Week 6: Lecture A

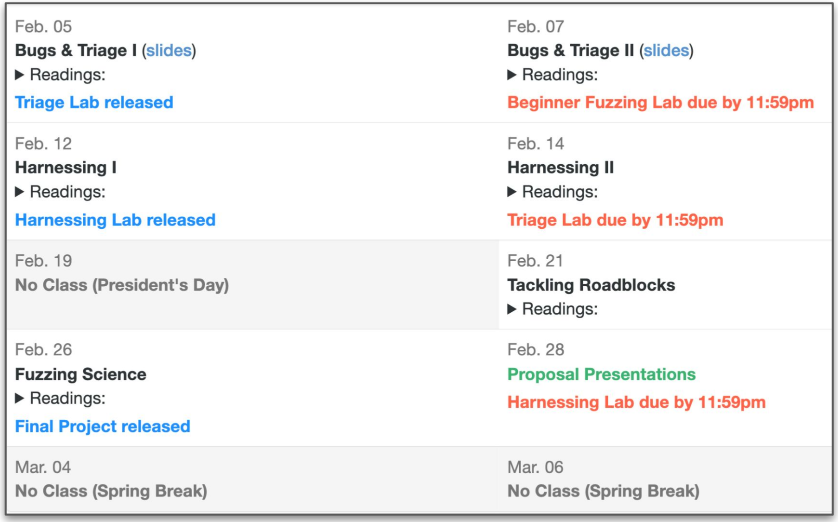
Harnessing I

Monday, February 12, 2024

Recap: Key Dates

Feb. 12	**Lab 3 released**
Feb. 14	**Lab 2 due**
Feb. 19	No class (President's Day)
Feb. 28	**Lab 3 due**
Feb. 28	**5-minute project proposals**
Mar. 04 & 06	No class (Spring Break)
Apr. 17 & 22	**Final project presentations**

cs.utah.edu/~snagy/courses/cs5963/schedule



Feb. 05 **Bugs & Triage I** (slides) ▶ Readings: **Triage Lab released**	Feb. 07 **Bugs & Triage II** (slides) ▶ Readings: **Beginner Fuzzing Lab due by 11:59pm**
Feb. 12 **Harnessing I** ▶ Readings: **Harnessing Lab released**	Feb. 14 **Harnessing II** ▶ Readings: **Triage Lab due by 11:59pm**
Feb. 19 **No Class (President's Day)**	Feb. 21 **Tackling Roadblocks** ▶ Readings:
Feb. 26 **Fuzzing Science** ▶ Readings: **Final Project released**	Feb. 28 **Proposal Presentations** **Harnessing Lab due by 11:59pm**
Mar. 04 **No Class (Spring Break)**	Mar. 06 **No Class (Spring Break)**

Recap: Lab 2 Overview

- **Assignment:** learn how to use AddressSanitizer (ASAN)
  - Read its documentation in https://clang.llvm.org/docs/AddressSanitizer.html

- **Replay the crashes you found in Lab 1 on an ASAN-instrumented binary**
  - Collect information on each crash
  - What do you observe?

- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)

- **Linux environments are recommended**
  - Use a VM if you don't have one!

Recap: Lab 2 Tips

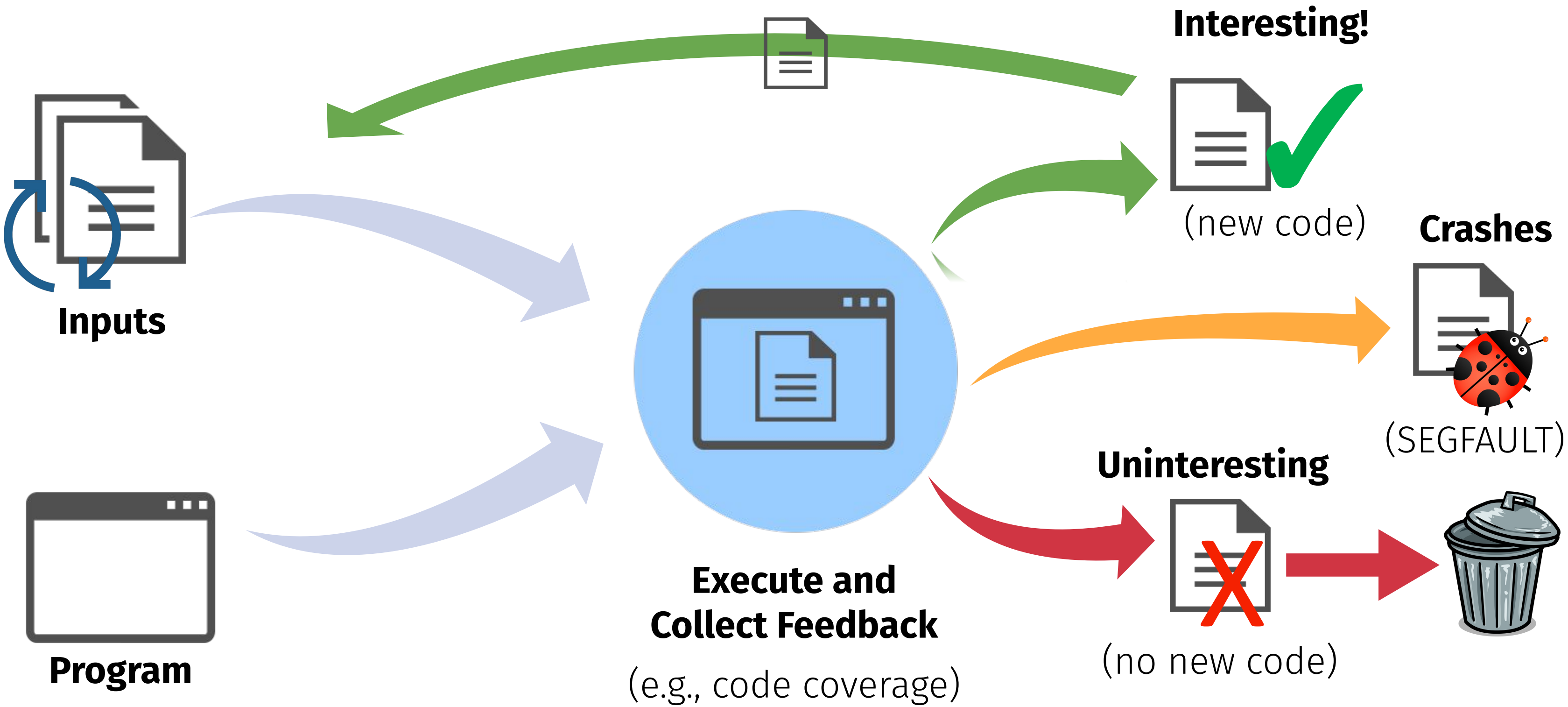
- **Re-run crashes on the ASAN instrumented binary**
  - Use Python to script collection of ASAN outputs
  - Do string post-processing to collect error types, crashing source line, etc.
  - Group and deduplicate crashes as you see fit

