Final Writeup

p7zip

Arnav Nidumolu, Atharva Kale, Pascal von Fintel, Patrick Negus

2023-05-14

Final Writeup

Public Github Repository - This should include all code you wrote for eg. static analysis, fuzzing harnesses, etc. If you built your target with instrumentation for the purposes of fuzzing, this should also include build scripts. If you performed reverse engineering on your target and eg. started renaming variables/functions/did work on that front, include the relevant ghidra files as well.

Start your writeup with a description of what you learned about this target. This should include some notes about the code layout, maybe some coding practices you noticed while going through the target or just more general functionality. Which parts of the target did you think were most interesting for the purposes of finding bugs?

Describe what you chose for your automated analysis portion and why. How did you set this up, did you encounter issues (eg. slow fuzzer performance), and if so what did you to improve on these issues.

What were the biggest challenges you faced when dealing with your target?

If given more time, what do you think would be good next steps to continue doing research on the target with the goal of finding bugs?

Contents:

Github Link

https://github.com/atharvakale343/p7zip-390r

Overview of the Target

p7zip is a fully compliant linux port of the open source *7zip* tool for Windows. It is a utility used to archive and extract various compression formats. It is primarily used in Windows GUI tools as an underlying utility to support their file compression features.

p7zip provides the following features:

1. Several compression algorithms (*lz4*, *zstd*, *Lizard*, etc…)
2. CLI frontend
3. Cryptographic algorithms for Home Exp archive encryption (*SHA256*, *AES*, *RAR5*, etc…)

Code Layout

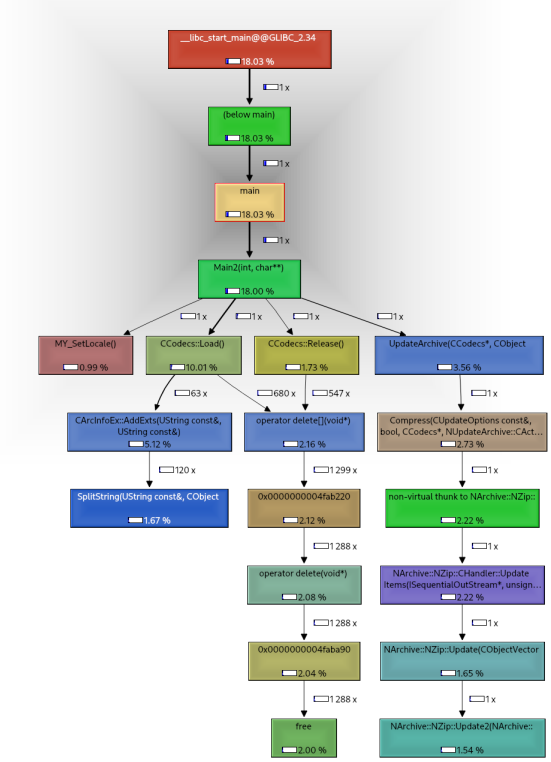

Figure 1: Call Graph for extract command

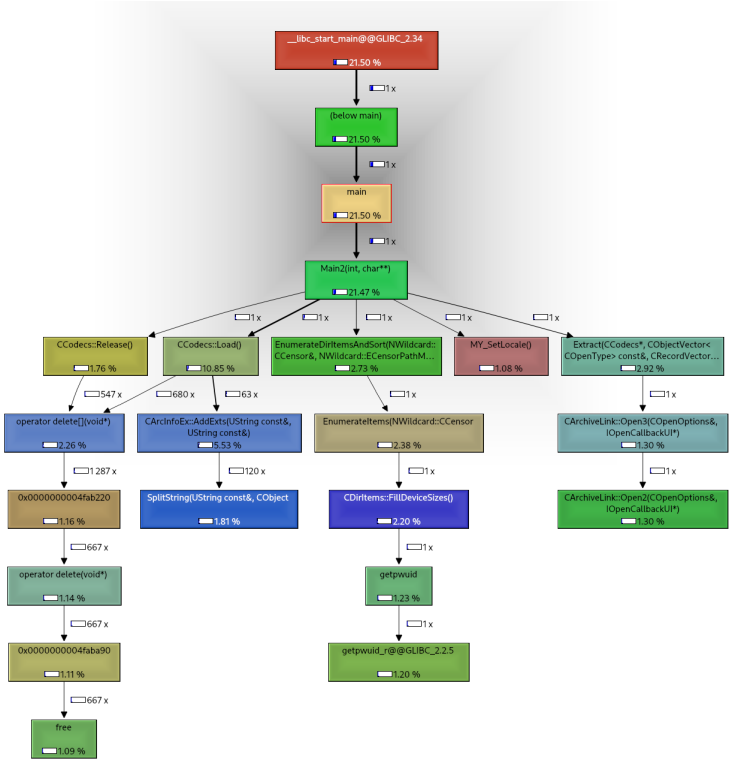

Figure 2: Call Graph for archive command

This is a very large codebase. Thankfully there was a bit of documentation on code layout. And because we were almost entirely using automated analysis tools, we didn't have to do a lot of interacting with the codebase directly - once we got everything working anyway.

