Checkpoint 1

p7zip

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

2023-04-12

Checkpoint 1

Contents:

Overview of the Target

Debug Environment

How to build the target

Pre-requisites:

1. CMake

Step 1:

```
git clone git@github.com:jinfeihan57/p7zip.git
```

We clone jinfeihan57's repo, which is a *Linux port* for the 7zip Windows utility. The port is *fully compliant* with the Windows equivalent, and supports all the same formats.

Step 2:

```
cp 7zip_gcc_dbg.mak p7zip/CPP/7zip/7zip_gcc.mak
```

We created a custom `Makefile` that patches the original build script to include debug flags. We copy the patch into the correct directory.

Step 3:

```
cd p7zip/CPP/7zip/Bundles/Alone2 && make -f makefile.gcc && cd -
```

We build the `7zz` tool, which is the primary binary from the project, which supports archiving and extracting the most number of formats.

Step 4:

```
PATH=$PATH:$PWD/p7zip/CPP/7zip/Bundles/Alone2/_o/bin
```

For development purposes, we update the current terminal session's `PATH` to include the path to the `7zz` binary.

Experiment with the Target

```
➜ 1-checkpoint git:(main) ✗ 7zz -h

7-Zip (z) 22.00 ZS v1.5.2 (x64) : Copyright (c) 1999-2022 Igor Pavlov : 2022-06-15
 64-bit locale=en_US.UTF-8 Threads:16

Usage: 7zz <command> [<switches>...] <archive_name> [<file_names>...] [@listfile]

<Commands>
  a : Add files to archive
  b : Benchmark
  d : Delete files from archive
  e : Extract files from archive (without using directory names)
  h : Calculate hash values for files
  i : Show information about supported formats
  l : List contents of archive
  rn : Rename files in archive
  t : Test integrity of archive
  u : Update files to archive
  x : eXtract files with full paths
```

Figure 1: List of commands

Simple tests

In the `playground` directory, we have some sample files setup for basic tests.

```
1 cd playground
2 7zz a files.zip file1.txt file2.txt
3 7zz e files.zip -ofiles_extracted
```

Target analysis

File format

```
➜ bin git:(master) ✗ file 7zz
7zz: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.
2, BuildID[sha1]=02bfdb94fd86cb3d5e02ceaff225f37714a05ad3,
 for GNU/Linux 3.2.0, with debug_info, not stripped
➜ bin git:(master) ✗
```

