Harnessing libarchive APIs

Introduction

In this lab assignment, I focused on harnessing fuzz testing techniques to evaluate the robustness and security of the libarchive APIs. Given its extensive support for various archive formats, libarchive presents a complex target with numerous potential attack vectors, making it an ideal candidate for such a comprehensive security assessment.

Methodology

The process began with developing a custom harness file, a controlled environment to execute specific libarchive API calls. This harness file is essential for directing the fuzzing tool toward meaningful interactions with the libarchive library, ensuring that the fuzzing process remains focused on the library's functionality.

1. AFL-Fuzzing source code instrumented libarchive with a Custom Harness.
2. AFL-Fuzzing the system's binary libarchive in QEMU mode.
3. AFL-fuzzing libArchive's libFuzzer harness using **afl_driver**.

For the first two modes, I wrote a harness file to fuzz some interesting APIs of libarchive. In the third mode, I used afl_driver to fuzz the harness file written for libFuzzer. The harness includes a comprehensive set of libArchive API functions to thoroughly test the library's capabilities for reading and processing archive files. The included functions and their purposes are as follows:

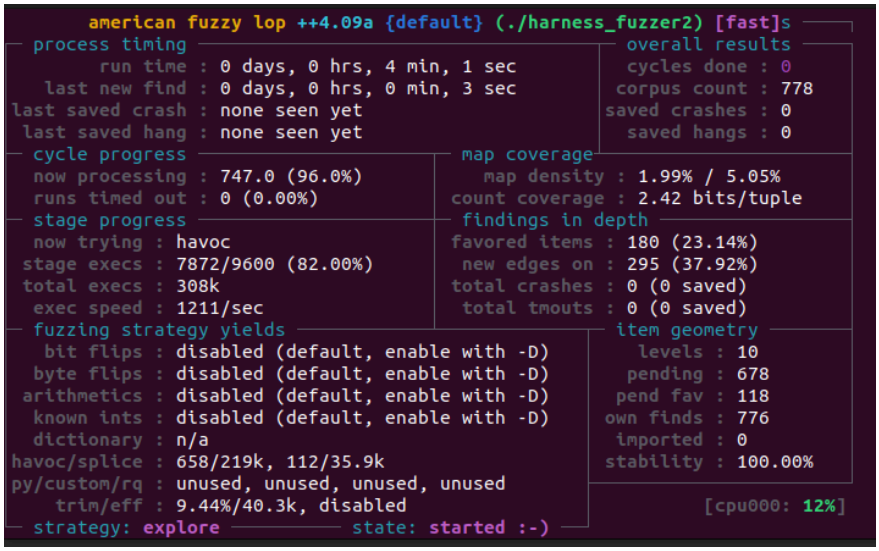
- **archive_read_new():** Initializes a new archive reading object. This is fundamental for any operation involving reading archives.
- **archive_read_support_filter_all()** and **archive_read_support_format_all():** These functions enable the archive object to read all supported compression and file formats, respectively, ensuring broad testing coverage.
- **archive_read_support_format_empty(),archive_read_support_format_raw(), and archive_read_support_format_gnutar():** Specifically enable support for empty, raw, and GNU tar formats to test libArchive's ability to handle these particular cases.
- **archive_read_set_options():** Configures the archive reading object with specific options to test libArchive's behavior under various conditions.

- **archive_read_open_memory():** Opens an archive for reading from memory, allowing the fuzzer to provide binary data as input directly.
- **archive_read_add_passphrase():** Adds a passphrase for reading encrypted archives, testing libArchive's encryption handling capabilities.
- **archive_read_next_header():** Iterates through the entries in the archive, essential for testing the library's ability to parse and navigate archive structures.
- **Accessing archive_entry properties (e.g., pathname, atime, birthtime, etc.):** Exercises the API for retrieving metadata about archive entries, essential for assessing libArchive's handling of entry properties.
- **archive_read_data():** Reads data from an archive entry, critical for testing data extraction functionality.
- **archive_read_has_encrypted_entries(),archive_read_format_capabilities(), archive_file_count(), archive_seek_data():** These functions test additional aspects of libArchive, including encryption detection, format capabilities inquiry, file counting within an archive, and data seeking capabilities.