DOC/readme.txt has some useful high level overviews of the codebase:

```
DOC                Documentation
---
  7zFormat.txt   - 7z format description
  copying.txt    - GNU LGPL license
  unRarLicense.txt - License for unRAR part of source code
  src-history.txt  - Sources history
  Methods.txt    - Compression method IDs
  readme.txt     - Readme file
  lzma.txt       - LZMA compression description
  7zip.nsi       - installer script for NSIS
  7zip.wix       - installer script for WIX


Asm - Source code in Assembler : optimized code for CRC, SHA, AES, LZMA decoding.

C   - Source code in C

CPP - Source code in C++

Common             common files for C++ projects

Windows            common files for Windows related code

7zip

  Common           Common modules for 7-zip

  Archive          files related to archiving

  Bundle           Modules that are bundles of other modules (files)

    Alone          7za.exe: Standalone version of 7-Zip console that supports only 7z/xz/cab/zip/gzip/bzip2/tar.
    Alone2         7zz.exe: Standalone version of 7-Zip console that supports all formats.
    Alone7z        7zr.exe: Standalone version of 7-Zip console that supports only 7z (reduced version)
    Fm             Standalone version of 7-Zip File Manager
    Format7z             7za.dll:  .7z support
    Format7zExtract      7zxa.dll: .7z support, extracting only
    Format7zR            7zr.dll:  .7z support, reduced version
    Format7zExtractR     7zxr.dll: .7z support, reduced version, extracting only
    Format7zF            7z.dll:   all formats
    LzmaCon        lzma.exe: LZMA compression/decompression
    SFXCon         7zCon.sfx: Console 7z SFX module
    SFXWin         7z.sfx: Windows 7z SFX module
    SFXSetup       7zS.sfx: Windows 7z SFX module for Installers

  Compress         files for compression/decompression

  Crypto           files for encryption / decompression

  UI

    Agent          Intermediary modules for FAR plugin and Explorer plugin
    Client7z       Test application for 7za.dll
    Common         Common UI files
    Console        7z.exe : Console version
    Explorer       7-zip.dll: 7-Zip Shell extension
    Far            plugin for Far Manager
    FileManager    7zFM.exe: 7-Zip File Manager
    GUI            7zG.exe: 7-Zip GUI version
```

Figure 3: Code Base Overview

Coding Observations

The functionality of the console version of this application is straightforward. The binary accepts command line arguments (main defined in MainAr.cpp), then passes them to main2(), where they are parsed.

From there, depending on the supplied command, extract/archive arguments are handled, with the respective call graphs given above.

P7zip's argument parser is robust, and given the functionality of our target (accepts and parses a file), it is clear that fuzzing is an obvious choice here for finding bugs.

Target Features

The main features of `7zz` is to extract from different compression formats and archive a collection of files. We decided to focus mainly on the `extract` or **e** command which accepts a compressed file and extracts its contents to the current directory.

We also looked into the `archive` or **a** command that compresses a list of files into a `.7z` file. This uses various compression algorithms such as LZ4, Brotli, Lizard, etc.

Automated Analysis

Fuzzing

Fuzzing was the main dynamic analysis technique we used against our target `p7zip`. We mainly fuzzed the extract (`e`) feature of our binary as the feature uses several decompression algorithms as part of its execution.

We used `afl-plus-plus` as the primary fuzzing tool.

https://github.com/AFLplusplus/AFLplusplus

Generating a corpus

We took a variety of steps to find a good enough corpus for our fuzzing efforts. The major approach here to was to search online for commonly used corpora. We wanted to find not only `.zip` format, but also as many different formats possible.

We found a decent corpus at https://github.com/strongcourage/fuzzing-corpus

This included the following formats:

- `.zip`
- `.gzip`
- `.lrzip`
- `.jar`

We added this as a target to our fuzzing Makefile.

```
get-inputs:
    rm -rf in_raw fuzzing-corpus && mkdir in_raw

    git clone -n --depth=1 --filter=tree:0 git@github.com:strongcourage/fuzzing-corpus.git
    cd fuzzing-corpus && git sparse-checkout set --no-cone zip gzip/go-fuzz lrzip jar && git checkout
    mv fuzzing-corpus/zip/go-fuzz/* in_raw
    mv fuzzing-corpus/jar/* in_raw
    mv fuzzing-corpus/gzip/go-fuzz/* in_raw
    mv fuzzing-corpus/lrzip/* in_raw
```

The next step was to choose only “interesting” inputs from this corpus. This includes small inputs that don’t crash that binary immediately.

We used the `afl-cmin` functionality to minimize the corpus.

```
afl-cmin -i in_raw -o in_unique -- $(BIN_AFL) e -y @@
```

Another important minimization step included `tmin`. This augments each input such that it can be as small as possible without compromising it’s ability to mutate and produce coverage in the instrumented target.

Unfortunately, this process takes a long time, and it only completed for us after a day.

```
cd in_unique; for i in *; do afl-tmin -i "$$i" -o "../in/$$i" -- ../$(BIN_AFL) e -y @@; done
```

The cybersec room servers come in handy here!

Experimenting with fuzzing composition flags

We discovered that it is not enough to fuzz a plain instrumented target with `afl-plus-plus`. The target binary may not be easily crashed with mutated inputs as `p7zip` has a robust input error checker. We took to fuzzing with various sanitizers instead to search for harder to find bugs.

We used the following sanitizers on our target:

- ASAN: Address Sanitizer: discovers memory error vulnerabilities such as use-after-free, heap/buffer overflows, initialization order bugs etc.

- MSAN: Memory Sanitizer: mainly used to discover reads to uninitialized memory such as structs etc.
- TSAN: Thread Sanitizer: finds race conditions

```
afl:
    rm -rf $(BIN_AFL)
    git clone $(GH_URL) $(BIN_AFL)
    cp 7zz-makefiles/$(BIN_DEFAULT).mak $(BIN_AFL)/CPP/7zip/7zip_gcc.
       mak
    cd $(BIN_AFL)/CPP/7zip/Bundles/Alone2 && CC=$(AFL_CC) CXX=$(AFL_CXX
       ) make -f makefile.gcc

afl-asan:
    rm -rf $(BIN_AFL_ASAN)
    git clone $(GH_URL) $(BIN_AFL_ASAN)
    cp 7zz-makefiles/$(BIN_AFL_ASAN).mak $(BIN_AFL_ASAN)/CPP/7zip/7
       zip_gcc.mak
    cd $(BIN_AFL_ASAN)/CPP/7zip/Bundles/Alone2 && AFL_USE_ASAN=1 CC=$(
       AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc

afl-msan:
    rm -rf $(BIN_AFL_MSAN)
    git clone $(GH_URL) $(BIN_AFL_MSAN)
    cp 7zz-makefiles/$(BIN_AFL_MSAN).mak $(BIN_AFL_MSAN)/CPP/7zip/7
       zip_gcc.mak
    cd $(BIN_AFL_MSAN)/CPP/7zip/Bundles/Alone2 && AFL_CC_COMPILER=LLVM
       AFL_USE_MSAN=1 CC=$(AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc

afl-tsan:
    rm -rf $(BIN_AFL_TSAN)
    git clone $(GH_URL) $(BIN_AFL_TSAN)
    cp 7zz-makefiles/$(BIN_AFL_TSAN).mak $(BIN_AFL_TSAN)/CPP/7zip/7
       zip_gcc.mak
    cd $(BIN_AFL_TSAN)/CPP/7zip/Bundles/Alone2 && AFL_USE_TSAN=1 CC=$(
       AFL_CC) CXX=$(AFL_CXX) make -f makefile.gcc
```

