Final Writeup

p7zip

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

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 Repository
- Overview of the Target
 - Code Layout
 - Coding Observations
 - Analyzing Target Features
- Automated Analysis
 - Fuzzing
 - * How was it set up
 - * Results etc...
 - Static Analysis
 - ٠..
- · Challenges Faced
 - ...
- · Next Steps

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

Call Graph for extract command

Call Graph for archive command

Coding Observations

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:
2
      rm -rf in_raw fuzzing-corpus && mkdir in_raw
3
      git clone -n --depth=1 --filter=tree:0 git@github.com:strongcourage
4
          /fuzzing-corpus.git
      cd fuzzing-corpus && git sparse-checkout set --no-cone zip gzip/go-
          fuzz lrzip jar && git checkout
6
      mv fuzzing-corpus/zip/go-fuzz/* in_raw
      mv fuzzing-corpus/jar/* in_raw
7
      mv fuzzing-corpus/gzip/go-fuzz/* in_raw
8
9
      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.

```
1 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.

```
1 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

```
1 afl:
       rm -rf $(BIN_AFL)
2
3
       git clone $(GH_URL) $(BIN_AFL)
4
       cp 7zz-makefiles/$(BIN_DEFAULT).mak $(BIN_AFL)/CPP/7zip/7zip_gcc.
           mak
5
       cd $(BIN_AFL)/CPP/7zip/Bundles/Alone2 && CC=$(AFL_CC) CXX=$(AFL_CXX
           ) make -f makefile.gcc
6
7 afl-asan:
8
       rm -rf $(BIN_AFL_ASAN)
       git clone $(GH_URL) $(BIN_AFL_ASAN)
9
       cp 7zz-makefiles/$(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
12
13 afl-msan:
14
       rm -rf $(BIN_AFL_MSAN)
       git clone $(GH_URL) $(BIN_AFL_MSAN)
15
16
       cp 7zz-makefiles/$(BIN_AFL_MSAN).mak $(BIN_AFL_MSAN)/CPP/7zip/7
           zip_gcc.mak
17
       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
18
19 afl-tsan:
20
       rm -rf $(BIN_AFL_TSAN)
21
       git clone $(GH_URL) $(BIN_AFL_TSAN)
       cp 7zz-makefiles/$(BIN_AFL_TSAN).mak $(BIN_AFL_TSAN)/CPP/7zip/7
22
           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.

```
1 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).

```
1 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.

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.

```
american fuzzy lop ++4.07a {main-afl-} (...Bundles/Alone2/_o/bin/7zz) [fast]
 process timing
                                                         overall results
       run time : 5 days, 0 hrs, 16 min, 37 sec
                                                        corpus count : 11.3k
  last new find : 0 days, 0 hrs, 0 min, 3 sec
last saved crash : none seen yet
 last saved hang : 0 days, 0 hrs, 13 min, 57 sec
                                                        saved hangs : 40
 cycle progress
                                       oxdot map coverage^{
m J}
 now processing : 11.3k.0 (99.7%)
                                        map density : 1.01% / 6.76%
count coverage : 5.10 bits/tuple
 runs timed out : 0 (0.00%)
 stage progress
                                         findings in depth
                                         favored items : 1016 (8.97%)
 now trying : havoc
stage execs : 4446/8000 (55.58%)
total execs : 207M
                                         new edges on : 1830 (16.16%)
                                        total crashes : 0 (0 saved)
total execs : 207M
                                         total tmouts : 293 (0 saved)
 exec speed : 667.6/sec
 fuzzing strategy yields -
                                                       - item geometry
  bit flips : disabled (default, enable with -D)
 byte flips : disabled (default, enable with -D)
arithmetics : disabled (default, enable with -D)
                                                        pend fav : 9
 known ints : disabled (default, enable with -D)
                                                       own finds : 11.0k
dictionary : n/a
havoc/splice : 6716/76.6M, 4317/130M
py/custom/rq : unused, unused, unused, unused
   trim/eff : disabled, disabled
                                                               [cpu000: 83%]
```

Figure 1: Main AFL Fuzzer