- **Didn't find any crashes in Lab 1?**
  - Try fuzzing fuzzgoat from https://github.com/fuzzstati0n/fuzzgoat
  - Should yield **lots** of crashes quickly

Questions?

Harnessing

Recap: Coverage-guided Fuzzing



Interesting!

(new code)

Crashes

(SEGFAULT)

Uninteresting

(no new code)

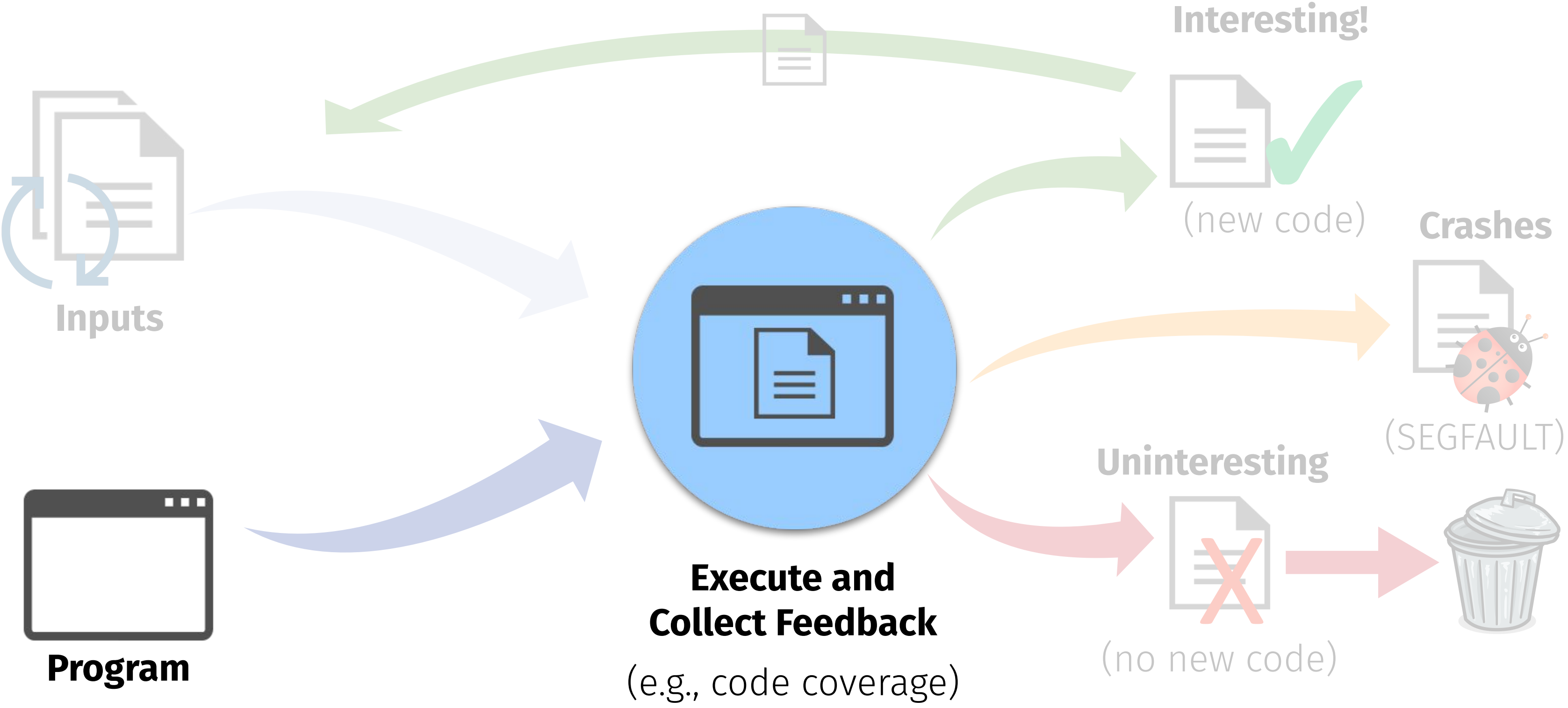
Inputs

Program

Execute and Collect Feedback

(e.g., code coverage)

Recap: Coverage-guided Fuzzing



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)

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)

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)

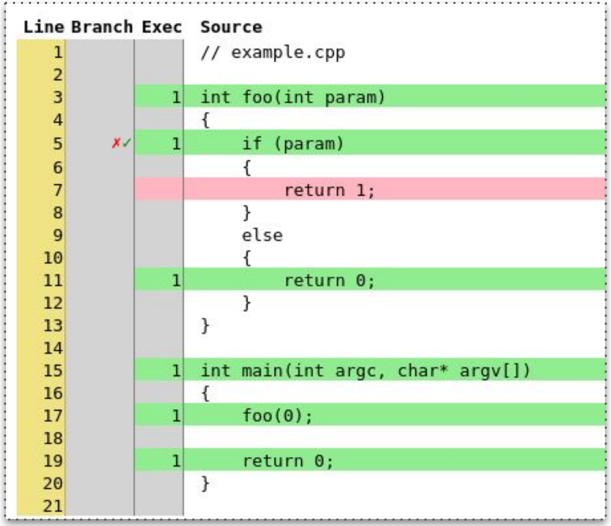
- **One of the most important (and difficult) parts of fuzzing**
  - Lots of domain expertise
  - Automating still an open problem