Extract command

Parallel Fuzzing

To start with, our approach was to fuzz the `extract` command of `7zz`. So we found an appropriate corpus and fuzzed with the `e` command-line argument (along with `-y` to account for same filenames / avoid user input hangs).

With all different sets of compilation flags that we mentioned previously, we compiled the binaries with AFL instrumentation. Then, to more effectively fuzz, we setup a parallel fuzzing environment in

one of the **CyberSec club** VMs.

We added the `afl-fuzz` commands in a `Makefile` and followed the official guide for using multiple cores. Below are the commands we utilized. All of our fuzzers shared the same input and output directores to keep track of current fuzzing state.

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 $(AFL_FUZZ)
    -M main-afl-$(HOSTNAME) -t 2000 -i in -o out -- $(BIN_AFL) e -y @@
```

Our main fuzzer used a regular instrumented AFL binary with no other `CFLAGS`. We used a timeout of 30 seconds to denote a hang (or infinite loops).

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 $(AFL_FUZZ)
    -S variant-afl-asan -t 2000 -i in -o out -- $(BIN_AFL_ASAN) e -y @@
```

Our variant fuzzers utilized binaries compiled with other flags (such as *asan* and *msan*). These had the same timeout as before of 30 seconds.

To keep track of all fuzzers and run them simultaneouly, we used `tmux` sessions with a separate window for each fuzzer.

Extract Fuzzing Results

We ran the fuzzers using multiple cores for around 5 days. We noticed no crashes in most of the variants, with ASAN being the exception. However, some fuzzers encountered hangs.

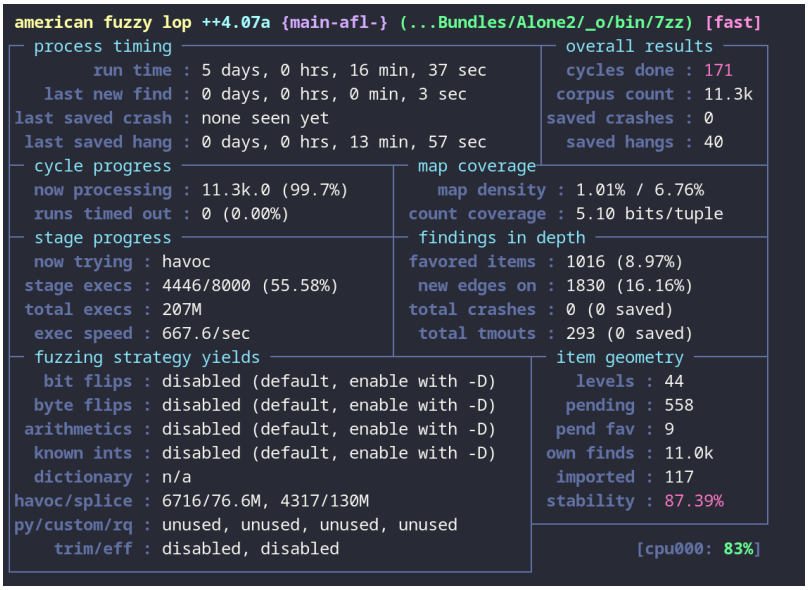

Figure 4: Main AFL Fuzzer

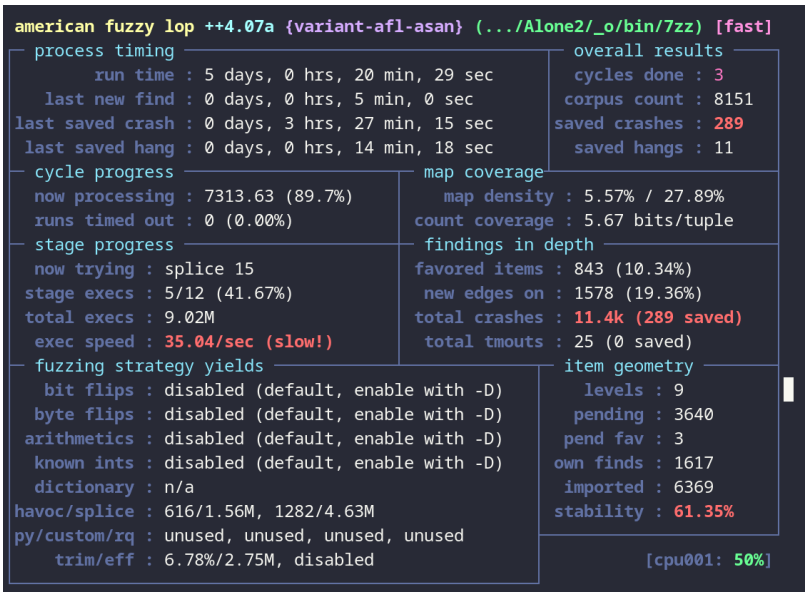

Figure 5: ASAN Variant Fuzzer

We tried running an input from `in/hangs` to check where an infinite loop could occur. But, all inputs

eventually terminated while taking longer than 30 seconds. Therefore, we concluded that these executions were incorrectly flagged as hangs due to large size of the file. We could possibly set the timeout even higher to avoid this issue.