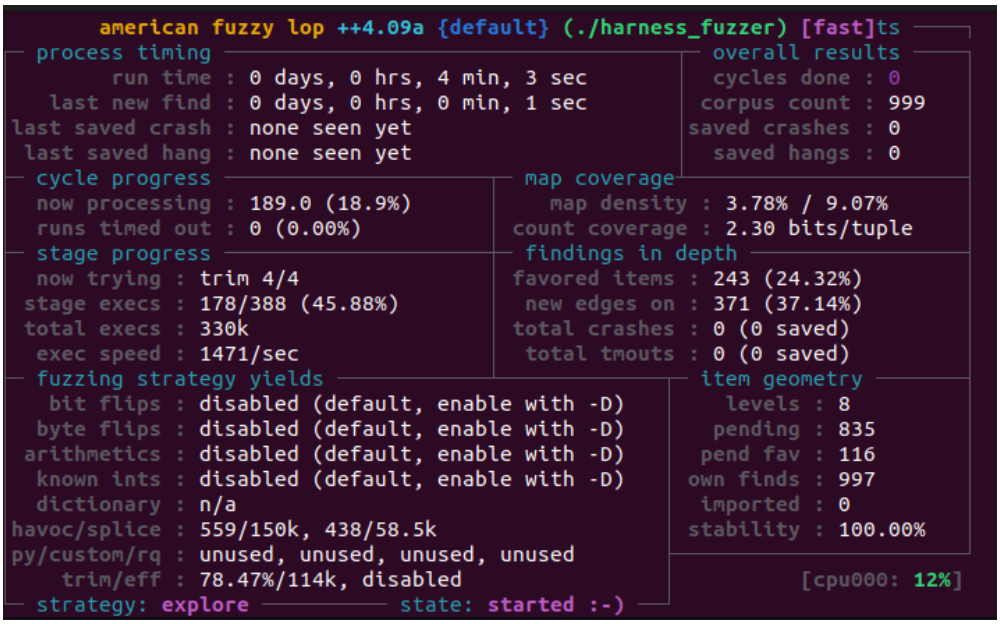
Findings and Analysis

I have not had any crashes in any of the modes. However, some AFL output metrics differed, such as the coverage path, new edges, and favored items. The output metrics are provided below:

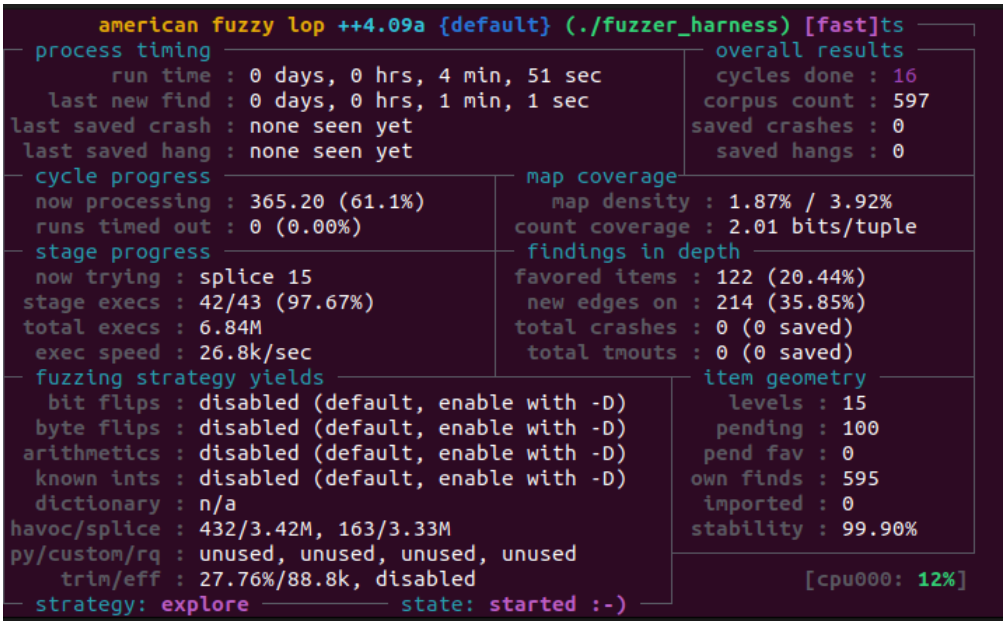
❖ AFL-Fuzzing source code instrumented libarchive with a Custom Harness:



❖ AFL-Fuzzing the system's binary libarchive in QEMU mode:



❖ AFL-fuzzing libArchive's libFuzzer harness using **afl_driver**:



Conclusion

In conclusion, the creation and execution of the fuzzing harness file for libarchive have significantly contributed to understanding the library's behavior under varied and unpredictable conditions. Lab 3 serves as a testament to the critical role of fuzz testing in modern software development and security analysis, offering valuable insights that

can guide future security efforts and the development of more resilient software systems.