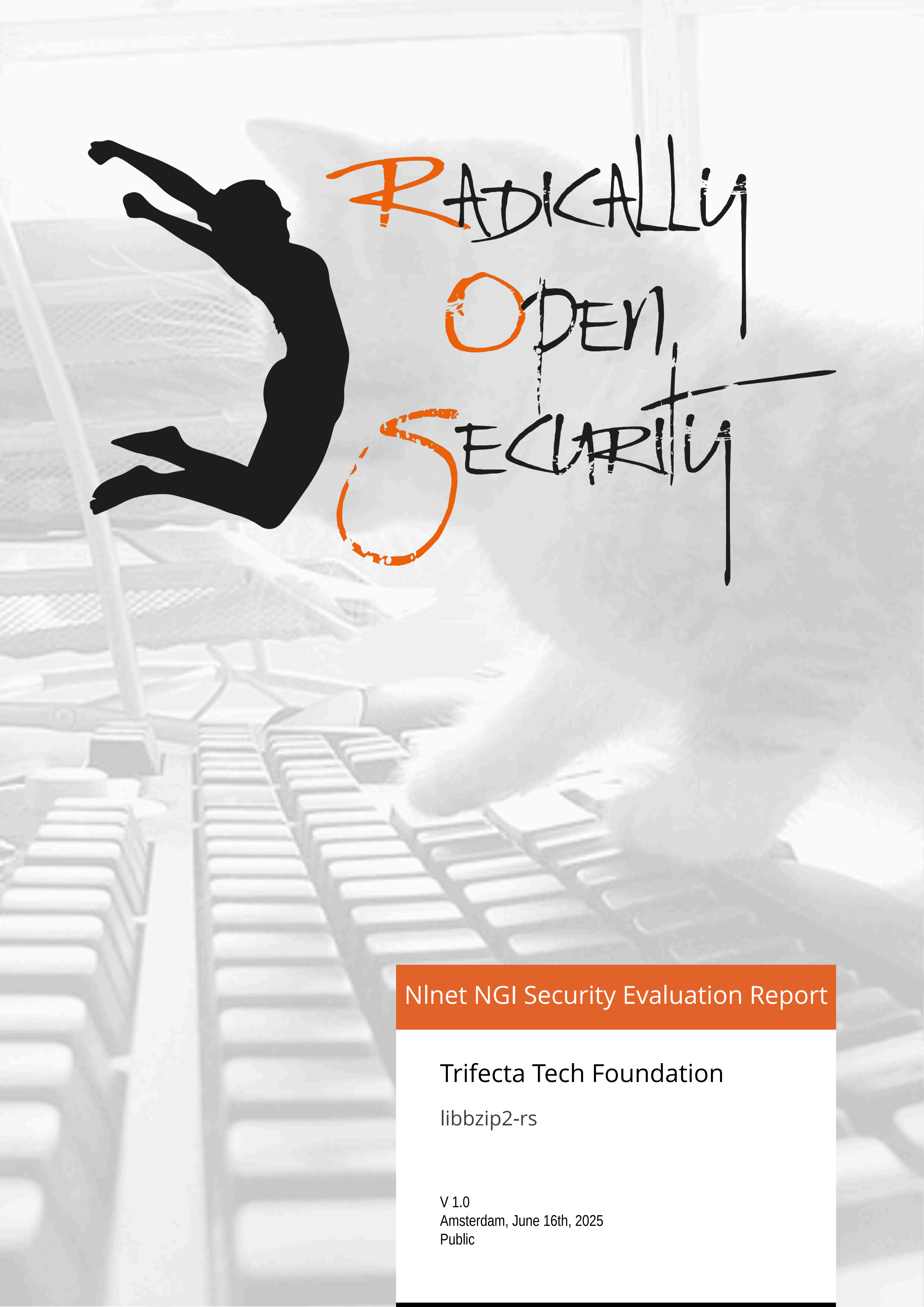

Radically Open Security

NInet NGI Security Evaluation Report

Trifecta Tech Foundation

libbzip2-rs

V 1.0
Amsterdam, June 16th, 2025
Public

Document Properties

Client	Trifecta Tech Foundation
Title	NLnet NGI Security Evaluation Report
Targets	• Fuzz testing of exposed attack surface • Evaluate and improve fuzz testing code and configurations • Assist the project in reproducing, fixing and reporting of discovered issues • Discuss potential improvements for documentation and testing • Prepare improved test code for adoption by the project
Version	1.0
Pentester	Christian Reitter
Authors	Christian Reitter, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	June 13th, 2025	Christian Reitter	Initial draft
0.2	June 16th, 2025	Marcus Bointon	Review
1.0	June 16th, 2025	Marcus Bointon	1.0

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary 4

1.1 Introduction 4

1.2 Scope of Work 4

1.3 Project Objectives 4

1.4 Timeline 5

1.5 Results In A Nutshell 5

2 Methodology 6

2.1 Planning 6

2.2 Risk Classification 6

3 Reconnaissance and Fingerprinting 8

4 Findings 9

5 Non-Findings 10

5.1 NF-001 — Security scans without findings 10

5.2 NF-002 — Fuzz harness bug in decompress_chunked.rs:32 10

5.3 NF-003 — Potential fuzzer improvement: use a fuzz dictionary 11

5.4 NF-004 — Fuzz harness bug in decompress_chunked.rs:100 11

5.5 NF-005 — Fuzz harness bug in compress.rs:33 12

5.6 NF-006 — Correctness bug in decompression 12

5.7 NF-007 — Incorrect BZ2_bzCompressInit parameter handling 13

5.8 NF-008 — Potential fuzzer improvement: avoid Corpus::Reject 14

5.9 NF-009 — Fuzz harness bug in decompress.rs 16

6 Future Work 17

7 Conclusion 18

Appendix 1 Testing Team 19

1 Executive Summary

1.1 Introduction

Between May 27, 2025 and June 11, 2025, Radically Open Security B.V. carried out a penetration test for Trifecta Tech Foundation and NLnet NGI Zero Core as part of the Zip linting and bzip2 in Rust project grant.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of Work

The scope of the penetration test was limited to the following targets:

- Fuzz testing of exposed attack surface
- Evaluate and improve fuzz testing code and configurations
- Assist the project in reproducing, fixing and reporting of discovered issues