Analyzing asan crashes

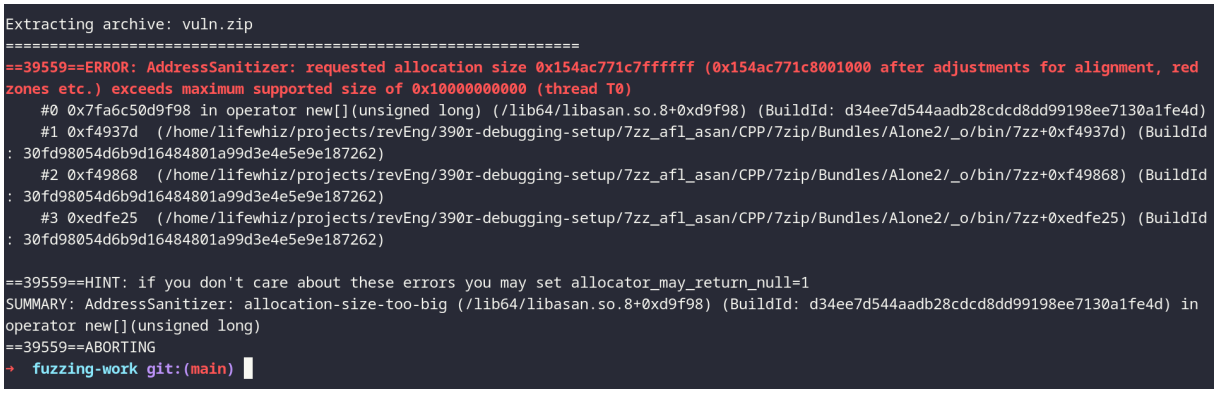

Figure 6: ASAN error output

Upon executing the ASAN compiled binary on one of the crash inputs, we found that it occured due to “request allocation size exceeding maximum supported size”. This likely occured due to malloc being called with a huge size argument and returning `NULL`.

```
HRESULT CUnpacker::UnpackData(IInStream *inStream,
    const CResource &resource, const CHeader &header,
    const CDatabase *db,
    CByteBuffer &buf, Byte *digest)
{
  // if (resource.IsSolid()) return E_NOTIMPL;

  UInt64 unpackSize64 = resource.UnpackSize;
  if (db)
    unpackSize64 = db->Get_UnpackSize_of_Resource(resource);

  size_t size = (size_t)unpackSize64;
  if (size != unpackSize64)
    return E_OUTOFMEMORY;

  buf.Alloc(size);

  CBufPtrSeqOutStream *outStreamSpec = new CBufPtrSeqOutStream();
  CMyComPtr<ISequentialOutStream> outStream = outStreamSpec;
  outStreamSpec->Init((Byte *)buf, size);

  return Unpack(inStream, resource, header, db, outStream, NULL, digest);
}
```

Figure 7: Location of error

The crash occurs at the `buf.Alloc` call, which executes a C++ **`new`** operation that internally calls `malloc`. Here, an argument of `unpackSize64` is passed into the function.

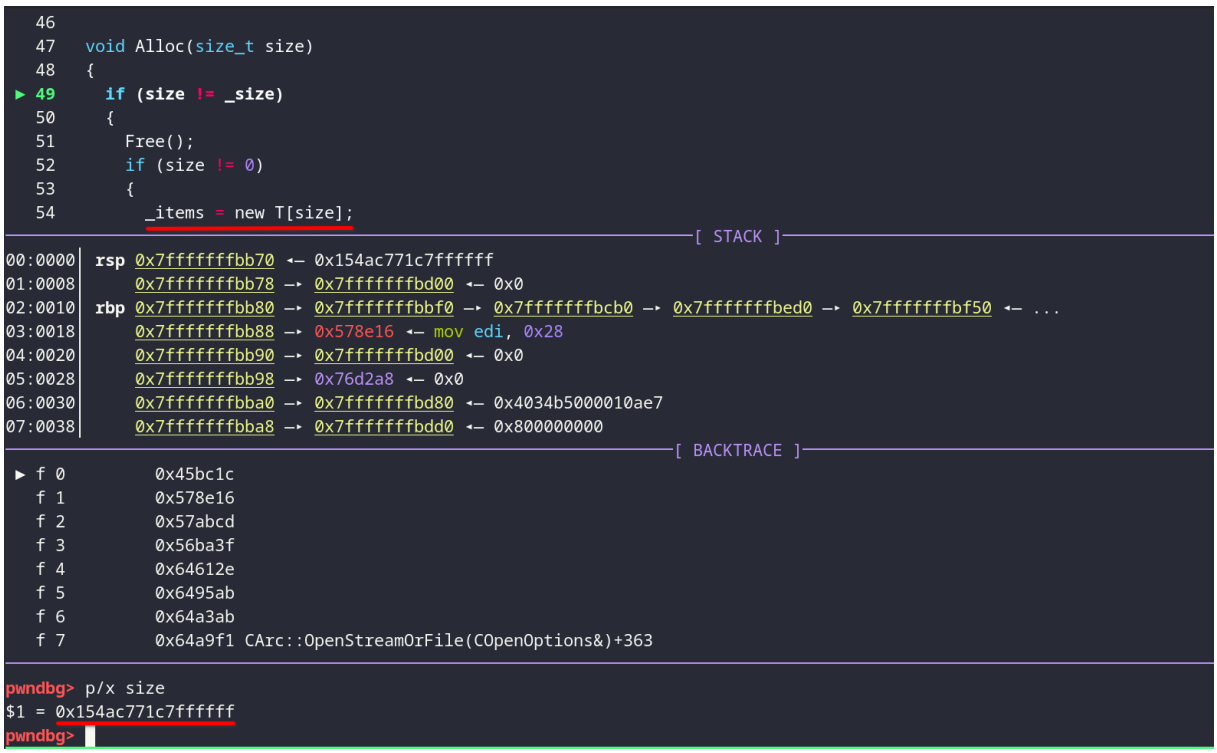

Figure 8: Analyzing size argument in GDB

As shown above, the `unpackSize64` argument is a large unsigned integer, so `malloc` fails to allocate this memory and ASAN instigates a crash. If we can control this size argument, this could be a potential bug.

