
59
60
61
```

```
    }
}
```

Executes `git` to discover the repo's git directory, then uses that path to print `cargo:rerun-if-changed=...` directives for key git state files:

RUST 67-97

```rust
match Command::new("git")
    .current_dir(current_dir)
    .arg("rev-parse")
    .arg("--git-dir")
    .output()
    .map(|o| o.stdout)
{
    Err(e) => {
        writeln!(
            w,
            "cargo:warning=Error getting git directory to get git revision: {e:?}"
        )?;
    }
    Ok(git_dir) => {
        let git_dir = String::from_utf8_lossy(&git_dir);
        let git_dir = git_dir.trim();

        // Require the build script to rerun if relavent git state changes which
        // changes the current git commit.
        //  - .git/index: Changes if the index/staged files changes, which will
        //  cause the repo to be dirty.
        //  - .git/HEAD: Changes if the ref currently in the working directory,
        //  and potentially the commit, to change.
        //  - .git/refs: Changes to any files in refs could cause the current
        //  commit to have changed if the ref in .git/HEAD is changed.
        // Note: That changes in the above files may not result in material
        // changes to the crate, but changes in any should invalidate the
        // revision since the revision can be changed by any of the above.
        writeln!(w, "cargo:rerun-if-changed={git_dir}/index")?;
        writeln!(w, "cargo:rerun-if-changed={git_dir}/HEAD")?;
        writeln!(w, "cargo:rerun-if-changed={git_dir}/refs")?;

```

Executes `git describe` and assigns its stdout (currently untrimmed) to `git_sha` as the revision string:

RUST 99-119

```rust
match Command::new("git")
    .current_dir(current_dir)
    .arg("describe")
    .arg("--always")
    .arg("--exclude='*'")
    .arg("--long")
    .arg("--abbrev=1000")
    .arg("--dirty")
    .output()
    .map(|o| o.stdout)
{
    Err(e) => {
        writeln!(
            w,
            "cargo:warning=Error getting git revision from {current_dir:?}: {e:?}"
        )?;
```

```
    }
    Ok(git_describe) => {
        git_sha = str::from_utf8(&git_describe).ok().map(str::to_string);
    }
}
```

If `git_sha` is set, prints `cargo:rustc-env=GIT_REVISION=...` so downstream compilation can read it via `env!option_env!`:

RUST 124–126

```
if let Some(git_sha) = git_sha {
    writeln!(w, "cargo:rustc-env=GIT_REVISION={git_sha}")?;
}
```

Impact

Under normal conditions the risk is negligible. Two concrete gaps remain: unchecked subprocess exit status can silently embed partial output as `GIT_REVISION`; unvalidated `git describe` output containing newlines could inject arbitrary `cargo:` directives into build script output.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- **Validate** `.cargo_vcs_info.json` `sha1` against `^[0-9a-f]{40}$` before using it.
- **Check subprocess status** and only use stdout when `output.status.success()` is true; otherwise emit `cargo:warning` and skip.
- **Trim and normalize**: apply `.trim()` to `git describe` output before embedding.
- **Control-character defense-in-depth**: reject/strip `\r`, `\n`, and other control characters from values before writing any `cargo:` line.

REMEDIATION COMMENT

Addressed in: 87047f32585896472d347f59f34527aabed2898c

SOLVED: The **Stellar team** solved this finding across commits a90c7ea, d946d5b, 87047f3, and 4aa9f27 by replacing `git describe` with `git rev-parse HEAD` (eliminating newline injection into `cargo:` directives), applying explicit `.trim()` to the SHA output, adding exit-code checking for `git status --porcelain`, and making a deliberate business decision to use the `.cargo_vcs_info.json` sha1 field verbatim rather than format-validating a Cargo-generated file.

HAL-003 SOLVED

Improper Handling of GIT_REVISION Leads to Downstream Compile-Time Build Failures

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

DESCRIPTION

`__init()` sets `GIT_REVISION` only when a revision string is successfully obtained. The `Result` returned by `__init()` is discarded within the `init` function (using `let _res = ...`).
As a result, under common conditions (non-git checkout, missing `git` binary, restricted execution environment, invalid UTF-8 output, or parse errors), the build script may emit no `cargo:rustc-env=GIT_REVISION=...` line.

Code Location

The `init` function ignores the result returned by `__init`:

RUST 48-50