- Discuss potential improvements for documentation and testing
- Prepare improved test code for adoption by the project

The scoped services are broken down as follows:

- Code audit and fuzz testing: 3 days
- Reporting: 8 hours
- Communication, shared sessions and other tasks: 8 hours
- **Total effort: 5 days**

1.3 Project Objectives

ROS will perform a limited code review of the libbzip2-rs library with Trifecta Tech Foundation in order to assess its security. To do so ROS will perform a code review on the newest public code revision of the Open Source project and guide Trifecta Tech Foundation in attempting to find vulnerabilities, exploiting and analyzing any such found to try and gain further understanding of the security impact. Where possible, ROS will assist with retesting of fixes and mitigations that become available during the engagement phase.

During project scoping, Trifecta Tech Foundation and ROS identified the evaluation and improvement of the automated fuzz testing components of `libbzip2-rs` as the primary goal of this review. Given the intended role of `libbzip2-rs` as a highly compatible drop-in replacement for existing `libbzip2` implementations, automated adversarial testing of `libbzip2-rs` robustness and differential testing are powerful mechanisms to increase trust in the new implementation as well as to spot regressions during future development. Due to the limited time available for review, other evaluations such as manual code analysis are only planned in selected areas, if indicated by other test results.

Please note that this report does not cover the "zip linting" aspect of the grant scope, since the Trifecta Tech Foundation determined that their work in this area is not yet ready for review. Instead, they asked us to separate and postpone the relevant review steps.

Initially reviewed libbzip2-rs version: commit `f5161491049dfa7dacbe8ac7f3614129be54b0be` on the `main` branch. This was the newest available code revision at the beginning of the review, and represents the code state shortly after the `v0.2.0` release.

During the review phase, we switched the analyzed version multiple times to account for improvements made by the project developers, including PR#106, PR#107, PR#108, PR#109, PR#110, PR#111, and PR#112.