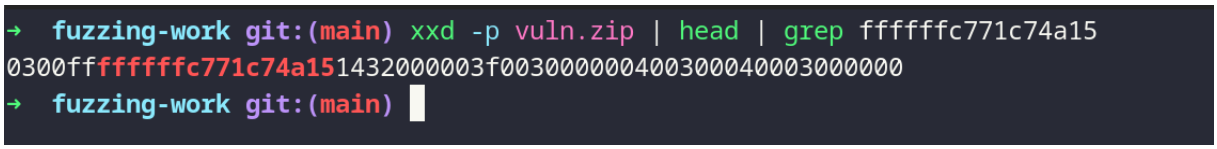

Figure 9: Size in File header

We analyzed the input file in more detail and found `unpackSize64` (in little endian) within the header of the file. So we can attempt to modify this offset within the header and control the amount of memory malloc'd. But, this is not an outright segfault since CPP error handling accounts for this and throws an exception, which is caught by the `p7zip` error handler.

However, this is a potential bug if combined with static/taint analysis so see if we could perform a possible overflow due to some arithmetic operations performed on this size argument.

Archive command

We also fuzzed the `archive` command of `7zz`. For this, we initially chose a corpus of *.txt* files, fuzzing the `a` command-line argument (along with `-y` to avoid user input hangs). But, we could not directly fuzz this since `afl-fuzz` only supports one cli argument and `a` can be used with multiple files with the following syntax:

```
7zz a files.zip file1.txt file2.txt file3.txt
```

We concluded that it would not be sufficient to just archive one file so we decided to create a harness which would allow for multiple file as arguments.

Harness

Our approach for the harness is as follows:

- As `afl-fuzz` allows for only one input to the target binary, our harness would accept one file name as argument.
- Contents of this input file would be divided into chunks of 1000 bytes and one new file will be created for each chunk.
- With a maximum limit of 15 files, these set of created files would be passed in as arguments to `7zz`.

We created a new *MainAr.cpp* with the `main` function being replaced by our harness which would mutate the input and pass it into *argv* of `main_7zz`, the original `main` function of p7zip. This way, we can fuzz the archive command with multiple files and potentially discover more bugs.

