Zlib

Automated Security Assessment

Zlib version 1.2.8
September 30th, 2016

Prepared For:
Gervase Markham, Chris Riley
Secure Open Source

Prepared By:
Artem Dinaburg | *Trail of Bits*
artem@trailofbits.com

Pascal Cuoq | *TrustInSoft*
cuoq@trust-in-soft.com

Changelog

August 25, 2016:	Initial report delivered
September 28, 2016:	Added Appendix B with Retest Results
September 30, 2016:	Publication version

Synopsis

In August 2016, Mozilla engaged Trail of Bits to perform a security assessment of Zlib as part of the Mozilla Secure Open Source (SOS) Fund. Zlib is an open source compression library with a history of security vulnerabilities. Mozilla is interested in the security of zlib because zlib is used in virtually every software package that requires compression or decompression. Securing zlib will help secure the core infrastructure of the Internet.

To perform this audit, Trail of Bits used the automated vulnerability discovery tools developed for the DARPA Cyber Grand Challenge[1]. These tools automatically audited zlib for security vulnerabilities, delivering confidence while reducing costs. Additionally, we teamed with colleagues at TrustInSoft to augment our assessment with the unique capabilities of their verification toolkit.

Engagement Goals

The goal of the engagement was to evaluate zlib for security vulnerabilities and to serve as a proof-of-concept for automated software audits. The audit used a two-pronged approach: testing to find bugs in the final compiled zlib library, and verification to find latent bugs in the zlib source code.

The testing tools were used to search for memory safety vulnerabilities, such as buffer overflows, heap overflows, out of bounds array access, and so on. Similar bugs had been previously identified in zlib. The verification tools were used to identify violations of the C standard; specifically, the use of undefined behavior.

Areas of Focus

The audit focused on the compression and decompression functionality provided by zlib. While zlib exposes several APIs, it is the compression and decompression functionality that typically processes untrusted input. This code may be reached via two general paths: processing raw data input and processing GZip formatted input. We tested both paths.

We did not inspect functions that query or set internal library state, provide replacements for C file I/O functionality (other than necessary to test GZip decoding), nor functions that deal with very specific zlib use cases, such as forcing specific decompression strategies. These functions are unlikely to accept user input or are very unlikely to be used in an application.

[1] https://www.cybergrandchallenge.com/

Executive Summary

Application Summary

Name	Zlib
Version	1.2.8 (http://zlib.net/zlib-1.2.8.tar.gz)
Type	Userspace Compression Library
Platform	Cross-Platform (tested on Linux)

Engagement Summary

Engagement Type	Bug Hunting + Verification
Engagement Tools	Trail of Bits CRS + TrustInSoft TIS-Interpreter[2]
Consultants Engaged	2
Total Engagement Effort	5 days

Vulnerability Summary

Total High Severity Issues	0
Total Medium Severity Issues	1
Total Low Severity Issues	4
Total Informational Severity Issues	0
Total	5

Category Breakdown

Undefined Behavior	5

[2] https://github.com/TrustInSoft/tis-interpreter

Recommendations Summary

Do not rely on undefined behavior. We have identified five areas where code in zlib invokes undefined behavior in the C standard. Use of this code does not currently generate buggy binaries, but it is possible that with future compilers or platforms these latent bugs may manifest in compiled code.

Opt-in support for legacy compilers and micro-optimizations. Cases of undefined behavior mostly appear in code that is designed to work around old, buggy compilers or to provide micro-optimizations for specific architectures. Making this behavior optional rather than default would result in cleaner builds for common zlib usage.

Deprecate support for ancient compilers. Support for ancient buggy compilers and hacks to work around their bugs needlessly increases the complexity of zlib's codebase. Rarely used platforms should be deprecated to simplify the zlib codebase, making the code easier to read, understand, and audit.

Deprecate architecture-specific micro-optimizations. Architecture-specific micro-optimizations should be eliminated. They needlessly increase complexity, may introduce undefined behavior, and are likely unnecessary with modern optimizing compilers.

Conduct regular security audits. Zlib is a great candidate for periodic security audits: the code base is very stable and updates are infrequent. The long release cycle allows ample time to perform and respond to an automated security audit. Periodic audits can prevent the introduction of new bugs into the zlib codebase.