1.4 Timeline

The security audit took place between May 27, 2025 and June 11, 2025.

1.5 Results In A Nutshell

During this code audit project we found no direct security issues.

Leveraging the improved fuzz testing setups, we discovered a functional problem edge case in `libbzip2-rs` related to decompression, which was fixed by the project developers.

Additionally, we discovered several bugs and other limitations in the existing fuzz testing harnesses that limited their usability and effectiveness, which we fixed together with the `libbzip2-rs` developers.

2 Methodology

2.1 Planning

Our general approach during code audits is as follows:

During the code audit we verify if the proper security controls are present, work as intended and are implemented correctly. If vulnerabilities are found, we determine the threat level by assessing the likelihood of exploitation of this vulnerability and the impact on the Confidentiality, Integrity and Availability (CIA) of the system. We will describe how an attacker would exploit the vulnerability and suggest ways of fixing it.

This requires an extensive knowledge of the platform the application is running on, as well as the extensive knowledge of the language the application is written in and patterns that have been used. Therefore a code audit is done by specialists with a strong background in programming.

During a code audit, we take the following approach:

1. **Thorough comprehension of functionality**
   We try to get a thorough comprehension of how the application works and how it interacts with the user and other systems. Having detailed documentation at this stage is very helpful, as it aids the understanding of the application.
2. **Comprehensive code reading**
   Goals of the comprehensive code reading are:
   - to get an understanding of the whole code
   - identify adversary controlled inputs and trace their paths
   - identify issues
3. **Automated code scans**
   Using specialized tools, we scan the codebase for common issue patterns and then manually review the flagged code positions for potential negative security implications.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**
  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**
  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Reconnaissance and Fingerprinting

We were able to gain information about the software and infrastructure through the following automated scans. Any relevant scan output will be referred to in the findings.

- cargo-fuzz – https://github.com/rust-fuzz/cargo-fuzz
- cargo-audit – https://github.com/rustsec/rustsec
- semgrep – https://semgrep.dev
- semgrep ruleset – https://github.com/semgrep/semgrep-rules
- Trailofbits semgrep ruleset – https://github.com/trailofbits/semgrep-rules

4 Findings

We did not identify any security issues.

5 Non-Findings

In this section we list non-security issues as well as documentation on work tasks:

5.1 NF-001 — Security scans without findings

cargo audit

```
cargo audit
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
      Loaded 782 security advisories (from [...].cargo/advisory-db)
    Updating crates.io index
    Scanning Cargo.lock for vulnerabilities (28 crate dependencies)
```

We analyzed different `libbzip2-rs` sub-directories which had a `Cargo.lock` file, but did not discover any problematic results.

The project developers indicated that they deliberately use as few external Rust crates as possible to avoid dependencies on external software for release versions of their new libraries. We think this is an excellent goal for a compression library and will reduce both maintenance requirements as well as security concerns caused by other dependencies in the future.

Semgrep scan

We analyzed `libbzip2-rs` with publicly available semgrep rulesets for common code problem patterns. This did not lead to any findings.

5.2 NF-002 — Fuzz harness bug in decompress_chunked.rs:32

During evaluation, we discovered false positives in the decompress_chunked.rs harness.

Crash log:

```
Running: fuzz/artifacts/decompress_chunked/crash-3c018ba965f07cb65dc78a246524dd63148a1a25

thread '<unnamed>' panicked at fuzz_targets/decompress_chunked.rs:32:5:
assertion `left == right` failed
  left: 3
 right: 4
```

We first observed this exception on an external file, which is not a bzip2-related file but is relatively large and complex.

The logged symptoms suggest that the compression step via the `libbzip2` C implementation did not succeed as expected. However, we traced the underlying issue to the use of statically sized input buffers which do not allow processing of larger inputs. The project developers fixed this via pull request no. 106.

Since the problem was only present in the fuzz test code, this had no impact on the library itself.

5.3 NF-003 — Potential fuzzer improvement: use a fuzz dictionary

The `libFuzzer` engine supports the use of special dictionaries to suggest useful data snippets to the fuzzer for use during mutation steps. For some targets, this advanced feature can help the engine significantly in constructing interesting inputs more quickly and efficiently through mutation, especially if the input is based on some sort of formal grammar.