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 $(AFL_FUZZ)
    -M main-afl-$(HOSTNAME) -t 30000 -i in -o out -- $(BIN_AFL_HARNESS)
    @@
```

Our `afl-fuzz` command just passes in a cli argument to the harness.

Archive Fuzzing Results

We also ran these fuzzers using multiple cores, but for a shorter timeframe of around 2 days. We did not notice any crashes or hangs in any of the fuzzers. Also, we were not able to observe the fuzzers for a longer duration and record more insights as the cybersec room servers had been shut down.

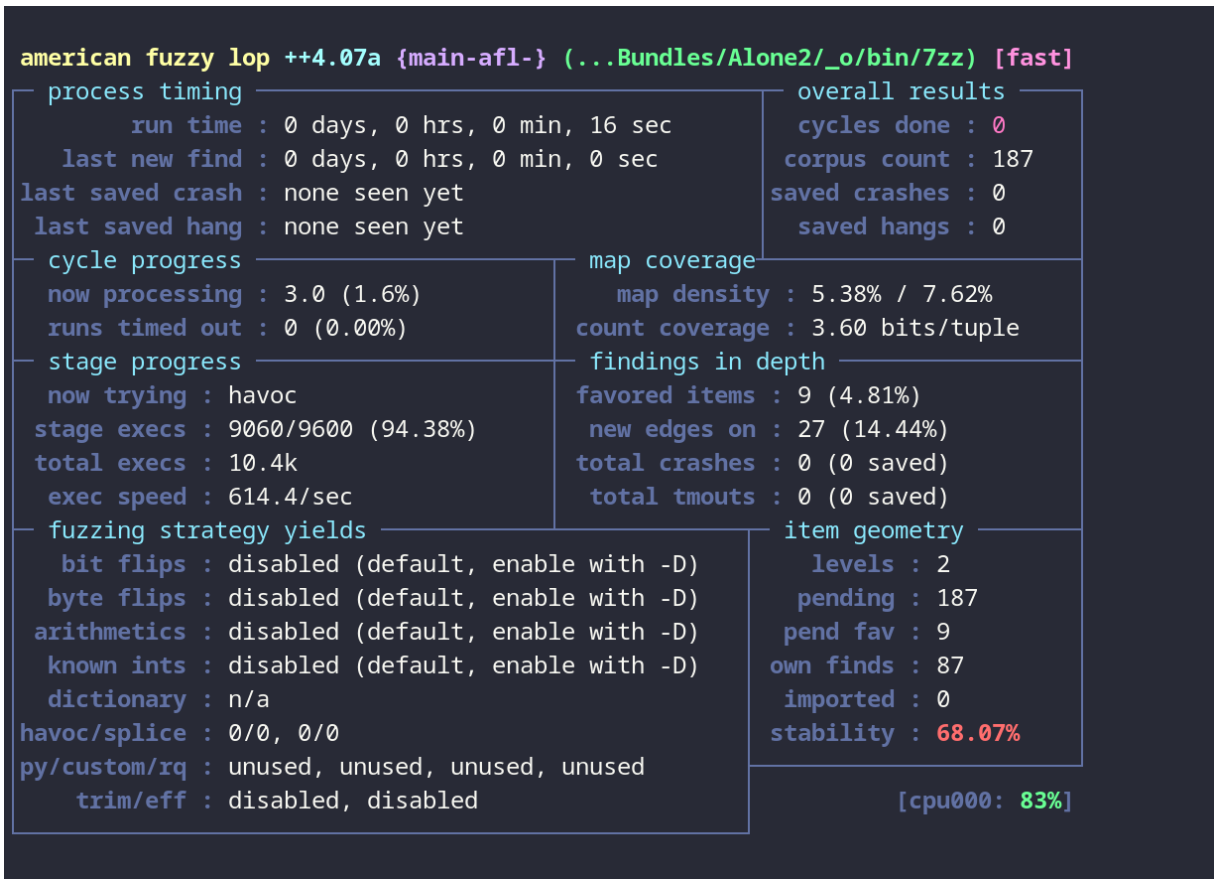

Figure 10: Main AFL Fuzzer

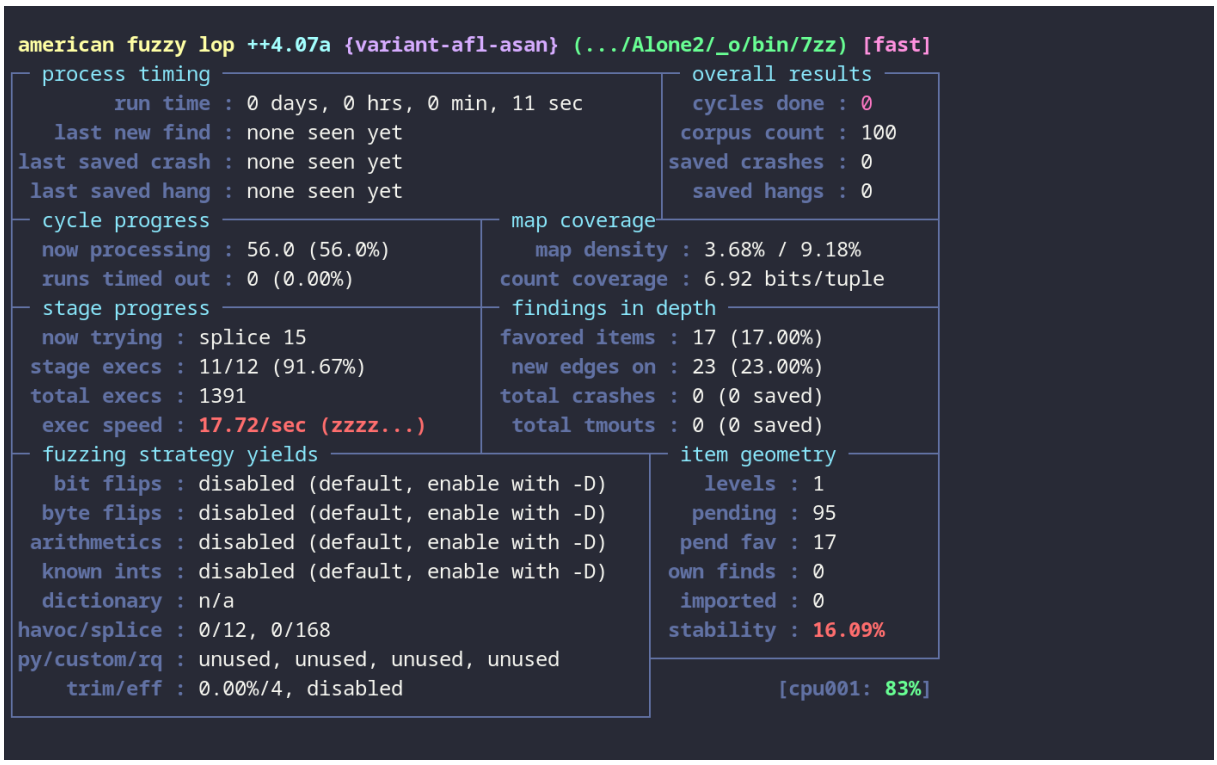

Figure 11: ASAN Variant Fuzzer

OSS-Fuzz and State of Fuzzing `p7zip`

When we were researching good ways to fuzz `p7zip`, we found a pull request on the repository that asks the maintainer if they would like `p7zip` to be part of a wider fuzzing initiative by Google called OSSFuzz which fuzzes popular open-source projects for free. Although this pull request was never merged, it hinted to us that it was possible that some organizations are fuzzing this project regularly.

Interestingly, we also found that the repository itself has a fuzzer set up for some of its codecs (compression algorithms).

For example, we found a dependency zstd, which is a real-time compression library build by Meta. This has a fuzzing subdirectory set up for fuzzing this library.

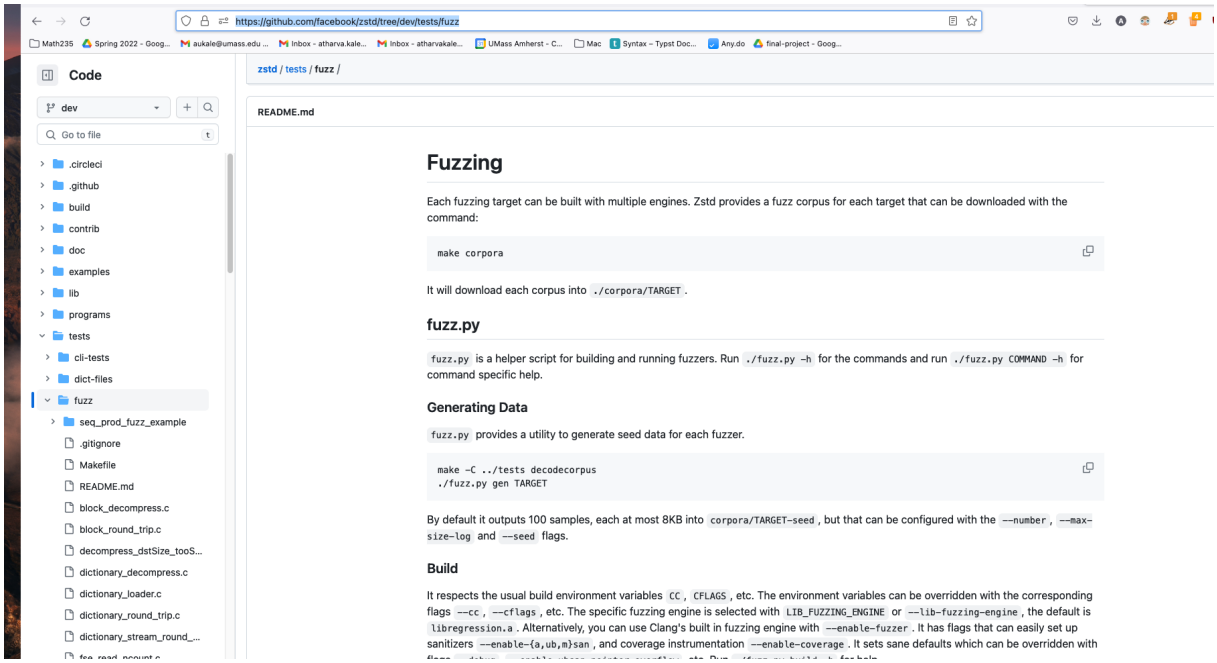

Figure 12: zstd Fuzzing Setup

We found that they are using libFuzzer, a coverage guided fuzzing library by LLVM. They also use AFL in tandem with their setup.

We found really interesting design choices in their fuzzing setup. One such choice was to incorporate unit-fuzzing, which is fuzzing small features of the target.

This was evident in a **simple LLVM harness** they wrote:

zstd / tests / fuzz / simple_compress.c

Code Blame 60 lines (51 loc) · 1.87 KB