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!**

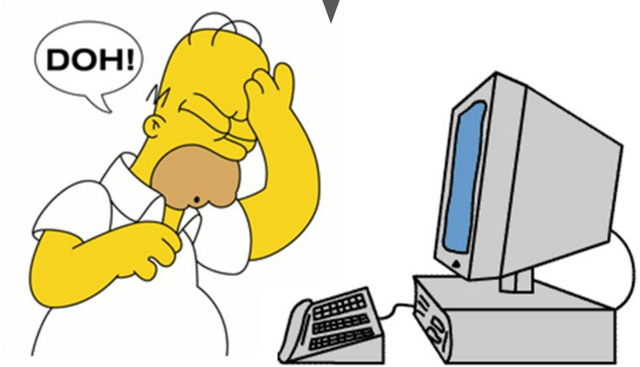


```
Line Branch Exec  Source
   1              // example.cpp
   2
   3           1  int foo(int param)
   4              {
   5      ✗✓   1      if (param)
   6                  {
   7                      return 1;
   8                  }
   9                  else
  10                  {
  11           1          return 0;
  12                  }
  13              }
  14
  15           1  int main(int argc, char* argv[])
  16              {
  17           1      foo(0);
  18
  19           1      return 0;
  20              }
  21
```

What makes a good harness?

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

crashes: 1,000 (1,000 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

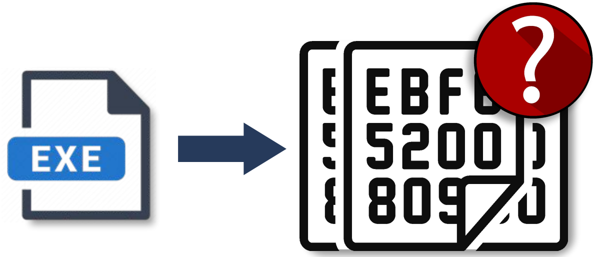
Open Source:

- 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

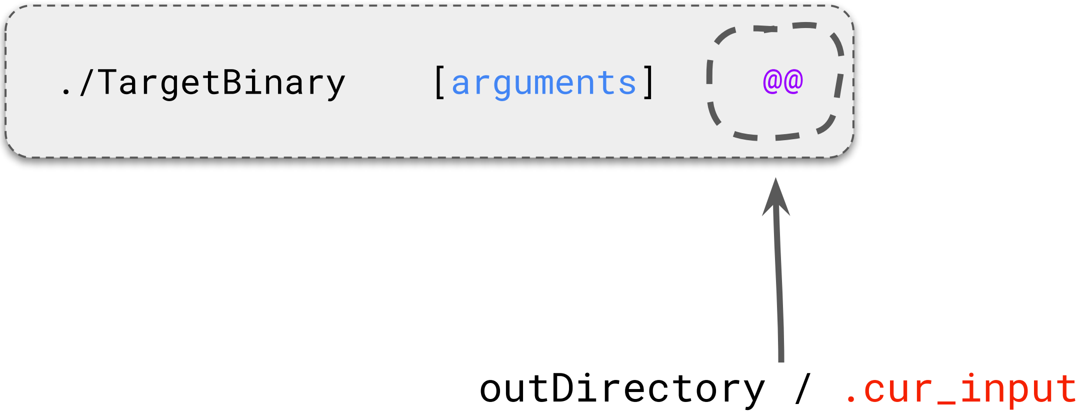
What fuzzers expect…

- **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**

```
int LLVMFuzzerTestOneInput (const char *Data, long Size) {


}
```

What fuzzers expect...

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

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

So what’s the difference?

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

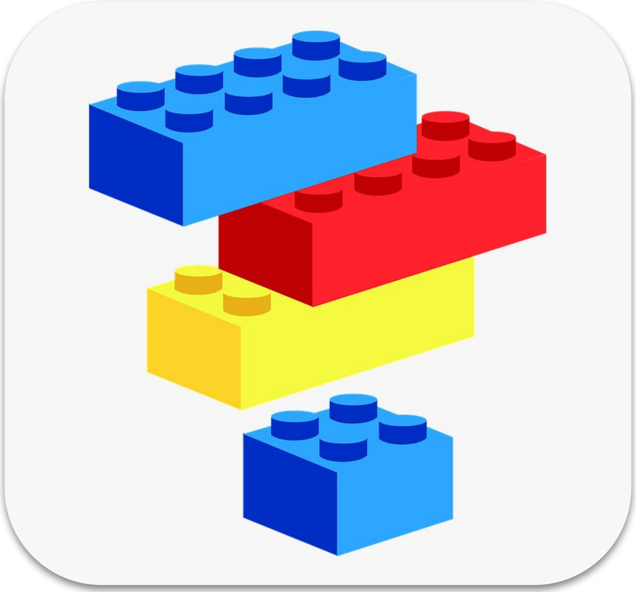
```
./TargetBinary    [arguments]     @@
```

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

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

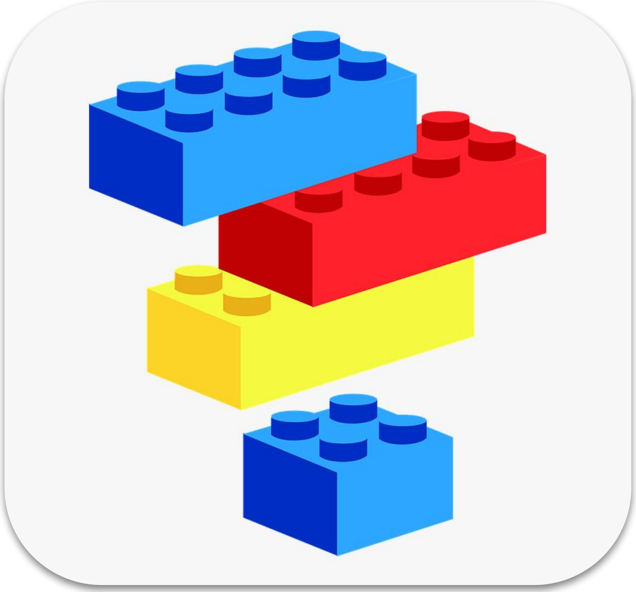
APIs vs Applications

- **API**
  - ???
- **Application**
  - ???