At the time of review `libbzip2-rs` did not use dictionaries. We found an existing fuzzer dictionary for bzip2 in a collection by Google, which could be useful for fuzz harnesses that take compressed data as inputs. However, due to the relative simplicity of the bzip2 binary format, this dictionary is limited to just three entries:

```
magic="BZ"
compress_magic="\x31\x41\x59\x26\x53\x59"
eos_magic="\x17\x72\x45\x38\x50\x90"
```

Additionally, we expect that these data snippets are already embedded in the seed corpus that `libbzip2-rs` uses, which makes them less valuable as dedicated dictionary entries.

Due to this, we suspect that the practical advantages of using the existing dictionary for future runs, for example, as part of the Continuous Integration (CI) workflows, only bring very limited benefits and may not be worth the maintenance effort.

We documented the existing bzip2 fuzzer dictionary in new fuzzer documentation, which was added to the project via pull request no. 109.

5.4 NF-004 — Fuzz harness bug in decompress_chunked.rs:100

During evaluation, we discovered false positives in the decompress_chunked.rs harness.

The assert at `assert_eq!(output, data);` in line 100 reported a mismatch between the expected `data` and the generated `output` after the compression and decompression round trip.

Notably, the output had a significant difference in length:

- `output.len()` is `32768`
- `data.len()` is `64980`

We first observed this issue by using the https://github.com/trifectatechfoundation/libbzip2-rs/tree/main/tests/input/quick folder as fuzzer inputs.

Further debugging revealed that this was a false positive caused by static buffer sizes, specifically `let mut output = vec![0u8; 1 << 15];` which only reserves a fixed `2^15` bytes of space for the output. The developers fixed this via pull request no. 107.

Since the problem was only present in the fuzz test code, this had no impact on the library itself.

5.5 NF-005 — Fuzz harness bug in compress.rs:33

During evaluation, we discovered false positives in the compress.rs fuzz harness.

The assert at `assert_eq!(error, BZ_OK);` in line 32 reported that the decompression failed, which should never happen since it operates on correctly compressed data from previous steps.

Crash log:

```
thread '<unnamed>' panicked at fuzz_targets/compress.rs:32:5:
assertion `left == right` failed
  left: -8
 right: 0
```

We found that the failures were related to the statically sized `let mut output = [0u8; 1 << 10];` buffer, which only holds `2^10 = 1024` bytes. Using a minimized version of the crashing input, the developers were able to reproduce the issue and fixed it via pull request no. 107. Since the problem was only present in the fuzz test code, this had no impact on the library itself.

5.6 NF-006 — Correctness bug in decompression

We discovered an interesting edge case during an in-depth evaluation of the decompress_random_inputs.rs fuzz test.

The `decompress_random_inputs.rs` test performs differential fuzz testing by running the same input through both the `libbzip2` (in C) and `libbzip2-rs` (in Rust) decode functions, which are supposed to behave identically. In the test code, the identical return code is checked via `assert_eq!(err_c, err_rs);`.

After several hours of fuzzing, we discovered an input that led to a difference in error codes between the two implementations:

```
thread '<unnamed>' panicked at fuzz_targets/decompress_random_input.rs:32:5:
assertion `left == right` failed
  left: -7
 right: -4
```

In this comparison, left is `err_c` with `BZ_UNEXPECTED_EOF = -7`, and right is `err_rs` with `BZ_DATA_ERROR = -4`.

We minimized this into a simple reproducer input, which we shared with the project developers.

Reproducer hexdump (35 bytes):

```
00000000  43 42 f7 5a ff 68 ff 32  41 31 a9 41 59 59 53 26  |CB.Z.h.2A1.AYYS&|
00000010  59 53 0b 59 35 03 59 4f  31 7e 01 01 01 01 a7 86  |YS.Y5.YO1~......|
00000020  a7 a5 a7                                          |...|
```

After ruling out other sources for false positives in this error code mismatch, the project developers were able to trace this behavior to an actual code bug in the `libbzip2-rs` library, which they corrected via pull request no.110.