Figure 2: File format

Mitigations

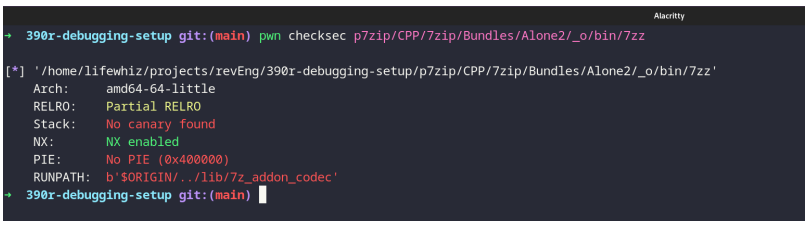

Figure 3: checksec mitigations

ROP Gadgets

Figure 4: List of ROP Gadgets

Shared Libraries

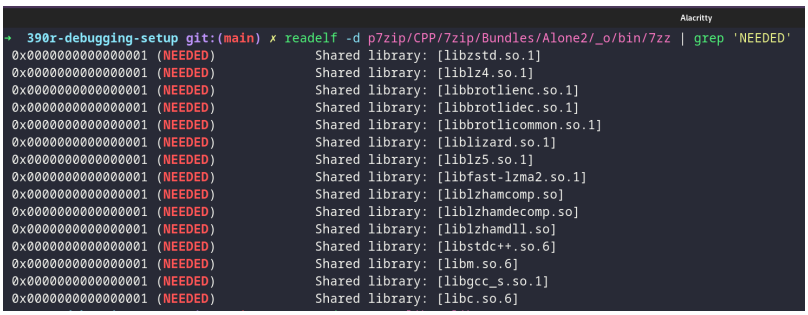

Figure 5: List of Shared Libraries

One Gadgets

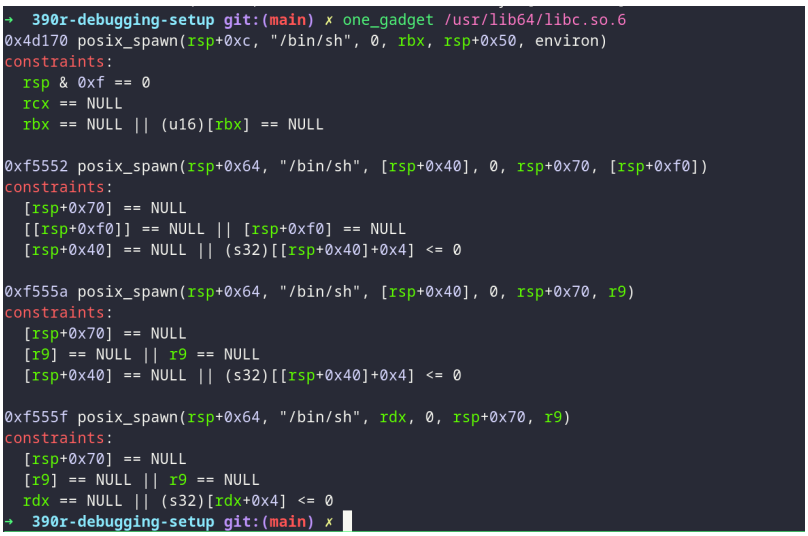

Figure 6: List of One Gadgets

Function call graph

The following can be used to analyze execution of the target and produce graphs. It requires `valgrind` and `kcachegrind` to be installed.

```
valgrind --callgrind-out-file=callgrind_vis2 --tool=callgrind 7zz e
    files.zip -ofiles_extracted
```

Use the `valgrind` command above to generate a `callgrind_vis2` file.

```
kcachegrind callgrind_vis2
```

Use the `kcachegrind` command to visualize the `callgrind_vis2`.

In the next two pages, we fine two function call graphs for the `archive` and `extract` subcommands.

Archive Command:

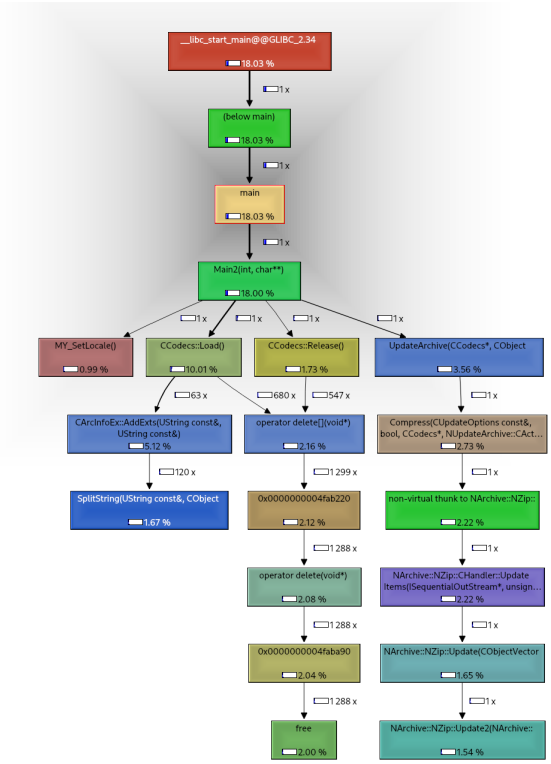

Figure 7: a subcommand

Extract Command:

