Week 6A: Harnessing 1

Stefan Nagy
University of Utah
Reminders

- **Lab 2: Triage due today**
  - Submit your 1–3 page report on Canvas by 11:59PM

- **Lab 3: Harnessing released**
  - Details later in the lecture
Questions?
Harnessing
Recap: Fuzzing

Trace code coverage, Monitor execution

Inputs

Program

New coverage

Triggers bugs

No new coverage
Recap: Fuzzing

Trace code coverage, Monitor execution

New coverage

Triggers bugs

No new coverage

Inputs

Program
Harnessing

- **Definition:** making a program fuzzable
  - Pass input data to a program’s core logic
  - Skip functionality we don’t care about
  - Drop-in integration with fuzzers (e.g., AFL)

- **Types of harnesses**
  - Target a **single** function (libFuzzer-style)
  - Target **many** functions (AFL-style)

- **One of the most important (and difficult) parts of fuzzing**
  - Lots of domain expertise
  - Automating still an open problem
What makes a good harness?

- **Speed:** avoid re-executing uninteresting code
  - GUI initialization
  - Server/client setup routines
  - Weird developer-added obstacles
  - **Cycles are precious—don’t waste them!**
What makes a good harness?

- **Coverage**: invoke interesting parts of the code
  - Higher coverage = more thorough testing
  - Test hard-to-reach functionality
  - Measure and improve harnesses
  - **Coverage blindspots** = missed bugs!
What makes a good harness?

- **Correctness**: reset necessary program state
  - Global variable state
  - Stack state
  - Heap state
  - **State errors = false positive crashes!**

10 crashes: 1000 (1000 unique)
Harnessing Open- vs. Closed-source Code

**Open Source:**
- Publicly-available source codebase
- Achieves security by *transparency*

- Generally easy to harness
Harnessing Open- vs. Closed-source Code

**Open Source:**
- Publicly-available source codebase
- Achieves security by *transparency*
- Generally easy to harness

**Closed Source:**
- Distributed as a precompiled binary
- Opaque to everyone but its developer
- Far more difficult to harness
Basic Harnessing
What fuzzers expect

- **AFL:** program takes **command-line** input

```
./TargetBinary [arguments]  
```
What fuzzers expect

- **AFL**: program takes **command-line** input
  - Load, send, and store input data as **files**

```
./TargetBinary [arguments]  
```

```
outDirectory / .cur_input
```
What fuzzers expect

- **libFuzzer**: inputs as **buffered data**

```c
int LLVMFuzzerTestOneInput (const char *Data, long Size) {
}
```
What fuzzers expect

- **libFuzzer:** inputs as **buffered data**
  - Fuzzer alters **Data** and **Size** objects

```c
int LLVMFuzzerTestOneInput (const char *Data, long Size) {
    DoSomethingInterestingWithMyAPI (Data, Size);
    return 0;
}
```
Identifying suitable targets

- **For libraries** (e.g., libJPEG):
  - Find API’s “consumer” functions
  - Review documentation and figure out how to call it

- **For applications** (e.g., Adobe Reader):
  - Find what directly loads data (e.g., calls `fopen()`)
  - Skip-over irrelevant setup code
Example: libArchive

```c
int archive_read_open(struct archive *, void *client_data, archive_open_callback *, archive_read_callback *, archive_close_callback *);

int archive_read_open2(struct archive *, void *client_data, archive_open_callback *, archive_read_callback *, archive_skip_callback *, archive_close_callback *);

int archive_read_open_FILE(struct archive *, FILE *file);

int archive_read_open_fd(struct archive *, int fd, size_t block_size);

int archive_read_open_filename(struct archive *, const char *filename, size_t block_size);

int archive_read_open_memory(struct archive *, void *buff, size_t size);
```
Example: libArchive

```c
int archive_read_open(struct archive *, void *client_data, archive_open_callback *,
archive_read_callback *, archive_close_callback *);

int archive_read_open2(struct archive *, void *client_data, archive_open_callback *,
archive_read_callback *, archive_skip_callback *, archive_close_callback *);

int archive_read_open_FILE(struct archive *, FILE *file);

int archive_read_open_fd(struct archive *, int fd, size_t block_size);

int archive_read_open_filename(struct archive *, const char *filename, size_t block_size);

int archive_read_open_memory(struct archive *, void *buff, size_t size);
```
Example: libArchive

```c
int archive_read_open(struct archive *, void *client_data, archive_open_callback *,
                     archive_read_callback *, archive_close_callback *);
int archive_read_open2(struct archive *, void *client_data, archive_open_callback *,
                        archive_read_callback *, archive_skip_callback *, archive_close_callback *);
int archive_read_open_FILE(struct archive *, FILE *file);
int archive_read_open_fd(struct archive *, int fd, size_t block_size);
int archive_read_open_filename(struct archive *, const char *filename, size_t block_size);
int archive_read_open_memory(struct archive *, void *buff, size_t size);
```
Example: libArchive

archive_read_data()
Read data associated with the header just read. Internally, this is a convenience function that calls archive_read_data_block() and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
Return the next available block of data for this entry. Unlike archive_read_data(), the archive_read_data_block() function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
A convenience function that repeatedly calls archive_read_data_block() to skip all of the data for this archive entry. Note that this function is invoked automatically by archive_read_next_header2() if the previous entry was not completely consumed.