Methodology

Testing Methodology

The Trail of Bits Cyber Reasoning Systems (CRS) uses a combination of symbolic and concrete vulnerability hunting strategies to overcome roadblocks that stymie other security tools. Developed to compete in the DARPA Cyber Grand Challenge, it has since been extended to operate on Linux binaries. The CRS uses a new technique we call analysis boosting to combine our novel low-latency fuzzer with two symbolic execution engines. Using this approach, we are able to gather a fine-grained picture of program execution and identify subtle flaws that a human auditor would likely miss.

The only prior requirement is the creation of a "driver" program to exercise target functionality in zlib. We created drivers for four zlib APIs -`compress`, `uncompress`, `gzread`, `gzwrite`- to exercise the compression, decompression, and GZip file format functionality inside zlib. These drivers were linked with zlib and run inside the CRS. The CRS then investigated various program states, switching between concrete and symbolic execution as needed to explore deeper in the zlib code. Each tested function was run through the CRS for two weeks on a 48-core machine.

The tested functions accept arguments that affect how zlib will process inputs. The chief of these are the input buffer size and the destination buffer size (for `compress` and `uncompress`) and the file mode for `gzread` and `gzwrite`. These were set to fixed values shown in Table 1. The buffer sizes were chosen to be large enough to test a variety of inputs, yet small enough to be analyzed in a reasonable timeframe.

Tested Function	**Input Buffer Size (bytes)**	**Output Buffer Size (bytes)**	**File Mode**
`compress`	4096	2048	n/a
`uncompress`	4096	163839	n/a
`gzread`	n/a	4096	"rb"
`gzwrite`	4096	n/a	"wb"

Table 1: Buffer sizes used when testing zlib functions

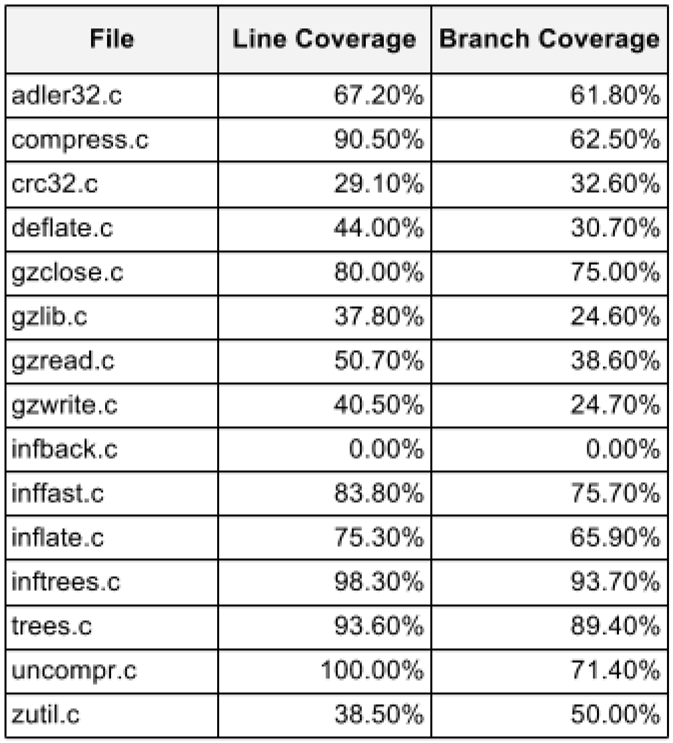
The file mode used with `gzread` and `gzwrite` can specify a compression or decompression strategy in addition to whether the file is opened for reading or writing (e.g. the file mode “wbR” would use the RLE compression strategy). The modes we selected will use the default compression strategies -- this is the most probable use-case of zlib, but will miss code coverage that is dependent on specific strategies.

To enhance the CRS’s thoroughness, the `uncompress` and gzread tests were seeded with data compressed at different compression levels. The `compress` and `gzwrite` tests were not seeded. All inputs were generated by the CRS with no prior knowledge of the input format or requirements.

Testing Coverage

During this assessment, we focused on typical zlib usage and code related to compression and decompression functionality. Zlib not only provides functionality for compressing and decompressing data, but it also includes functionality to check compression bounds, control internal library state, operate on file streams, special functionality to operate on large files, etc.