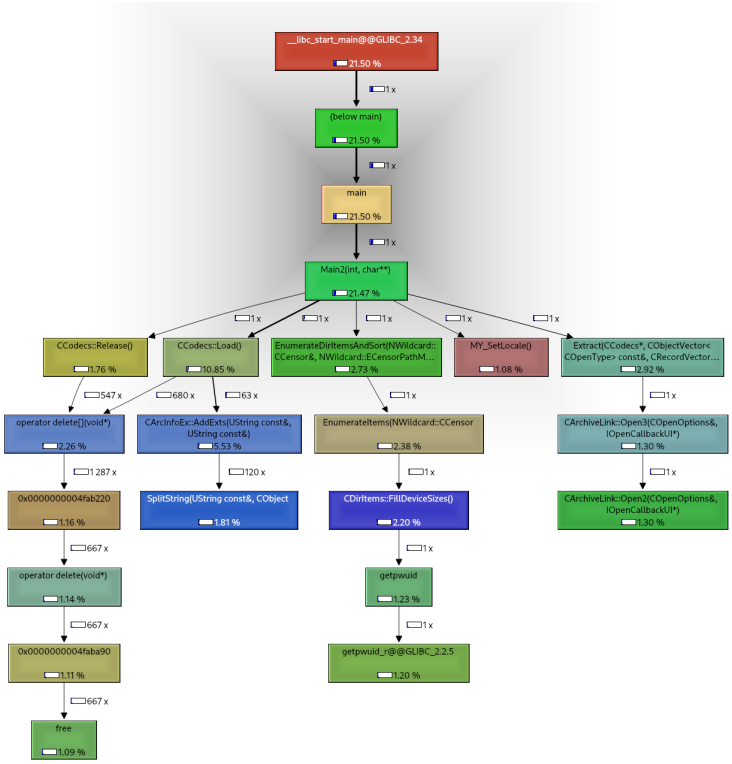

Figure 8: e subcommand

Mapping out the Target Code-Base

TODO

Future Plans

Fuzzing

```
git clone https://github.com/AFLplusplus/AFLplusplus
cd AFLplusplus
# sudo dnf -y install gcc-plugin-devel-12.2.1
# sudo dnf -y install llvm-devel
make all
sudo make install
CC=/usr/local/bin/afl-gcc-fast CXX=/usr/local/bin/afl-g++-fast make -f
    -B makefile.gcc
```

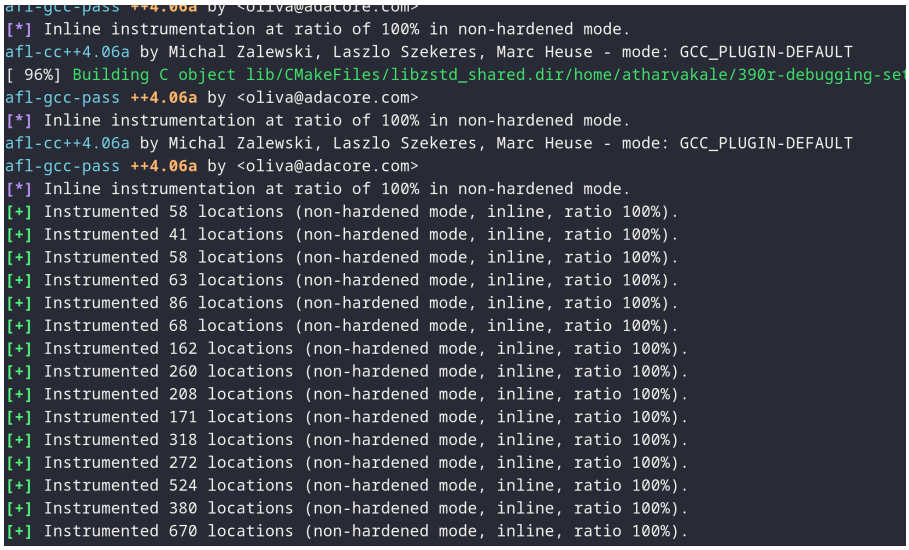

Figure 9: Compiling with AFL Source Code Instrumentation