This emphasizes the value of differential testing for spotting edge cases and non-compliant behavior. In this case, it is a major asset to have two separate implementations with identical APIs and specifications which can be compared at runtime.

5.7 NF-007 — Incorrect BZ2_bzCompressInit parameter handling

We discovered that the `libbzip2-rs` compression API has a mismatch between its code documentation and its actual behavior for the `verbosity` parameter.

The code documentation for `BZ2_bzCompressInit()` explicitly states that for `verbosity` values outside the range of 0 to 4, the compression will abort with the `BZ_PARAM_ERROR` error:

```
/// Prepares the stream for compression.
///
/// # Returns
///
/// - [`BZ_PARAM_ERROR`] if any of
///     - `strm.is_null()`
///     - `!(1..=9).contains(&blockSize100k)`
///     - `!(0..=4).contains(&verbosity)`
///     - `!(0..=250).contains(&workFactor)`
///     - no [valid allocator](bz_stream#custom-allocators) could be configured
[...]
```

However, the implementation in `BZ2_bzCompressInitHelp()` does not actually do that, see here. Instead, larger values for `verbosity` are passed into the state and used there.

Fortunately, we think this has no further negative impact, since verbosity values greater than `4` (maximum verbosity) are effectively treated identically to `4` by the existing code.

This behavior is inconsistent with both the code documentation, as well as with the behavior on the decompression API, which does enforce the value range correctly (see here).

The `libbzip2-rs` developers acknowledged this issue. During their investigation, they discovered that the original `libbzip2` C implementation, which `libbzip2-rs` is based on, also did not enforce the value, see here. The error in `libbzip2-rs` was likely copied over while translating the `libbzip2` implementation.

We recommend correcting either the code documentation or the implementation, but see this as a non-security issue.

5.8 NF-008 — Potential fuzzer improvement: avoid Corpus::Reject

One of the fuzz test definitions makes use of the `Corpus::Reject` mechanism, which is a special directive for the libFuzzer engine that forces it to respond differently to a given fuzz test input. See the libFuzzer documentation for more information.

The `libbzip2-rs` developers anticipated that their use of this mechanism was overall beneficial by helping to steer the fuzzer in the direction of more important inputs. However, we think that this mechanism is counterproductive to use in many cases, and only recommend enabling it selectively, such as for special corpus collection filter tasks.

For this reason, we proposed a code change as part of PR 109 which changes the use of this mechanism (via a custom cargo crate feature that activates it at build time) from enabled by default to disabled by default. The pull request was accepted.

Here is a synthetic example case which shows the counterproductive effects of this mechanism on the fuzzing success. Consider the following fuzz harness file, which we created purely for demonstration purposes and is not related to `bzip2`:

```
#![no_main]
use libfuzzer_sys::{fuzz_target, Corpus};

fuzz_target!(|fuzz_data: &[u8]| -> Corpus {
    let invalid_input = if cfg!(feature = "reject-invalid-in-corpus") {
        // instruct libFuzzer to reject and ignore this input
        Corpus::Reject
    } else {
        // normal neutral behavior
        Corpus::Keep
    };

    let secret = b"secret";

    // ensure we don't run out of bounds during the following accesses
    if fuzz_data.len() >= secret.len() {
        // give the fuzzer some incremental signal on the error path
        if fuzz_data[0] == secret[0] {
            if fuzz_data[1] == secret[1] {
                if fuzz_data[2] == secret[2] {
                    if fuzz_data[3] == secret[3] {
                        if fuzz_data[4] == secret[4] {
                            if fuzz_data[5] == secret[5] {
                                panic!("reached the simulated target");
                                return Corpus::Keep;
                            }
                        }
                    }
                }
            }
        }
    }
    invalid_input
});
```

Building this fuzz target with the `cargo` parameters `--features=reject-invalid-in-corpus` will activate the `Corpus::Reject` mechanism and use it to explicitly reject all inputs that don't satisfy a complex requirement (in this case, ones that don't match a special prefix). This approximates rejecting all inputs if they do not satisfy a complex file parser. In this configuration, the fuzz test forcefully ignores most code coverage gains that it can find within the error paths as well as the corresponding inputs. In effect, it prevents the fuzz engine from effectively evolving its fuzz inputs any further due to a lack of signal for "interesting" inputs and by being forbidden from collecting imperfect inputs in the corpus to make them available for future runs.