The CRS generated very high coverage for the location of previous zlib vulnerabilities[3]: the Huffman tree code (inftrees.c: 98.3% line coverage, 93.7% branch coverage, trees.c: 93.6% line coverage, 89.4% branch coverage). The file infback.c has 0% coverage; it is an alternative to inflate.c and is used when callback style I/O is preferred.



File	Line Coverage	Branch Coverage
adler32.c	67.20%	61.80%
compress.c	90.50%	62.50%
crc32.c	29.10%	32.60%
deflate.c	44.00%	30.70%
gzclose.c	80.00%	75.00%
gzlib.c	37.80%	24.60%
gzread.c	50.70%	38.60%
gzwrite.c	40.50%	24.70%
infback.c	0.00%	0.00%
inffast.c	83.80%	75.70%
inflate.c	75.30%	65.90%
inftrees.c	98.30%	93.70%
trees.c	93.60%	89.40%
uncompr.c	100.00%	71.40%
zutil.c	38.50%	50.00%

Table 2: Code coverage in zlib

This coverage is a result of both invoking the compression related code directly and via GZip functionality. This compares very favorably to the hand-crafted unit tests that come with zlib, which generate 100% coverage for infback.c, inffast.c, and inftrees.c, 98.6% coverage for inflate.c, and almost zero coverage for anything else.

[3] https://www.immunityinc.com/downloads/zlib.pdf

Compression Functionality

The tested compression function, `compress` is an alias for `compress2` with a default compression level (equivalent to setting compression level 6). In turn, `compress2` will call into core zlib code: it will initialize a deflate stream (`deflateInit`), deflate the input buffer (`deflate`), and close the deflate stream (`deflateEnd`).

This core deflate functionality is present in deflate.c. The CRS achieved 44.0% line coverage in deflate.c. That number is deceiving. Much of the functionality requires setting custom parameters for compression strategy, memory size, window bits, etc., and is unreachable from the `compress` function (or typical use of zlib). Conversely, the functionality responsible for the actual deflation process (trees.c) has very high coverage under the CRS (93.6% line coverage). This is the part of the code most readily reached via varying inputs (instead of configuration options), and it is very well tested.

Decompression Functionality

The function responsible for decompression, `uncompress`, is effectively the mirror opposite of `compress`, and has very similar behavior. The `uncompress` function is a wrapper for `uncompress2`, which itself calls into core zlib code in inflate.c: `inflateInit`, `inflate`, `inflateEnd`.

The decompression code is well covered: 75.3% of inflate.c and 83.8% of inffast.c was reached via the CRS. This code is reachable by both the `uncompress` function and via GZip related code paths. Most of the uncovered code is only reachable via different entry points not utilized during typical zlib usage (e.g. `inflateSync`, `inflateCopy`, `inflatePrime`).

Similarly to compression, the code responsible for the deflation of input data (inftrees.c), which is most readily reached by varying input data instead of configuration options, is very well tested, at 98.3% line coverage.

GZip Functionality

Zlib includes compression-friendly replacements for common libc file operation functions. These functions transparently operate on GZip[4] compressed data, automatically compressing and decompressing file streams using zlib. After our initial investigation of `compress` and `uncompress`, it became evident that some functionality was only reachable via GZip related code, so we also tested `gzread` and `gzwrite`, the compression friendly replacements of `fread` and `fwrite`.

[4] http://www.zlib.org/rfc-GZip.html

File	Line Coverage (%)	Exported Functions
gzclose.c	80.0%	**gzclose**
gzlib.c	37.8%	gzopen, gzopen64, **gzdopen**, gzbuffer, gzrewind, gzseek64, gzeek, gtell64, gztell, gzoffset64, gzoffset, gzeof, **gzerror**, gzclearerr
gzread.c	50.7%	gzgets, gzdirect, **gzclose_r**, **gzread**, gzgetc, gzgetc_
gzwrite.c	40.5%	**gzwrite**, gzputc, gzputs, gzvprintf, gzprintf, gzflush, gzsetparams, **gzclose_w**

Table 3: Code coverage of GZip functionality in zlib. The functions we created drivers for and tested are in **bold**.

The code coverage for GZip related functionality is in Table 3. Our goal was to enhance testing of compression and decompression functionality reachable via GZip and to stress GZip format parsing code. The unreached code includes many different entry points for libc replacement functions (i.e. `gzseek`, `gztell`, `gzprintf`, etc.) which were not evaluated.