```c
#include "zstd.h"
#include "zstd_errors.h"
#include "zstd_helpers.h"
#include "fuzz_data_producer.h"
#include "fuzz_third_party_seq_prod.h"

static ZSTD_CCtx *cctx = NULL;

int LLVMFuzzerTestOneInput(const uint8_t *src, size_t size)
{
    FUZZ_SEQ_PROD_SETUP();

    /* Give a random portion of src data to the producer, to use for
    parameter generation. The rest will be used for (de)compression */
    FUZZ_dataProducer_t *producer = FUZZ_dataProducer_create(src, size);
    size = FUZZ_dataProducer_reserveDataPrefix(producer);

    size_t const maxSize = ZSTD_compressBound(size);
    size_t const bufSize = FUZZ_dataProducer_uint32Range(producer, 0, maxSize);

    int const cLevel = FUZZ_dataProducer_int32Range(producer, kMinClevel, kMaxClevel);

    if (!cctx) {
        cctx = ZSTD_createCCtx();
        FUZZ_ASSERT(cctx);
    }

    void *rBuf = FUZZ_malloc(bufSize);
    size_t const ret = ZSTD_compressCCtx(cctx, rBuf, bufSize, src, size, cLevel);
    if (ZSTD_isError(ret)) {
        FUZZ_ASSERT(ZSTD_getErrorCode(ret) == ZSTD_error_dstSize_tooSmall);
    }
    free(rBuf);
    FUZZ_dataProducer_free(producer);
#ifndef STATEFUL_FUZZING
    ZSTD_freeCCtx(cctx); cctx = NULL;
#endif
    FUZZ_SEQ_PROD_TEARDOWN();
    return 0;
}
```

Figure 13: Simple Compress Harness

In this harness, they first produce an input using the libFuzzer's input generator. They then call into the function they are fuzzing-`ZSTD_compressCCtx`.

```
262
263  /*! ZSTD_compressCCtx() :
264   *  Same as ZSTD_compress(), using an explicit ZSTD_CCtx.
265   *  Important : in order to behave similarly to `ZSTD_compress()`,
266   *  this function compresses at requested compression level,
267   *  __ignoring any other parameter__ .
268   *  If any advanced parameter was set using the advanced API,
269   *  they will all be reset. Only `compressionLevel` remains.
270   */
271  ZSTDLIB_API size_t ZSTD_compressCCtx(ZSTD_CCtx* cctx,
272                                       void* dst, size_t dstCapacity,
273                                 const void* src, size_t srcSize,
274                                       int compressionLevel);
275
```

Figure 14: `ZSTD_compressCCtx()`

This seems to be a major function in their library that handles compression of certain input frames.

These more advanced approaches seem more well suited to a project of this size, as fuzzing with a small harness may provide a higher exec speed.

Static Analysis

We used three main tools for static analysis: CppCheck, CodeQl, and Flawfinder.

- CppCheck relies on multiple integrated tools for analyzing source; focuses on detecting undefined behavior
- CodeQL abstracts the source to a QL-language IR, which can then be queried
- Flawfinder is a syntatic analysis engine that scans for vulnerable code patterns

We used three seperate tools because static analysis tools are significantly more effective at finding vulnerabilities when combined*

CPPCheck/Flawfinder in particular when run alone struggle to identify vulnerabilities, according to Lip et al. 2022 empirical study (preprint)

CppCheck

CppCheck tagged a large number of errors, but most were false positives.

One such error reported undefined bit shifting:

2594	shiftTooManyBits	758	error	Shifting 32-bit value by 63 bits is undefined behaviour

Figure 15: Bit Shift False Positive

But this was just due to an innocuous macro:

```
#define GetUi64(p) (GetUi32(p) | ((UInt64)GetUi32(((const Byte *)(p)) + 4) << 32))
```

Figure 16: Macro

More promising was a potential null pointer bug:

390r-debugging-setup\p7zip\CPP\7zip\Archive\Zip\ZipHandlerOut.cpp

41	nullPointerRedundantCheck	476	warning	Either the condition 'password' is redundant or there is possible null pointer dereference: s++.
41	nullPointerArithmeticRedundantCheck	682	warning	Either the condition 'password' is redundant or there is pointer arithmetic with NULL pointer.

Figure 17: Nullptr Warning

But this was checked for in the source:

```
37  static bool IsSimpleAsciiString(const wchar_t *s)
38  {
39    for (;;)
40    {
41      wchar_t c = *s++;
42      if (c == 0)
43        return true;
44      if (c < 0x20 || c > 0x7F)
45        return false;
46    }
47  }
```