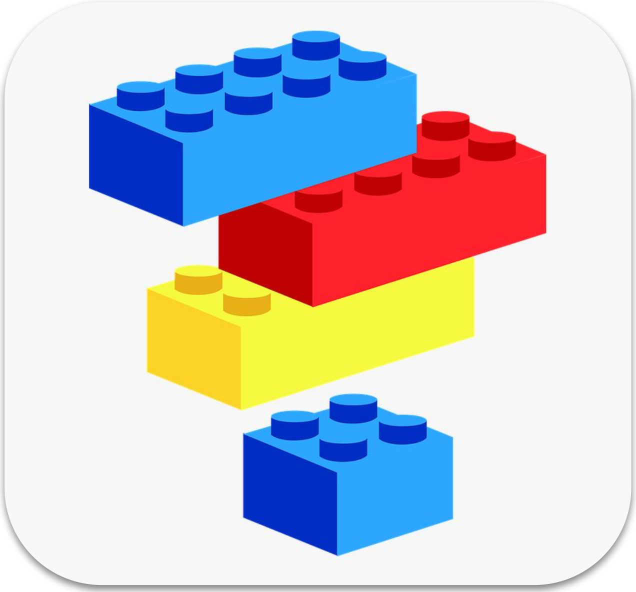
APIs vs Applications

- **API**
  - Application Programming Interface
  - Suite of functions for some niche purpose
  - Intended to be used by other applications
- **Application**
  - ???



APIs vs Applications

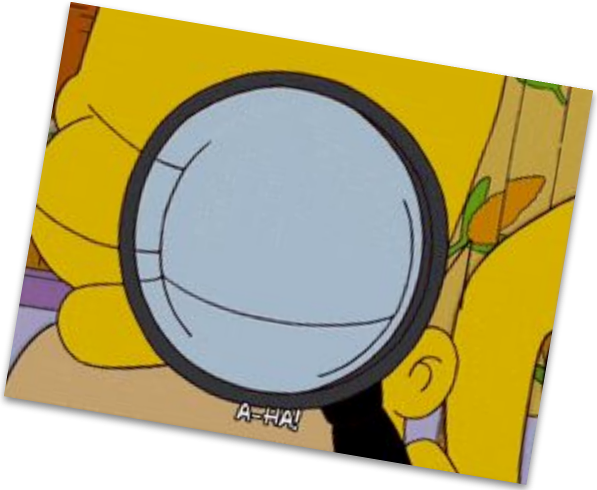
- **API**
  - Application Programming Interface
  - Suite of functions for some niche purpose
  - Intended to be used by other applications
- **Application**
  - Self-contained programs
  - Single- or multi-purpose
  - “Consumes” external APIs