Verification Methodology

TrustInSoft relies on a combination of formal method techniques that, together, deliver powerful and efficient solutions for high-assurance software security analysis. The tool used in this instance, TIS-interpreter, interprets C programs statement by statement from beginning to end, verifying at each statement whether the program can invoke undefined behavior. TIS-interpreter can detect violations of the C standard even when applied to regression tests that have never revealed previous problems.

Verification Details

TIS-interpreter, set to interpret the source code according to the implementation-defined choices of GCC on x86-64, was launched on the minigzip test program provided with zlib with inputs generated by afl-fuzz, and directly on the `compress` and `uncompress` functions using inputs generated by the CRS. Execution in TIS-interpreter is tens of thousands of times slower than execution of the same code when compiled. A minimal test suite exercising every reached statement was extracted from the set of inputs generated by the CRS in order to be executed inside TIS-interpreter.

TIS-interpreter was used in a mode in which both the possibilities of a null pointer and a valid pointer being returned are considered, for each individual allocation. Thus, the results obtained capture any possible behavior of zlib along the execution of the tests for any interlacing of allocation successes and failure.

In addition, the list of remaining allocated memory blocks was checked to be empty for each of these possible executions, thus checking that zlib does not have any memory leak for any sequence of allocations successes and failures when handling the inputs.

Findings Summary

This section covers our findings from both a software testing and a software verification standpoint. Software testing attempts to identify vulnerabilities that are present in compiled binary code; software verification identifies vulnerabilities that are present at the source level. These vulnerabilities may not manifest as problems right now, but could at a later date (e.g. when compilers make different assumptions about undefined behavior).

Software Testing

The CRS is especially tuned for identifying memory safety violations. This class of bug encompasses many common security vulnerabilities, such as buffer overflows, use-after-free errors, stack overflows and heap overflows. Using the CRS, we were unable to identify memory safety issues with the `compress`, `uncompress`, `gzread` and `gzwrite` functions in zlib. We conclude that the assessed code is highly unlikely to harbor these types of bugs.

Software Verification

Using TIS-Interpreter, clang, and human review, several instances of zlib relying on undefined behavior were identified, when compiled with default settings. Relying on undefined behavior has caused security issues in the past, and may result in bugs[5] in compiled zlib code in the future.

Table of Findings

#	Title	Severity
1	Incompatible declarations for external linkage function deflate	Medium
2	Accessing a buffer of char via a pointer to unsigned int	Low
3	Out-of-bounds pointer arithmetic in inftrees.c	Low
4	Undefined Left Shift of Negative Number	Low
5	Big Endian Out Of Bounds Pointer	Low

[5] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

Findings

Finding 1: Incompatible declarations for external linkage function deflate

Severity: Medium
Type: Undefined Behavior

Description:
The declaration and implementation of the `deflate` function use incompatible types. The first argument to deflate is of type `zstreamp`, which is a pointer to an internal structure that has a member of type `struct internal_state`. In a default compilation of zlib, `struct internal_state` is re-defined after the declaration of `deflate`.

Even though current compilers process this code without issues, this bug is rated severity medium for two reasons. First, the `deflate` function is used by virtually every application that includes zlib. Second, there is no telling what interactions the bug could have in the future with link-time optimizations and type-based alias analyses, both features that are present (but not default) in clang.

Source Reference (zlib.h):

```
/* hack for buggy compilers */
#if !defined(ZUTIL_H) && !defined(NO_DUMMY_DECL)
    struct internal_state {int dummy;};
#endif
```

The reason this defect is not detected by ordinary compilation chains is that `deflate` is declared with incompatible types in different compilation units.

This results from a workaround for buggy compilers. Venturing a guess as to the origin of this wart, some ancient C compilers must have complained (wrongly) when '`struct internal_state;`' was used in a compilation unit without the list of members of the struct, and their respective types, being defined. If so, the comment is correct in calling the compilers buggy: this is one of the very few abstraction mechanisms in standard C, and it has been widely used as a result in plenty of other C code written since zlib. The buggy compilers, if they were still used, would choke on everything else apart from zlib.

Recommendations:
This bug should be fixed immediately by either removing support for buggy compilers, or by making `NO_DUMMY_DECL` a default compile-time option, thereby forcing buggy compilers to opt-in to workaround behavior.

In the longer term, support for old and buggy compilers should be deprecated. The result will be less code, fewer bugs, and a cleaner, easier-to-read and -audit codebase.