archive_read_data_into_fd()
A convenience function that repeatedly calls archive_read_data_block() to copy the entire entry to the provided file descriptor.
Example: libArchive

```c
archive_read_data()
   Read data associated with the header just read. Internally, this is
   a convenience function that calls archive_read_data_block() and
   fills any gaps with nulls so that callers see a single continuous
   stream of data.
archive_read_data_block()
   Return the next available block of data for this entry. Unlike
   archive_read_data(), the archive_read_data_block() function avoids
   copying data and allows you to correctly handle sparse files, as
   supported by some archive formats. The library guarantees that
   offsets will increase and that blocks will not overlap. Note that
   the blocks returned from this function can be much larger than the
   block size read from disk, due to compression and internal buffer
   optimizations.
archive_read_data_skip()
   A convenience function that repeatedly calls
   archive_read_data_block() to skip all of the data for this archive
   entry. Note that this function is invoked automatically by
   archive_read_next_header2() if the previous entry was not completely
   consumed.
archive_read_data_into_fd()
   A convenience function that repeatedly calls
   archive_read_data_block() to copy the entire entry to the provided
   file descriptor.
```
Example: libArchive

archive_read_data()
Read data associated with the header just read. Internally, this is a convenience function that calls archive_read_data_block() and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
Return the next available block of data for this entry. Unlike archive_read_data(), the archive_read_data_block() function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
A convenience function that repeatedly calls archive_read_data_block() to skip all of the data for this archive entry. Note that this function is invoked automatically by archive_read_next_header2() if the previous entry was not completely consumed.

archive_read_data_into_fd()
A convenience function that repeatedly calls archive_read_data_block() to copy the entire entry to the provided file descriptor.
Example: libArchive

archive_read_data()
Read data associated with the header just read. Internally, this is a convenience function that calls `archive_read_data_block()` and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
Return the next available block of data for this entry. Unlike `archive_read_data()`, the `archive_read_data_block()` function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
A convenience function that repeatedly calls `archive_read_data_block()` to skip all of the data for this archive entry. Note that this function is invoked automatically by `archive_read_next_header2()` if the previous entry was not completely consumed.

archive_read_data_into_fd()
A convenience function that repeatedly calls `archive_read_data_block()` to copy the entire entry to the provided file descriptor.

IF file header valid...

archive_read_open ( ... )
Example: libArchive

```c
int LLVMFuzzerTestOneInput(const char* buf, size_t len) {
    int ret,
    ssize_t r;
    struct archive *a = archive_read_new();

    Buffer buffer = {buf, len};
    archive_read_open(a, &buffer, NULL, NULL);

    std::vector<uint8_t> databuf (getpagesize(), 0);
    struct archive_entry *entry;

    while(1) {
        ret = archive_read_next_header(a, &entry);
        if (ret == ARCHIVE_EOF || ARCHIVE_FATAL)
            Break;
        while ((r = archive_read_data(a, databuf.data(),
            databuf.size())) > 0);
        if (r == ARCHIVE_FATAL)
            break;
    }
    archive_read_free(a);
    return 0;
}
```
Useful Heuristics

- **No source code?**
  - Capture a few *call traces* when running some valid inputs
  - Look for functions that call interesting functions (e.g., `fopen()`)
  - **Work backwards and figure out how to call them**
    - E.g., what validity-checking functions to call first
Useful Heuristics

- **Have source code?**
  - ... or a reasonably-precise decompilation?
  - Consider McCabe’s **Cyclomatic Complexity**
    - Complexity = \#Edges - \#Nodes + 2(#ConnectedComponents)
    - Example = 9 - 8 + 2(1) = 3
    - **Higher score = more interesting target**

Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity
Trial and error

- Writing harnesses is not a one-and-done task
  - There is always room for improvement

- Write an initial harness, test it, and reflect
  - Is the harness actually correct?
    - Yes: no (or few) false positive crashes
    - No: lots of false positive crashes
  - Am I executing interesting functionality?
    - Yes: coverage increases over time
    - No: coverage plateaus after some time
  - Study your target, and find ways improve your harnesses
Questions?
Lab 3: Harnessing

- **Assignment:** write your own AFL-friendly harness for libArchive
  - Read its documentation in: [https://linux.die.net/man/3/libarchive](https://linux.die.net/man/3/libarchive)

- **Create a harness that reads data from files**
  - What functions did you try?
  - What worked and what didn’t?

- **Deliverable:** a 1–3 page report detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
  - Submit your harness code in your report
  - Free to team up (max 3 students per group)
  - Submit one report per group

- **Linux environments are recommended**
  - See me if you don’t have one
Lab 3 Tips

- Read its documentation and get inspiration from others’ code
  - Understand the libArchive manpages
  - Look at how others (e.g., non-fuzzing projects) use its API

- Validate your results
  - Measure code coverage of the libArchive codebase
  - Look for increasing code coverage over time
Location Change

- Now in **MEB 3485** permanently on **Tuesdays and Thursdays**
  - No longer in Warnock