When started on an empty corpus, which is the initial default situation, this fuzzer fails to make any progress in finding the prepared `panic!()` bug for many minutes of runtime.

The log output is sparse and doesn't show any notable progress:

```
#2  INITED exec/s: 0 rss: 30Mb
WARNING: no interesting inputs were found so far. Is the code instrumented for coverage?
This may also happen if the target rejected all inputs we tried so far
```

In comparison, a fuzz run with the normal configuration without `Corpus::Reject` will find and trigger the target code branch in less than a second. Iterative steps and coverage signal from the coverage counters help it to individually reach each of the nested `if` branches. This is reflected in the log, which shows the corresponding `cov: ` counter increase as the deeper code branches are discovered and logged:

```
INFO: A corpus is not provided, starting from an empty corpus
2   INITED cov: 8 ft: 9 corp: 1/1b exec/s: 0 rss: 30Mb
228 NEW    cov: 9 ft: 10 corp: 2/7b lim: 6 exec/s: 0 rss: 30Mb L: 6/6 MS: 1 InsertRepeatedBytes-
6863    NEW    cov: 10 ft: 11 corp: 3/13b lim: 10 exec/s: 0 rss: 30Mb L: 6/6 MS: 5
 InsertRepeatedBytes-ChangeByte-ChangeBit-ChangeASCIIInt-InsertByte-
68469   NEW    cov: 11 ft: 12 corp: 4/20b lim: 10 exec/s: 0 rss: 30Mb L: 7/7 MS: 1 InsertByte-
297413  NEW    cov: 12 ft: 13 corp: 5/26b lim: 10 exec/s: 0 rss: 30Mb L: 6/7 MS: 4 EraseBytes-
CopyPart-ChangeASCIIInt-ChangeBit-
317189  NEW    cov: 13 ft: 14 corp: 6/33b lim: 10 exec/s: 0 rss: 30Mb L: 7/7 MS: 1 InsertByte-
323801  NEW    cov: 14 ft: 15 corp: 7/42b lim: 10 exec/s: 0 rss: 30Mb L: 9/9 MS: 2 EraseBytes-
CopyPart-
```

We ran both experiments with `-- -max_len=10 -reduce_inputs=0` libFuzzer parameters to give the `libFuzzer` engine a strong size hint while avoiding extra log entries from unnecessary optimization steps, which should be favorable to both variants. Note that the `exec/s: 0` display is an incorrect statistic given by the fuzzer and does not actually represent the number of executions per second.

In our understanding, coverage counters on error paths and invalid inputs can be very valuable for incrementally discovering important edge cases or to help the fuzzer find fully valid inputs. We think this also applies to fuzzing file-parser-related code such as the bzip2 decompression steps. By rejecting most inputs that don't parse perfectly, the coverage-guided fuzzer engine effectively degrades to a black box fuzzer, severely limiting its practical effectiveness in finding problems that are not just a few mutation steps away from already known inputs in the corpus. For this reason, we recommend avoiding the use of `Corpus::Reject` except for marking known-uninteresting narrow error cases or performing filtering during specially configured fuzzer runs.

5.9 NF-009 — Fuzz harness bug in decompress.rs

During our review, we found two functional issues in the decompress.rs fuzz harness.

Consider the following code:

```
match unsafe { BZ2_bzDecompress(&mut strm) } {
[...]
            libbz2_rs_sys::BZ_FINISH_OK | libbz2_rs_sys::BZ_OUTBUFF_FULL => {
                let add_space: u32 = Ord::max(1024, output.len().try_into().unwrap());
                output.resize(output.len() + add_space as usize, 0);

                // If resize() reallocates, it may have moved in memory.
                strm.next_out = output.as_mut_ptr().cast::<core::ffi::c_char>();
                strm.avail_out += add_space;
            }
```

The intention of this code is to detect a full output buffer, and then to resume decompression with a resized buffer. However, `BZ2_bzDecompress()` does not trigger `BZ_FINISH_OK` or `BZ_OUTBUFF_FULL` if the output buffer is full. Additionally, the output pointer reassignment of `strm.next_out` to the `output` buffer does not account for the offset of data already written to `output`, which would have corrupted the output data.