Finding 2: Accessing a buffer of char via a pointer to unsigned int

Severity: Low
Type: Undefined Behavior

Description:
In the `crc32_little` and `crc32_big` functions, a pointer to unsigned int is used to access a buffer of type char*.

Source reference (crc32.c):

```
#define DOLIT4 c ^= *buf4++; \
        c = crc_table[3][c & 0xff] ^ crc_table[2][(c >> 8) & 0xff] ^ \
            crc_table[1][(c >> 16) & 0xff] ^ crc_table[0][c >> 24]
#define DOLIT32 DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4
…
local unsigned long crc32_little(crc, buf, len)
    unsigned long crc;
    const unsigned char FAR *buf;
    unsigned len;
{
…
    while (len && ((ptrdiff_t)buf & 3)) {
        c = crc_table[0][(c ^ *buf++) & 0xff] ^ (c >> 8);
        len--;
    }

    buf4 = (const z_crc_t FAR *)(const void FAR *)buf;
    while (len >= 32) {
        DOLIT32;
        len -= 32;
    }
    while (len >= 4) {
        DOLIT4;
        len -= 4;
…
```

The body of `crc32_little` contains four similar loops that each update `c`, increment `buf`, and decrement `len`. One of the loops accesses the buffer through a pointer to words (z_crc_t), for speed.

This access violates “strict aliasing rules,” as they are known (after the name of the GCC optimization -fstrict-aliasing). In short, the strict aliasing rules mandate that memory is read with the same type that was used to write it[6].

Optimizing compilers assume that the strict aliasing rules violated here are always respected. They do not make an effort to detect violations of strict aliasing, and hence do not warn about such violations. It does not matter if the reading and writing of the buffer occurs inside or outside of the same function, or within different compilation units; strict aliasing is still violated.

Current optimizing compilers will produce erroneous results, like different CRC32 values for the same memory contents, in a toy example that violates strict aliasing. The only complications that prevent current compilers from applying similar transformations to more complex code, such as zlib, are:

1) The type `char` is special (see C11 6.5p7). This does **not** mean that it is permissible to write values as chars but read them via a pointer to words. The `char` type scares compiler authors into assuming that writes through `char` lvalues can modify anything. One reason for this is because the standard differentiates between dynamically allocated memory and program variables. Compilers do not yet have static analyses to infer whether a pointer to char points to dynamic memory or program variables.

2) Separate compilation units. The modifications of the buffer, as an array of chars, take place in a different compilation unit than that of the CRC32 functions.

Future compilers may resolve both of these complications.

Recommendations:
There are several possible fixes for this bug:

- Do nothing. There may not be a uniformly superior version with the same combination of speed, standards conformance, and portability.

- If speed doesn't matter, suppress three out of the four similar loops. This alternative also makes the big- and little-endian versions of the CRC computation identical, so that the code in crc32.c can be greatly simplified:

[6] http://port70.net/~nsz/c/c11/n1570.html#6.5p7

```
diff --git a/crc32.c b/crc32.c
index 979a719..d867796 100644
--- a/crc32.c
+++ b/crc32.c
@@ -250,25 +250,9 @@ local unsigned long crc32_little(crc, buf, len)
     unsigned len;
 {
     register z_crc_t c;
-    register const z_crc_t FAR *buf4;

     c = (z_crc_t)crc;
     c = ~c;
-    while (len && ((ptrdiff_t)buf & 3)) {
-        c = crc_table[0][(c ^ *buf++) & 0xff] ^ (c >> 8);
-        len--;
-    }
-
-    buf4 = (const z_crc_t FAR *)(const void FAR *)buf;
-    while (len >= 32) {
-        DOLIT32;
-        len -= 32;
-    }
-    while (len >= 4) {
-        DOLIT4;
-        len -= 4;
-    }
-    buf = (const unsigned char FAR *)buf4;

     if (len) do {
         c = crc_table[0][(c ^ *buf++) & 0xff] ^ (c >> 8);
```

- If speed matters and non-standard GCC extensions in the source code (recognized by clang) are acceptable, use the may_alias type attribute. This would be a good opportunity to replace both the 4-at-a-time and the 32-at-a-time loops by a single 8-at-a-time loop that takes advantage of modern 64-bit instruction sets.