```
77  inline HRESULT StringToBstr(LPCOLESTR src, BSTR *bstr)
78  {
79    *bstr = ::SysAllocString(src);
80    return (*bstr) ? S_OK : E_OUTOFMEMORY;
81  }
```

CodeQL

CodeQL creates a database from source, which can be queried for dataflow analysis.

Running it against the default CPP queries produced nothing interesting:

```
ANALYSIS SUMMARY:

Hits = 500
Lines analyzed = 240937 in approximately 6.18 seconds (38963 lines/second)
Physical Source Lines of Code (SLOC) = 183216
Hits@level = [0] 131 [1]  41 [2] 439 [3]   9 [4]  11 [5]   0
Hits@level+ = [0+] 631 [1+] 500 [2+] 459 [3+]  20 [4+]  11 [5+]   0
Hits/KSLOC@level+ = [0+] 3.44402 [1+] 2.72902 [2+] 2.50524 [3+] 0.109161 [4+] 0.0600384 [5+]   0
Minimum risk level = 1
```

Figure 18: CodeQL analysis output

Next we worked on trying to find bugs associated with allocation functions. We designed a query to find all calls to malloc where the allocation size came from somewhere that had arithmetic operations on the way. This was only looking at local flow at first.

```
import cpp
import semmle.code.cpp.dataflow.DataFlow

from Function malloc, FunctionCall fc, Expr src
where malloc.hasGlobalName("malloc")
  and fc.getTarget() = malloc
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(fc.getArgument(0)))
  and src.toString().regexpMatch(".*([\\+\\/\\-\\=\\*]+).*")
select src
```

Figure 19: First codeQL Pass

This just looks for all instances of malloc, where the source has arithmetic operations.

We get 82 results from this pass, such as this one:

```
466  FSE_CTable* FSE_createCTable (unsigned maxSymbolValue, unsigned tableLog)
467  {
468      size_t size;
469      if (tableLog > FSE_TABLELOG_ABSOLUTE_MAX) tableLog = FSE_TABLELOG_ABSOLUTE_MAX;
470      size = FSE_CTABLE_SIZE_U32 (tableLog, maxSymbolValue) * sizeof(U32);
471      return (FSE_CTable*)malloc(size);
472  }
```

Figure 20: First codeQL Hit

But all of the hits were in the C section of the code, and we'd been looking at the cpp version. Where

the new/new[] operator replaces malloc for object initialization. But there was a wrapper function Alloc() which takes in a size parameter, so we rewrote the query to find all calls to this function

```
import cpp
import NewDelete
import semmle.code.cpp.Print


from
  Function alloc, FunctionCall fc, Expr src
where alloc.getName() = "Alloc"
  and fc.getTarget() = alloc
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(fc.getArgument(0)
  and src.toString().regexpMatch(".*([\\+\\/\\-\\=\\*]+)*.")

select src
```

Figure 21: Second codeQL Pass

Through tracking the basic dataflow into the `alloc()` sink by writing a query we were able to pin down the source of the bug that triggered the program crash (invalid size passed to the **new** operator):

```
HRESULT CDatabase::OpenXml(IInStream *inStream, const CHeader &h, CByteBuffer &xml)
{
  CUnpacker unpacker;
  return unpacker.UnpackData(inStream, h.XmlResource, h, this, xml, NULL);
}
```

Figure 22: `Size Bug source`

The `XMLResource` element is part of the header field of the file, which can theoretically be controlled. However, as noted, this triggers CPP error handling so is not a serious bug.

FlawFinder

FlawFinder is a purely syntactical engine, which means it doesn't do any control flow, data flow, etc. analysis. It simply scans for vulnerable code patterns.

As such, most warnings are not going to be particularly interesting:

```
HRESULT CDatabase::OpenXml(IInStream *inStream, const CHeader &h, CByteBuffer &xml)
{
  CUnpacker unpacker;
  return unpacker.UnpackData(inStream, h.XmlResource, h, this, xml, NULL);
}
```

Figure 23: `Flawfinder results`

However, there was one particular flag that caught our eyes, and it had to do with a specific use of `strcpy()` in WimHandler.cpp:

```
if ((unsigned)method < ARRAY_SIZE(k_Methods))
  strcpy(temp, k_Methods[(unsigned)method]);
else
  ConvertUInt32ToString((UInt32)(unsigned)method, temp);
```

Figure 24: `WimHandler strcpy()`

It was stated in the presentation that this was a potential segfault/buffer overflow. This was not correct, as `method` is cast as `unsigned`, and as such there is no risk of overflow with this specific `strcpy`.

Method is a part of the WIM file header and is derived from the compression flags. It would be interesting if this was the source of a bug, but alas.

We examined the `ConvertUInt32ToString()` function as well, just in case, but it appears to be robust.

Challenges Faced

Working with a large C/C++ codebase

It was our first exposure to working with a large C/C++ codebase. Although neatly organized at first glance, the project quickly turned into a codebase with a bunch of build scripts. Our first challenge was to figure out how to get a debug and release build going. Documentation on the dependencies was sparse, so this involved a compile-and-fail cycle to find all the dependencies for our systems. However, this experience provided us with great insight on how real world C++ projects are build, and gave us some direction on how to design such a codebase for a project in the future.

Bug Hunting False Positives

Another challenge for this target had to do with the static analysis portion. We ran our target through three utilities, and in total there were thousands of reported errors/warnings. Given the size and complexity of the codebase, bug hunting false positives was a chore.

Next Steps

- We noticed slow fuzzing especially with the `harness` so *Snapshot fuzzing* could help in making the process more effective and gain more coverage.
- Writing more in depth queries in CodeQL to perform more comprehensive control-flow analysis.
- Perform *Data Flow Analysis* in tandem with simpler syntactic analysis tools/dynamic analysis using CodeQL or similar.
- Delve into the compile process in order to emit LLVM bitcode and consequently the IR so we can write passes on the target.
- Find more corpora for the `archive` command as only *.txt* files might not be enough to discover bugs.
- Set up variant analysis based on other commonly found bugs in open-source projects.