Both issues have been fixed via pull request no. 109, which was merged by the developers.

6 Future Work

- **Run extended fuzz tests in CI**
  Currently, all fuzz tests in GitHub Actions run on a fixed start corpus and only for a very limited time. This is enough to trigger bugs which are found very quickly using the existing fuzzer corpus, but does not reach more complex issues which would be found only after some minutes or hours. Consider separate CI runs, such as optional tests that run daily or weekly, which have more time budget or are able to resume their run using the corpus directory from prior runs.
- **Use multi-threading in CI**
  Currently, the fuzz tests in GitHub Actions run in their default threading mode, which uses a single vCPU core. Consider scaling the fuzz test operations to all available logical vCPU cores by enabling parallelization. Currently, normal GitHub Linux runners for public projects have four cores allocated by default, which could enable up to 4x of total test executions within a given run duration.
- **Consider testing against more `libbzip2` versions**
  The differential testing concept used in multiple `libbzip2-rs` fuzz tests allows the discovery of anomalies and mismatches between the tested `libbzip2-rs` and `libbzip2` versions. However, older or newer versions of the two implementations may behave differently against each other. Consider optionally testing more versions.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your information security.

7 Conclusion

During this review, we discovered two functionality-related bugs in the library, but no security issues.

Unlike other audits, this review was primarily focused on leveraging and improving the existing automated testing mechanisms within the `libbzip2-rs` project to extend its error-finding capabilities. Notably, the project developers have already spent a significant amount of time over recent months to ensure their new Rust implementation is exactly aligned with the existing C implementation, including the creation of custom fuzz tests. Additionally, they configured their Continuous Integration (CI) system to do limited automatic tests and fuzzing. We think that both steps helped to catch and avoid issues that we would usually find and flag in code auditing projects.

Despite these preparations, we still discovered several bugs and undocumented limitations in the fuzz tests that limited their usefulness for testing. This is in the nature of these tests, which are often experimental, but can lead to reliability issues and blind spots. For example, several fuzz tests did not correctly handle test inputs above a certain size, leading to false-positives. In one fuzz test, the state handling was also incorrect, limiting the discovered complexity. Together with the project developers, we were able to identify and fix these issues. We also gave the fuzzer engine more control over parameters such as the compression level, and helped the project developers with the evaluation of advanced fuzz mechanisms and experimental setups, such as running non-standard runtime sanitizers for special error detection.

Through these efforts, we discovered an off-by-one error in a bounds check in the decompression stage, which had so far evaded detection. Although this bug did not have any security implications, it underlines the value of fuzz testing for a project like `libbzip2-rs`, which aims to transfer existing functionality into a different programming language while avoiding translation mistakes or introducing new bugs.

Overall, we think the project is in a good shape when it comes to security.

We would like to thank the `libbzip2-rs` team for their support and feedback during the review, especially Folkert de Vries.

Finally, we want to emphasize that security is a process that must be continuously evaluated and improved – this code audit is just a one-time snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your project's information security. We hope that this audit report, and the detailed explanations of our findings or non-findings, will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing Team

Christian Reitter *(pentester)*	Christian is an IT Security Consultant with experience in the area of software security and security relevant embedded devices. After his M.Sc. in Computer Science, he has worked as a developer and freelance security consultant with a specialization in fuzz testing. Notable published research includes several major vulnerabilities in popular cryptocurrency software and hardware wallets. This includes remote recovery of wallet keys with weak entropy, remote code execution on hardware wallets, remote theft of secret keys from hardware wallets and circumvention of 2FA protection. He also discovered multiple memory issues in well-known smartcard driver stacks.
Melanie Rieback *(approver)*	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.