- If speed matters and the C code must be portable, it can be built with the -fno-strict-aliasing option when using the clang or gcc compilers.

- See one additional solution in Appendix B.

In the long term, platform-specific micro-optimizations should be deprecated. These optimizations make the code more complex and may harbor latent bugs, such as this one. Modern compilers are much better at optimizing and vectorizing code than they used to be. Removing micro-optimizations will allow for simpler, more bug-free code.

Finding 3: Out-of-bounds pointer arithmetic in inftrees.c

Severity: Low
Type: Undefined Behavior

Description:
Zlib computes out-of-bounds pointers in several places, even though these pointers are not dereferenced. Still, using pointer arithmetic in order to go out of the bounds of the pointed block is forbidden by the C standard, and compiler optimizations exist that assume code does not do this[7].

Places where out-of-bounds pointer arithmetic is used are shown below.

Source Reference (inftrees.c):
```
   60     static const unsigned short lbase[31] = { /* Length codes 257..285 base
*/
   …
   63     static const unsigned short lext[31] = { /* Length codes 257..285 extra
*/
   …
   187        base = lbase;
   188        base -= 257;
   189        extra = lext;
   190        extra -= 257;
```

There is another instance in inffast.c where, with the default configuration (`POSTINC` **not** defined), a pointer one-before-the-beginning is computed as the initial value for the variables `in` and `out`.

Source Reference (inffast.c):
```
    24 #ifdef POSTINC
    …
    27 #else
    28 #  define OFF 1
    29 #  define PUP(a) *++(a)
    30 #endif
    …
    99     in = strm->next_in - OFF;
    …
    101          out = strm->next_out - OFF;
    …
    122                  hold += (unsigned long)(PUP(in)) << bits;
```

[7] https://lwn.net/Articles/278137/

Recommendations:
Rewrite the code in inftrees.c to avoid out-of-bound pointer computation.

The workaround for inffast.c is simpler: `POSTINC` must be defined in inffast.c. Defining `POSTINC` does not have any functional effects, but may not be as fast on some platforms, judging by the comments in inffast.c. These performance differences may not be evident on modern compilers that can better re-order instructions.

In the longer term, zlib should be periodically audited to detect potential undefined behavior. The code base is very stable and updates are infrequent, ensuring there is ample time to perform audits and fix code based on the findings.

Finding 4: Undefined Left Shift of Negative Number

Severity: Low
Type: Undefined Behavior

Description:
While testing the possible fix of the strict aliasing issue (Finding 2), we identified an invalid left shift of a negative number.

Source Reference (inflate.c):

```
if (strm == Z_NULL || strm->state == Z_NULL) return -1L << 16;
```

Left shifts of negative value are undefined, but in practice this will probably continue to have the desired behavior.

Recommendation:
Change -1L << 16 to (~0xFFFFL). Potential sample code:

```
if (strm == Z_NULL || strm->state == Z_NULL) return (~0xFFFFL);
```

Finding 5: Big Endian Out Of Bounds Pointer

Severity: Low
Type: Undefined Behavior

Description:
The `crc32_big` function will create an out of bound pointer when it decrements buf4 prior to use.

Source Reference (crc32.c):

```
…
while (len && ((ptrdiff_t)buf & 3)) {
    c = crc_table[4][(c >> 24) ^ *buf++] ^ (c << 8);
    len--;
}

buf4 = (const z_crc_t FAR *)(const void FAR *)buf;
buf4--;
…
```

Using pointer arithmetic in order to go out of the bounds of the pointed block is forbidden by the C standard. Compiler optimizations assume code does not do this.

Recommendation:
This issue may be solved by removing the dependence on buf4 while addressing Finding 2 (strict aliasing violation), or by a change similar to this commit. After this change, there are no longer any detectable issues with the big-endian version of zlib.

Appendix A. Generated Inputs and Coverage Data

Attached with this report are:

- Raw coverage data and HTML formatted code coverage information (zlib_coverage.tar.gz)
- The inputs to `compress` to generate that coverage (compress_inputs.tar.gz)
- The inputs to `uncompress` to generate that coverage (uncompress_inputs.tar.gz)
- The inputs to `gzread` used to generate code coverage (gzread_inputs.tar.gz)
- The inputs to `gzwrite` used to generate code coverage (gzwrite_inputs.tar.gz)

The input sets were extracted from the CRS, and are representative tests of different code paths that the CRS was able to reach.