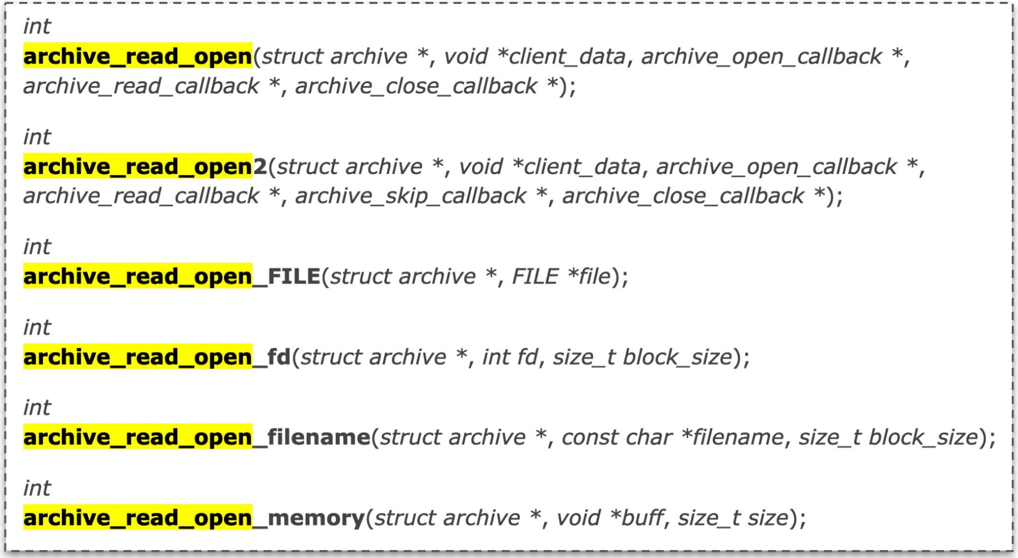
Identifying Suitable Targets

- **For APIs** (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

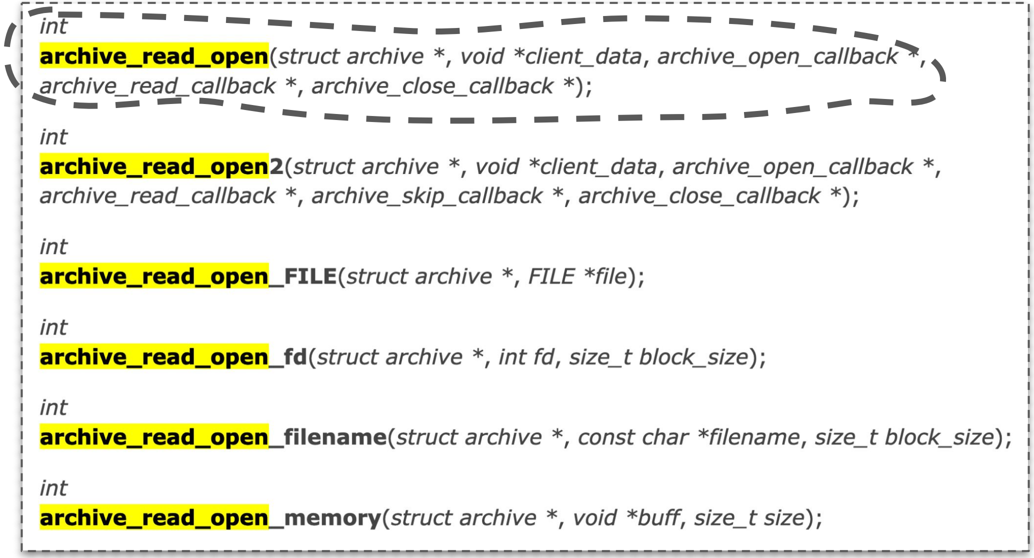


Example: libArchive



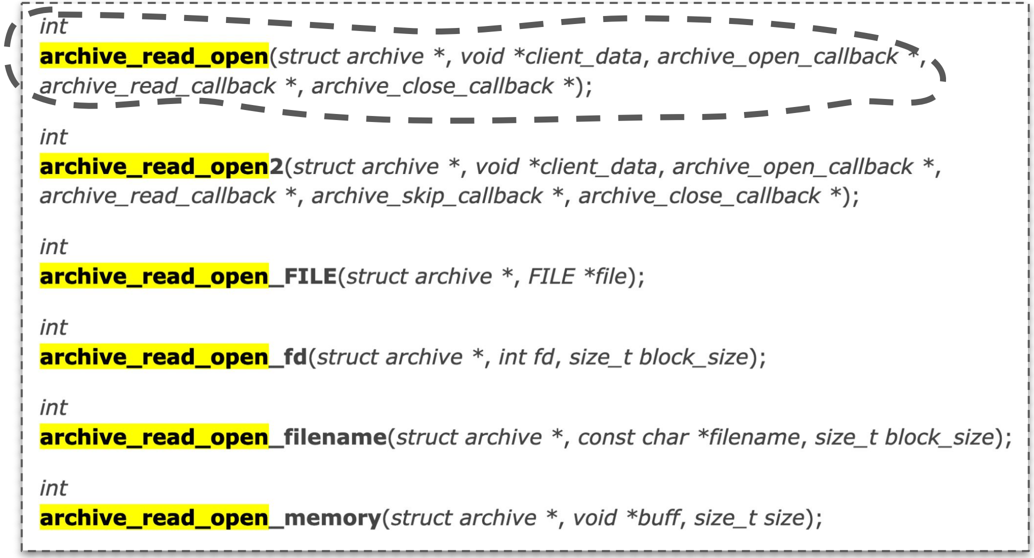
```
int
archive_read_open(struct archive *, void *client_data, archive_open_callback *,
archive_read_callback *, archive_close_callback *);

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

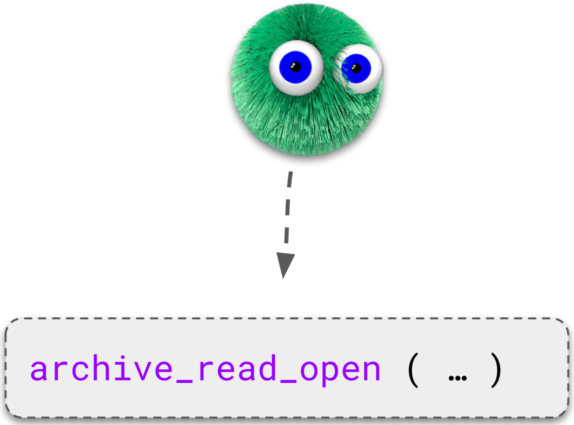


```
int
archive_read_open(struct archive *, void *client_data, archive_open_callback *,
archive_read_callback *, archive_close_callback *);

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

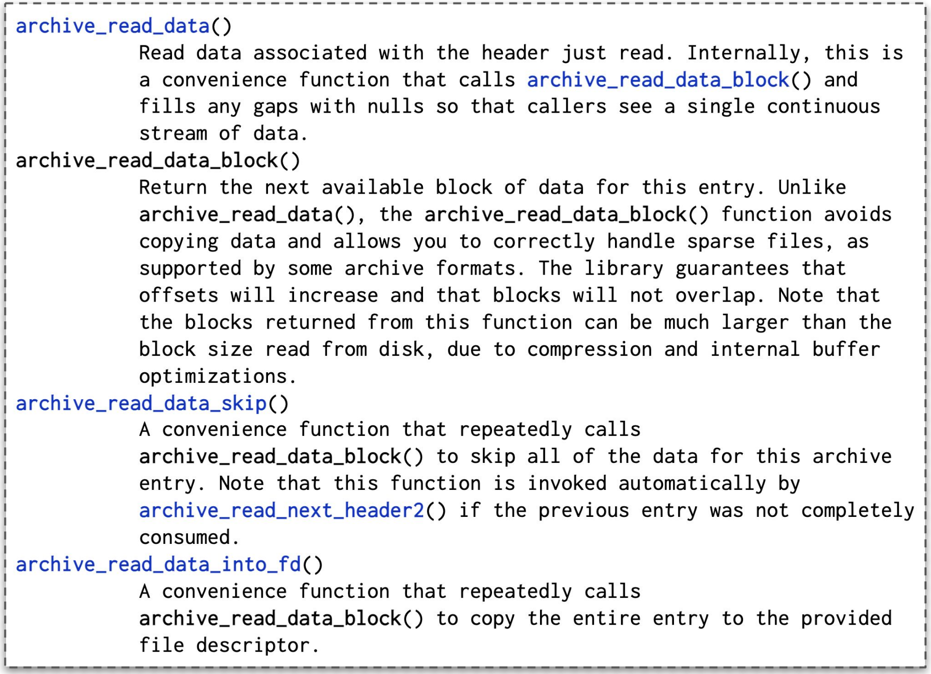


```
int
archive_read_open(struct archive *, void *client_data, archive_open_callback *,
archive_read_callback *, archive_close_callback *);

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);
```

