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 archive encryption (*SHA256*, *AES*, *RAR5*, etc…)

Code Layout

Coding Observations

Target Features

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). We also fuzzed `archive` TODO

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 2 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 2 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.

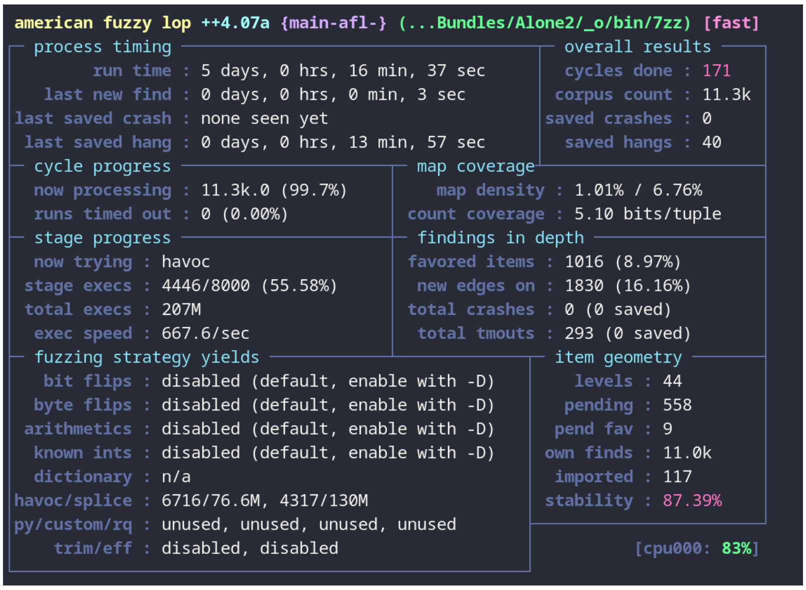
```
american fuzzy lop ++4.07a {main-afl-} (...Bundles/Alone2/_o/bin/7zz) [fast]
─ process timing ─────────────────────────────────────┬─ overall results ──
        run time : 5 days, 0 hrs, 16 min, 37 sec       │  cycles done : 171
   last new find : 0 days, 0 hrs, 0 min, 3 sec         │ corpus count : 11.3k
last saved crash : none seen yet                       │saved crashes : 0
 last saved hang : 0 days, 0 hrs, 13 min, 57 sec       │  saved hangs : 40
─ cycle progress ─────────────────────┬─ map coverage─┴────────────────────
  now processing : 11.3k.0 (99.7%)    │    map density : 1.01% / 6.76%
  runs timed out : 0 (0.00%)          │ count coverage : 5.10 bits/tuple
─ stage progress ─────────────────────┼─ findings in depth ────────────────
  now trying : havoc                  │ favored items : 1016 (8.97%)
 stage execs : 4446/8000 (55.58%)     │  new edges on : 1830 (16.16%)
 total execs : 207M                   │ total crashes : 0 (0 saved)
  exec speed : 667.6/sec              │  total tmouts : 293 (0 saved)
─ fuzzing strategy yields ────────────┴─────────────┬─ item geometry ──────
   bit flips : disabled (default, enable with -D)   │    levels : 44
  byte flips : disabled (default, enable with -D)   │   pending : 558
 arithmetics : disabled (default, enable with -D)   │  pend fav : 9
  known ints : disabled (default, enable with -D)   │ own finds : 11.0k
  dictionary : n/a                                  │  imported : 117
havoc/splice : 6716/76.6M, 4317/130M                │ stability : 87.39%
py/custom/rq : unused, unused, unused, unused       ├──────────────────────
    trim/eff : disabled, disabled                   │          [cpu000: 83%]
```

Figure 1: Main AFL Fuzzer

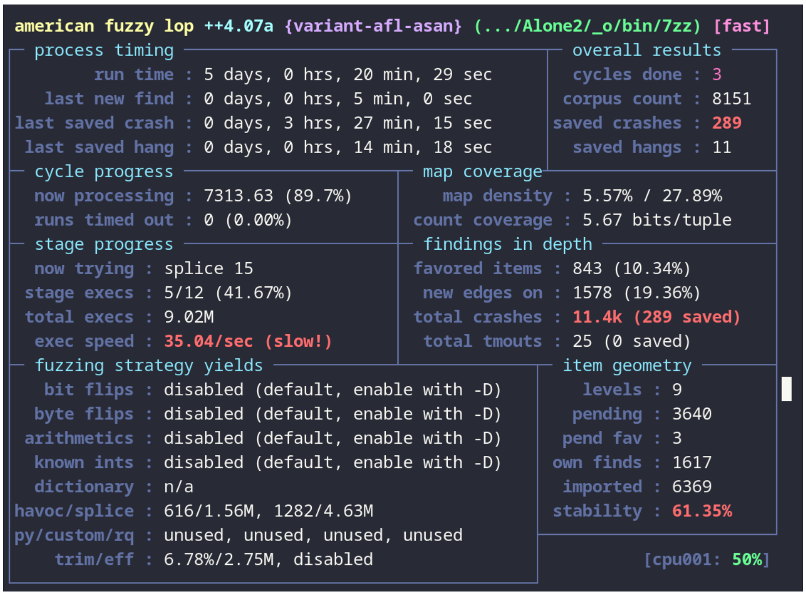
```
american fuzzy lop ++4.07a {variant-afl-asan} (.../Alone2/_o/bin/7zz) [fast]
─ process timing ─────────────────────────────────────┬─ overall results ──
        run time : 5 days, 0 hrs, 20 min, 29 sec       │  cycles done : 3
   last new find : 0 days, 0 hrs, 5 min, 0 sec         │ corpus count : 8151
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       │  saved hangs : 11
─ cycle progress ─────────────────────┬─ map coverage─┴────────────────────
  now processing : 7313.63 (89.7%)    │    map density : 5.57% / 27.89%
  runs timed out : 0 (0.00%)          │ count coverage : 5.67 bits/tuple
─ stage progress ─────────────────────┼─ findings in depth ────────────────
  now trying : splice 15              │ favored items : 843 (10.34%)
 stage execs : 5/12 (41.67%)          │  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)   │    levels : 9
  byte flips : disabled (default, enable with -D)   │   pending : 3640
 arithmetics : disabled (default, enable with -D)   │  pend fav : 3
  known ints : disabled (default, enable with -D)   │ own finds : 1617
  dictionary : n/a                                  │  imported : 6369
havoc/splice : 616/1.56M, 1282/4.63M                │ stability : 61.35%
py/custom/rq : 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

Archive command

Harness

Results

Static Analysis

Challenges Faced

Next Steps