```rust
pub fn init() {
  let _res = __init(&mut std::io::stdout(), &std::env::current_dir().unwrap());
}
```

Code snippet from the `__init` function:

RUST 124-128

```rust
if let Some(git_sha) = git_sha {
    writeln!(w, "cargo:rustc-env=GIT_REVISION={git_sha}")?;
}

Ok(())
```

Impact

Downstream crates using `env!("GIT_REVISION")` fail to compile with no helpful diagnostic when built outside a git environment. This is a reliability concern for vendored or release builds.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Consider always setting a value (e.g., `unknown`) and emitting `cargo:warning` when it cannot be derived.
- Alternatively, document strongly that downstream code should use `option_env!("GIT_REVISION")` with a fallback.

- Do not discard `__init()` errors silently; warn on failure.

REMEDIATION COMMENT

Addressed in: a3437f3da84b6f88747b52d7ff45df3252ad0a06

SOLVED: The **Stellar team** solved this finding by documenting that downstream code should use `option_env!("GIT_REVISION")` with a fallback, and by emitting warnings on failures inside `init`.

HAL-004 SOLVED

Unchecked current_dir() in init() Panics and Aborts the Build Permanently

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

DESCRIPTION

`init()` in `crate-git-revision-0.0.6/src/lib.rs` calls `std::env::current_dir().unwrap()` with no error handling, converting any `Err` return directly into a panic that kills the build script process.

No active attacker is required to trigger this. Any build environment where the working directory is inaccessible at the time `init()` runs — a sandboxed runner with restricted filesystem access, a container where the working directory is unmounted between setup and build phases, or an ephemeral CI job where the directory is deleted mid-execution — will cause `current_dir()` to return `Err`. The `.unwrap()` call then panics unconditionally, aborting the build immediately. Because `GIT_REVISION` is never emitted in this path, any downstream crate using `env!("GIT_REVISION")` will also hard-fail to compile on the next invocation, compounding the outage.

Code Location

`init()` — unchecked `current_dir()` call that panics on failure:

RUST 48-50

```rust
48  pub fn init() {
49      let _res = __init(&mut std::io::stdout(), &std::env::current_dir().unwrap());
50  }
```

Impact

An inaccessible working directory causes an unconditional panic instead of a graceful `cargo:warning`, producing a misleading backtrace. The triggering condition is exceptional and requires build-environment control — the same precondition as far more impactful attacks.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- **Replace `current_dir()` with `CARGO_MANIFEST_DIR`**: Cargo unconditionally injects `CARGO_MANIFEST_DIR` as an environment variable pointing to the crate's manifest directory in every build script invocation. This is the semantically correct source of the crate directory in a build script context and is immune to working-directory mutations.
- **Never panic in a build script**: emit `cargo:warning` and return gracefully on any unrecoverable condition so that the build degrades without aborting.

REMEDIATION COMMENT

Addressed in: 96f837802262663b8adad4b6b299b5c6e1a5bf1b

SOLVED: The **Stellar team** solved this finding by replacing `current_dir().unwrap()` in `init()` with `env::var_os("CARGO_MANIFEST_DIR")` and by using `expect` instead of `unwrap` to surface a clear message at the panic site.

HAL-005 SOLVED

Incorrect Quoting of git describe Exclude Argument Produces Unexpected Revision Formats

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

DESCRIPTION

The code attempts to instruct Git to "ignore all tags" by providing an `--exclude` pattern. Because the command is executed directly (not via a shell), the surrounding quote characters are not stripped. Consequently, Git receives the pattern including the literal quotes, which typically does not match any tags.

Code Location

Code snippet from the `__init` function:

RUST 99-107

```rust
match Command::new("git")
  .current_dir(current_dir)
  .arg("describe")
  .arg("--always")
  .arg("--exclude='*'")
  .arg("--long")
  .arg("--abbrev=1000")
  .arg("--dirty")
  .output()
```

Impact

Tags are not reliably excluded, so `git describe` may include tag names in the generated revision string (for example, `v1.2.3-0-g<hash>` rather than just the commit hash). As a result, the value embedded into `GIT_REVISION` can vary, causing inconsistent version strings across repositories and environments.

PROOF OF CONCEPT

Code

The PoC code has been added into existing test suite located in the `crate-git-revision-0.0.6/src/test.rs` file:

RUST

```rust
#[test]
fn poc_git_describe_exclude_quoting_behavior() {
    let tempdir = tempfile::tempdir().unwrap();
    let git_dir = tempdir.path();

    init_git_repo(git_dir);