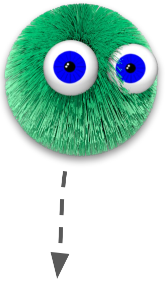


```
archive_read_open ( … )
```

Example: libArchive

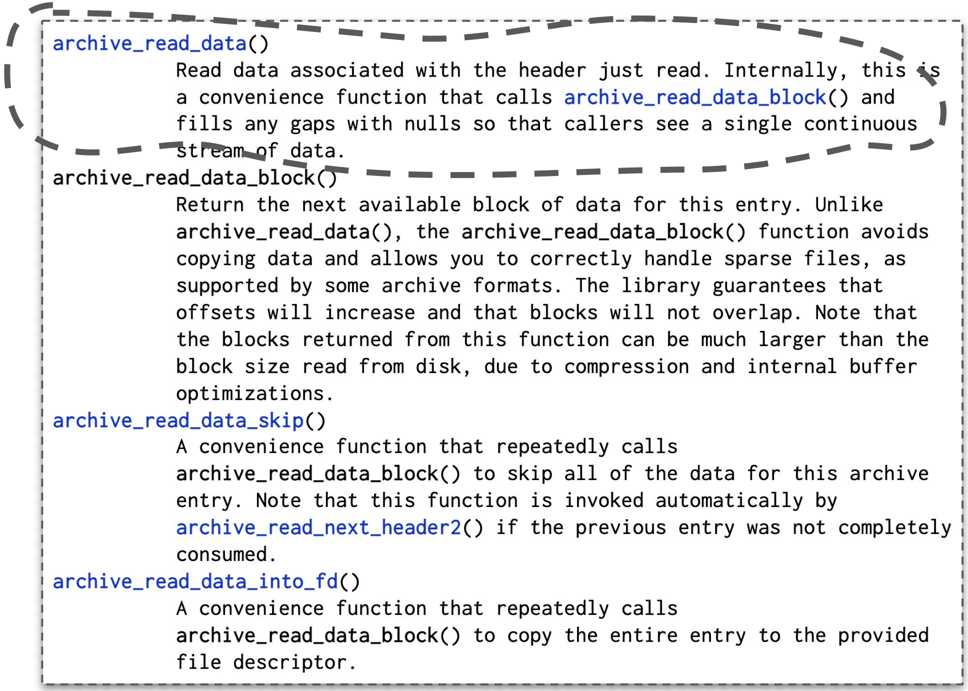


```
archive_read_data()
          Read data associated with the header just read. Internally, this is
          a convenience function that calls archive_read_data_block() and
          fills any gaps with nulls so that callers see a single continuous
          stream of data.
archive_read_data_block()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



```
archive_read_open ( … )
```

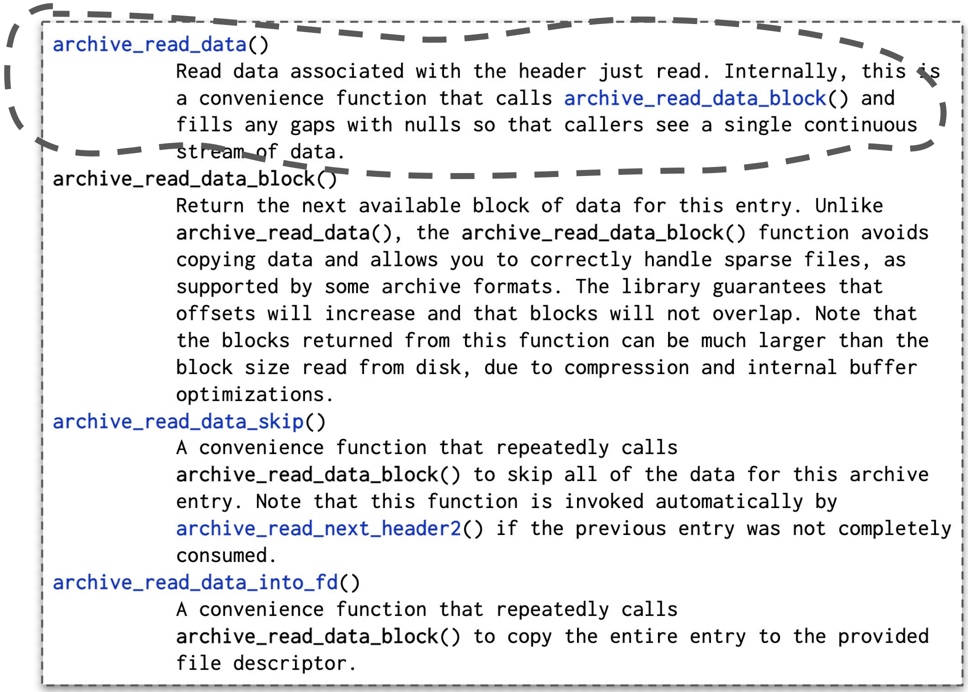
Example: libArchive



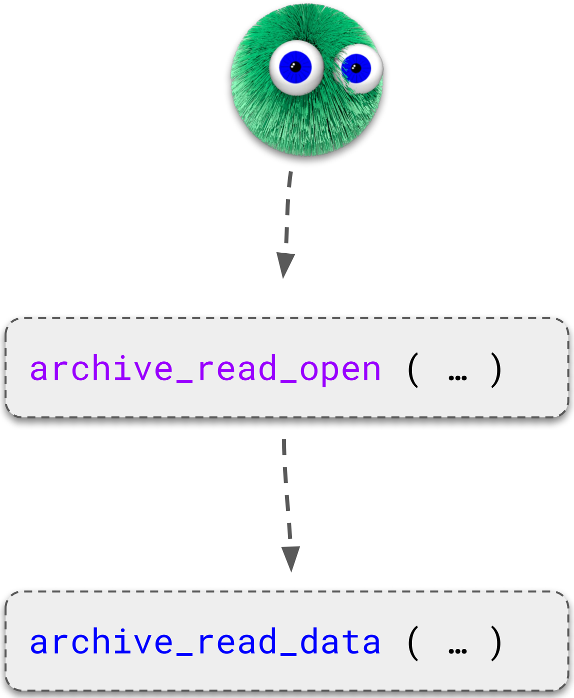
```
archive_read_data()
          Read data associated with the header just read. Internally, this is
          a convenience function that calls archive_read_data_block() and
          fills any gaps with nulls so that callers see a single continuous
          stream of data.
archive_read_data_block()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