Appendix B. Fix Log

Zlib made the following modifications to their codebase as a result of this report. Each of the fixes was verified by the audit team.

Finding 1: Incompatible declarations for external linkage function deflate (Med)
https://github.com/madler/zlib/commit/3fb251b363866417122fe54a158a1ac5a7837101

Finding 2: Accessing a buffer of char via a pointer to unsigned int (Low)
There is currently no available fix for Finding 2 that meets Zlib's compatibility and performance goals. Currently available compilers produce correct code for this construction, however, we cannot predict what future compilers may or may not do.

Finding 3: Out-of-bounds pointer arithmetic in inftrees.c (Low)
https://github.com/madler/zlib/commit/6a043145ca6e9c55184013841a67b2fef87e44c0
https://github.com/madler/zlib/commit/9aaec95e82117c1cb0f9624264c3618fc380cecb

Finding 4: Undefined left shift of negative number (Low)
https://github.com/madler/zlib/commit/e54e1299404101a5a9d0cf5e45512b543967f958

Finding 5: Big-endian out-of-bounds pointer (Low)
https://github.com/madler/zlib/commit/d1d577490c15a0c6862473d7576352a9f18ef811

Alternate Solution for Finding 2

The Zlib development team pointed out that compilers would not be able to vectorize and unroll that code if it were one loop, and that performance is critical.

On the next page is a potential `crc32_little` that avoids a strict aliasing violation. We have verified that recent versions of clang and gcc optimize the `memcpy` to a memory load, even without specifying an optimization level. We have tested but not profiled this as a possible solution.

If the proposed solution does not meet performance goals, and there is not a simple way to both keep the manually unrolled loops and not use buf4, another option would be to add -fno-strict-aliasing to build flags or the may_alias attribute to buf4, for compilers that support such features.

Potential changes to crc32_little (crc32.c):

```
#define DOLIT4 memcpy(&v, buf, sizeof(z_crc_t)); \
        c ^= v; buf += sizeof(z_crc_t); \
        c = crc_table[3][c & 0xff] ^ crc_table[2][(c >> 8) & 0xff] ^ \
            crc_table[1][(c >> 16) & 0xff] ^ crc_table[0][c >> 24];
#define DOLIT32 DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4; DOLIT4

/* ========================================================================= */
local unsigned long crc32_little(crc, buf, len)
    unsigned long crc;
    const unsigned char FAR *buf;
    unsigned len;
{
    register z_crc_t c;
    z_crc_t v;

    c = (z_crc_t)crc;
    c = ~c;
    while (len && ((ptrdiff_t)buf & 3)) {
        c = crc_table[0][(c ^ *buf++) & 0xff] ^ (c >> 8);
        len--;
    }

    while (len >= 32) {
        DOLIT32;
        len -= 32;
    }
    while (len >= 4) {
        DOLIT4;
        len -= 4;
    }

    if (len) do {
        c = crc_table[0][(c ^ *buf++) & 0xff] ^ (c >> 8);
    } while (--len);
    c = ~c;
    return (unsigned long)c;
}
```

About Trail of Bits

Since 2012, Trail of Bits has helped secure some of the world's most targeted organizations and devices. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code.

Our clientele - ranging from Facebook to DARPA - lead their industries. Their dedicated security teams come to us for our foundational tools and deep expertise in reverse engineering, cryptography, virtualization, malware behavior and software exploits. We help them assess their products or networks, and determine the modifications necessary for a secure deployment. We're especially well suited for the technology, finance and defense industries.

After solving the problem at hand, we continue to refine our work in service to the deeper issues. The knowledge we gain from each engagement and research project further hones our tools and processes, helping us extend software engineers' abilities. We believe the most meaningful security gains hide at the intersection of human intellect and computational power.

About TrustInSoft

TrustInSoft provides exhaustive source code analyzers deployed in safety critical domains, such as aeronautics, defense, energy, railways, space, and telecom. TrustInSoft reduces cyber risks and lower the cost of designing safety-critical systems. These solutions validate mission-critical software and eliminate attack vectors, both allowing more efficient use of limited resources, and reducing liabilities. TrustInSoft's solutions rely on a combination of formal method techniques that, together, deliver powerful and efficient solutions for high-assurance software security analysis.