    // Create an *annotated* tag at HEAD so that `git describe` would normally
    // prefer it (by default, `git describe` prefers annotated tags).
    let output = Command::new("git")
```

```
        .current_dir(git_dir)
        .arg("tag")
        .arg("-a")
        .arg("v0.1.0")
        .arg("-m")
        .arg("test tag")
        .output()
        .unwrap();
    assert!(output.status.success());

    // Incorrect quoting: quotes are passed literally to git, so the exclude glob
    // is unlikely to match and tags are *not* excluded.
    let out_bad = Command::new("git")
        .current_dir(git_dir)
        .arg("describe")
        .arg("--always")
        .arg("--exclude='*'")
        .arg("--long")
        .arg("--abbrev=1000")
        .output()
        .unwrap();
    assert!(out_bad.status.success());
    let out_bad = str::from_utf8(&out_bad.stdout).unwrap().trim().to_string();

    // Correct glob argument: excludes all tags, so `git describe` falls back to a
    // hash-based description (because --always is set).
    let out_good = Command::new("git")
        .current_dir(git_dir)
        .arg("describe")
        .arg("--always")
        .arg("--exclude=*")
        .arg("--long")
        .arg("--abbrev=1000")
        .output()
        .unwrap();
    assert!(out_good.status.success());
    let out_good = str::from_utf8(&out_good.stdout).unwrap().trim().to_string();

    // With the broken quoting, the output should start with the tag name.
    assert!(
        out_bad.starts_with("v0.1.0-"),
        "expected broken exclude to allow tag-based describe; got: {out_bad:?}"
    );

    // With the correct exclude, the output should not start with the tag and
    // should match a hash-based describe format. Depending on git version and
    // whether any tags are reachable, `git describe --long --always` may return
    // either a plain hash or `<hash>-0-g<hash>`.
    assert!(
        !out_good.starts_with("v0.1.0-"),
        "expected correct exclude to avoid tag-based describe; got: {out_good:?}"
    );
    assert!(
        Regex::new(r"^[0-9a-f]{7,40}(-0-g[0-9a-f]{7,40})?$")
            .unwrap()
            .is_match(&out_good),
        "expected hash-based describe output; got: {out_good:?}"
    );
}
```

Execution

```sh
```

```
cargo test --package crate-git-revision --lib --
test::poc_git_describe_exclude_quoting_behavior --exact
```

Result

```
running 1 test
```

Analysis

This PoC confirms that when `git describe` is invoked without a shell, passing `--exclude='*'` preserves the quotes literally, so tags are not excluded and the revision can become tag-based (e.g., `v0.1.0-…`) instead of a hash-only format.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Replace `.arg("--exclude='*'")` with `.arg("--exclude=*")` (or remove the option if tag-based output is acceptable).
- Trim stdout from `git describe` before embedding, to avoid carrying a trailing newline into `GIT_REVISION`.

REMEDIATION COMMENT

Addressed in: a90c7ea80f28659b691319e0a580f586589dd872

SOLVED: The **Stellar team** solved this finding in the specified commit by replacing `.arg("--exclude='*'")` with `.arg("--exclude=*")`, correctly passing the bare glob to git and preventing tag names from appearing in the generated revision string.

HAL-006 SOLVED

Non-Hermetic Dependency on git and Working Tree May Break Reproducible Build Requirements

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

DESCRIPTION

When `.cargo_vcs_info.json` is absent, the crate shells out to `git` and includes dirty state. This makes outputs depend on:

- presence and behavior of external `git`
- repository metadata and checkout style
- working tree modifications during build

For teams with reproducible build requirements, this can violate policy unless explicitly controlled.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Document expectations for non-git builds and CI environments.
- Provide a mode to disable dirty suffix and/or disable calling out to `git` (use only published metadata), if reproducibility is a goal.

REMEDIATION COMMENT

Addressed in: 8c69d77337d6b96c25f890cedc7e14c771a908eb

SOLVED: The **Stellar team** solved this finding by expanding the crate-level documentation with sections covering builds without version info, shallow clone support, reproducible-build expectations around clean working trees, the stripping of path-redirecting `GIT_*` environment variables, and the `GIT_TERMINAL_PROMPT=0` behavior. They also decided not to add a mode that disables Git checks and relies solely on published metadata, as this would introduce support for self-modifying workspaces during the build process, which is outside the library's intended scope.

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.