```
american fuzzy lop ++4.07a {variant-afl-asan} (.../Alone2/_o/bin/7zz) [fast]
  process timing
                                                             - overall results -
  run time : 5 days, 0 hrs, 20 min, 29 sec
last new find : 0 days, 0 hrs, 5 min, 0 sec
last saved crash : 0 days, 3 hrs, 27 min, 15 sec
                                                           saved crashes : 289
last saved hang : 0 days, 0 hrs, 14 min, 18 sec
 cycle progress
 now processing : 7313.63 (89.7%)
runs timed out : 0 (0.00%)
                                               map density : 5.57% / 27.89%
                                          count coverage : 5.67 bits/tuple
 stage progress
now trying : splice 15
stage execs : 5/12 (41.67%)
                                           favored items : 843 (10.34%)
                                            new edges on : 1578 (19.36%)
 total execs : 9.02M
                                           total crashes : 11.4k (289 saved)
 exec speed : 35.04/sec (slow!)
                                           total tmouts : 25 (0 saved)
  fuzzing strategy yields -
                                                            item geometry
  bit flips : disabled (default, enable with -D)
 byte flips : disabled (default, enable with -D)
                                                            pending : 3640
arithmetics : disabled (default, enable with -D)
known ints : disabled (default, enable with -D)
                                                            imported: 6369
havoc/splice : 616/1.56M, 1282/4.63M
pv/custom/rg : unused, unused, unused, unused
    trim/eff : 6.78%/2.75M, disabled
                                                                    [cpu001: 50%]
```

Figure 2: 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

```
Extracting archive: vuln.zip

==39559==ERROR: AddressSanitizer: requested allocation size 0x154ac771c7ffffff (0x154ac771c8001000 after adjustments for alignment, red zones etc.) exceeds maximum supported size of 0x10000000000 (thread T0)

#0 0x7fa6c50d9f98 in operator new[](unsigned long) (/lib64/libasan.so.8+0xd9f98) (BuildId: d34ee7d544aadb28cdcd8dd99198ee7130a1fe4d)
#1 0xf4937d (/home/lifewhiz/projects/revEng/390r-debugging-setup/7zz_afl_asan/CPP/7zip/Bundles/Alone2/_o/bin/7zz+0xf4937d) (BuildId: 30fd98054d6b9d16484801a99d3e4e5e9e187262)

#2 0xf49868 (/home/lifewhiz/projects/revEng/390r-debugging-setup/7zz_afl_asan/CPP/7zip/Bundles/Alone2/_o/bin/7zz+0xf49868) (BuildId: 30fd98054d6b9d16484801a99d3e4e5e9e187262)

#3 0xedfe25 (/home/lifewhiz/projects/revEng/390r-debugging-setup/7zz_afl_asan/CPP/7zip/Bundles/Alone2/_o/bin/7zz+0xedfe25) (BuildId: 30fd98054d6b9d16484801a99d3e4e5e9e187262)

=39559==HINT: if you don't care about these errors you may set allocator_may_return_null=1

SUMMARY: AddressSanitizer: allocation-size-too-big (/lib64/libasan.so.8+0xd9f98) (BuildId: d34ee7d544aadb28cdcd8dd99198ee7130a1fe4d) in operator new[](unsigned long)

=39559==ABORTING

* fuzzing-work git:(main)
```

Figure 3: 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 4: 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.

```
void Alloc(size_t size)
          if (size != _size)
  50
             Free();
               _items = new T[size];
00:000
         rsp 0x7fffffffbb70 ← 0x154ac771c7ffffff
01:0008
              \underline{0x7ffffffbb78} \rightarrow \underline{0x7ffffffbd00} \leftarrow \underline{0x0}
02:0010
         rbp 0x7ffffffbb80 → 0x7fffffffbbf0 → 0x7fffffffbb00 → 0x7fffffffbb0 → 0x7fffffffbbf0 ← ...
              0x7ffffffbb88 → 0x578e16 ← mov edi, 0x28
03:0018
              0x7ffffffbb90 → 0x7ffffffbd00 ← 0x0
04:0020
              <u>0x7ffffffbb98</u> → 0x76d2a8 ← 0x0
05:0028
              0x7ffffffbba0 → 0x7fffffffbd80 → 0x4034b5000010ae7
06:0030
07:0038
              \underline{0x7ffffffbba8} \rightarrow \underline{0x7ffffffbdd0} \leftarrow 0x800000000
                                                                          -r backtrace 1-
                0x45bc1c
   f 1
                0x578e16
   f 2
               0x57abcd
   f 3
                0x56ba3f
                0x64612e
                0x6495ab
                0x64a9f1 CArc::OpenStreamOrFile(COpenOptions&)+363
  = 0x154ac771c7ffffff
```

Figure 5: 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.

```
→ fuzzing-work git:(main) xxd -p vuln.zip | head | grep ffffffc771c74a15
0300ffffffffc771c74a151432000003f003000000400300040003000000
→ fuzzing-work git:(main)
```

Figure 6: 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:

```
1 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 a fl-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.

```
1 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.

Results

TODO

OSS-Fuzz

Static Analysis

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.

Real world projects

Next Steps