`archive_read_open ( … )`

Example: libArchive



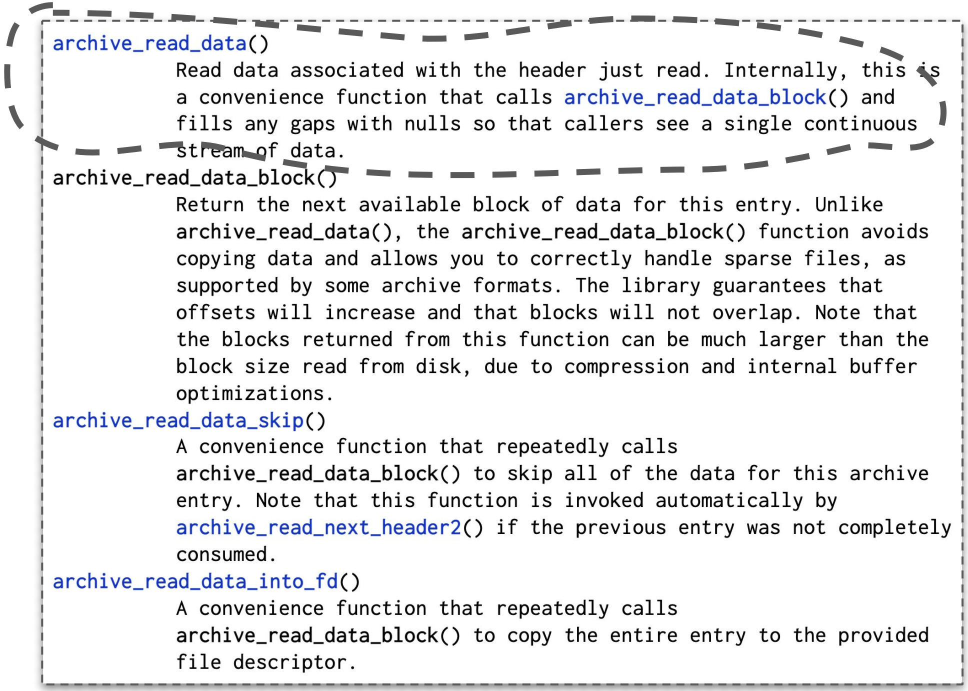
```
archive_read_data()
          Read data associated with the header just read. Internally, this is
          a convenience function that calls archive_read_data_block() and
          fills any gaps with nulls so that callers see a single continuous
          stream of data.
archive_read_data_block()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



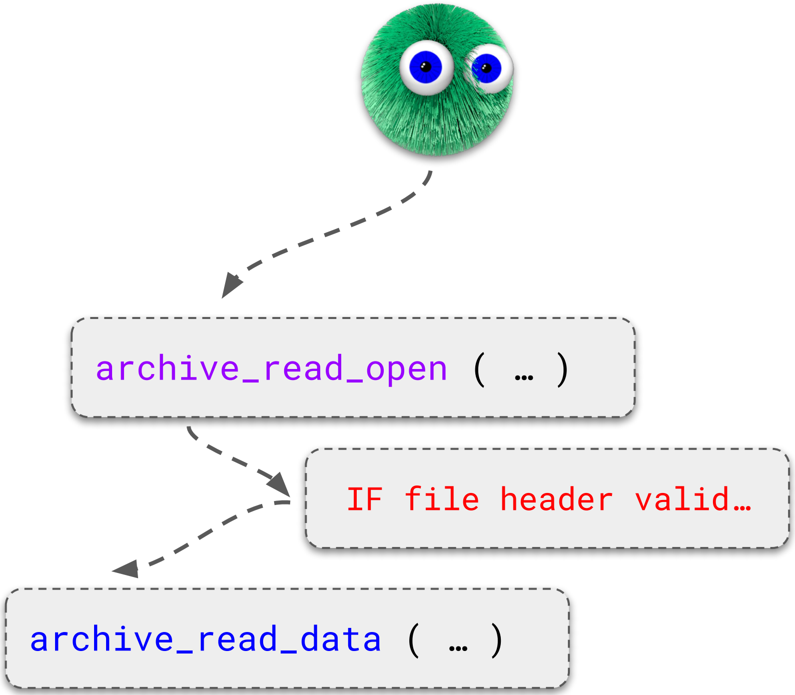
archive_read_open (…)

archive_read_data (…)

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

```
int LLVMFuzzerTestOneInput(*buf, 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

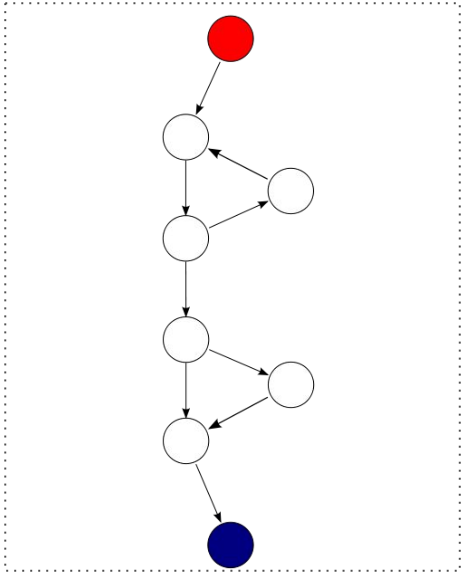
Pin

DynamoRIO

QEMU

Useful Heuristics

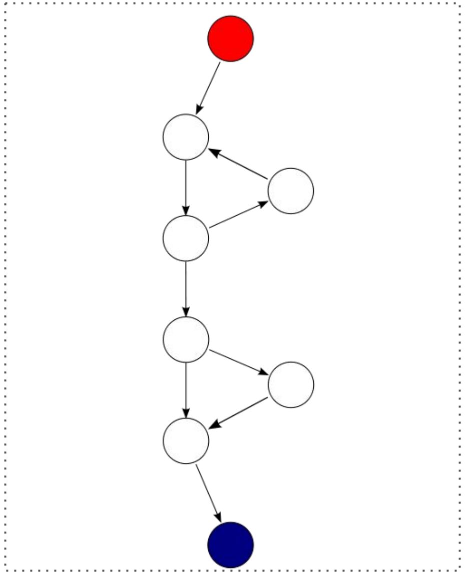
- **Have source code?**
  - … or a reasonably-precise decompilation?
  - Consider McCabe's **Cyclomatic Complexity**
    - `#Edges - #Nodes + 2(#ConnectedComponents)`



Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity

Useful Heuristics

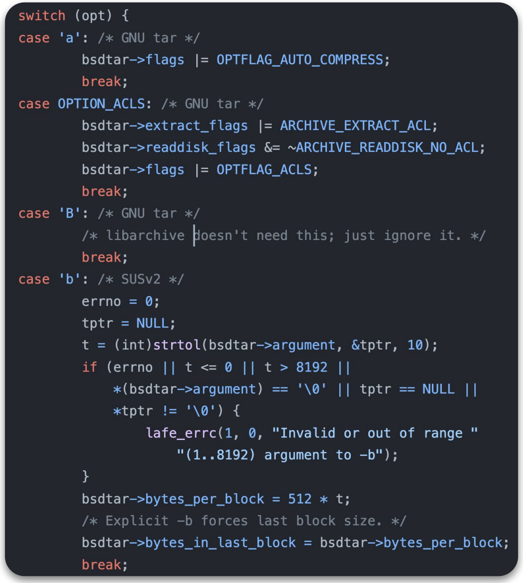
- **Have source code?**
  - ... or a reasonably-precise decompilation?
  - Consider McCabe's **Cyclomatic Complexity**
    - `#Edges - #Nodes + 2(#ConnectedComponents)`
    - Example = `9 - 8 + 2(1) = 3`
  - **Higher score** considered **more interesting target**



Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity

Useful Heuristics

- **Have source code?**
  - … or a reasonably-precise decompilation?
  - Consider McCabe's **Cyclomatic Complexity**
    - #Edges - #Nodes + 2(#ConnectedComponents)
    - Example = 9 - 8 + 2(1) = 3
  - **Higher score** considered **more interesting target**
    - **Caveat:** switch table for parsing cmdline opts
    - High CC yet irrelevant to the fuzzer—**why?**

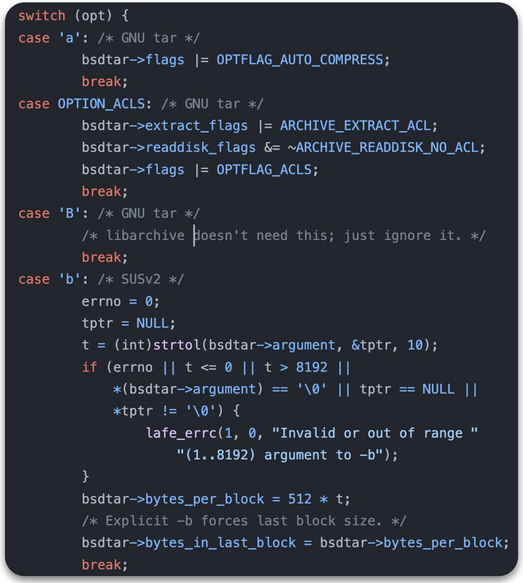


```c
switch (opt) {
case 'a': /* GNU tar */
        bsdtar->flags |= OPTFLAG_AUTO_COMPRESS;
        break;
case OPTION_ACLS: /* GNU tar */
        bsdtar->extract_flags |= ARCHIVE_EXTRACT_ACL;
        bsdtar->readdisk_flags &= ~ARCHIVE_READDISK_NO_ACL;
        bsdtar->flags |= OPTFLAG_ACLS;
        break;
case 'B': /* GNU tar */
        /* libarchive doesn't need this; just ignore it. */
        break;
case 'b': /* SUSv2 */
        errno = 0;
        tptr = NULL;
        t = (int)strtol(bsdtar->argument, &tptr, 10);
        if (errno || t <= 0 || t > 8192 ||
            *(bsdtar->argument) == '\0' || tptr == NULL ||
            *tptr != '\0') {
                lafe_errc(1, 0, "Invalid or out of range "
                    "(1..8192) argument to -b");
        }
        bsdtar->bytes_per_block = 512 * t;
        /* Explicit -b forces last block size. */
        bsdtar->bytes_in_last_block = bsdtar->bytes_per_block;
        break;
```

Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity

Useful Heuristics

- **Have source code?**
  - ... or a reasonably-precise decompilation?
  - Consider McCabe’s **Cyclomatic Complexity**
    - `#Edges - #Nodes + 2(#ConnectedComponents)`
    - Example = `9 - 8 + 2(1) = 3`
  - **Higher score** considered **more interesting target**
    - **Caveat:** switch table for parsing cmdline opts
    - High CC yet irrelevant to the fuzzer—**why?**
      - Path is hardcoded pre-fuzzing!



```c
switch (opt) {
case 'a': /* GNU tar */
        bsdtar->flags |= OPTFLAG_AUTO_COMPRESS;
        break;
case OPTION_ACLS: /* GNU tar */
        bsdtar->extract_flags |= ARCHIVE_EXTRACT_ACL;
        bsdtar->readdisk_flags &= ~ARCHIVE_READDISK_NO_ACL;
        bsdtar->flags |= OPTFLAG_ACLS;
        break;
case 'B': /* GNU tar */
        /* libarchive doesn't need this; just ignore it. */
        break;
case 'b': /* SUSv2 */
        errno = 0;
        tptr = NULL;
        t = (int)strtol(bsdtar->argument, &tptr, 10);
        if (errno || t <= 0 || t > 8192 ||
            *(bsdtar->argument) == '\0' || tptr == NULL ||
            *tptr != '\0') {
                lafe_errc(1, 0, "Invalid or out of range "
                    "(1..8192) argument to -b");
        }
        bsdtar->bytes_per_block = 512 * t;
        /* Explicit -b forces last block size. */
        bsdtar->bytes_in_last_block = bsdtar->bytes_per_block;
        break;
```

Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity

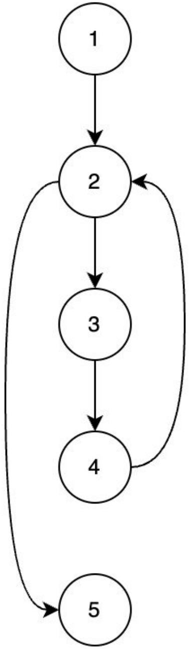
Useful Heuristics

```
int i = 1;
while (i <= 5)
{
    cout<<i<<endl;
    i++;
}
```

CC score

= 5 - 5 + 2(1)

= 2

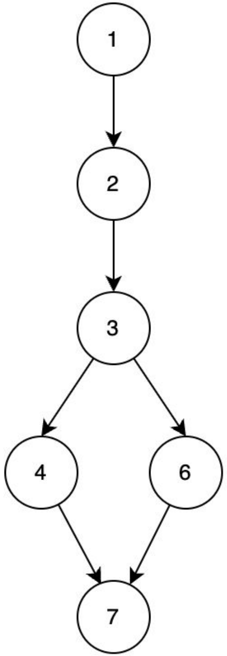


```
int a = 1;
int b = 2;
if (a>b)
  cout<<"a is greater";
else
  cout<<"b is greater";
```

CC score

= 6 - 7 + 2(1)

= 1



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?

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?

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?