Detecting Leaks of Sensitive Data Due to Stale Reads

IEEE SecDev 2018

Will Snavely*,
Will Klieber* (presenting),
Ryan Steele*,
David Svoboda*,
Andrew Kotov*

*Software Engineering Institute (SEI)
Carnegie Mellon University
Pittsburgh, PA, USA

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-1108

Motivation

Invalid memory reads can leak sensitive information.

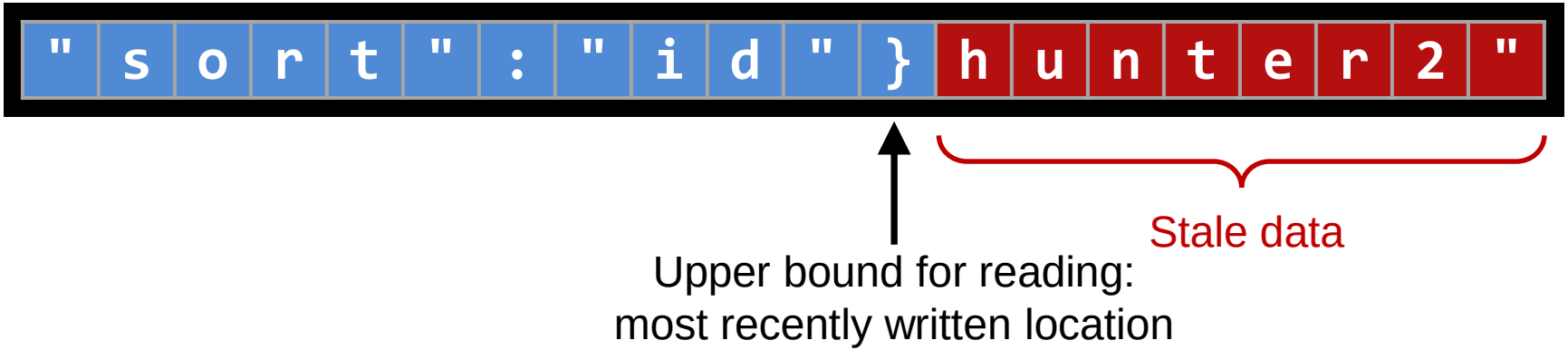
- Heartbleed (2014, OpenSSL).
- Cloudbleed (2017, HTML parser).
- Unaffected by mitigations such as ASLR and DEP.
- Even reads within the allocated bounds can lead to leaks in the case of a re-usable buffer with stale data.
  This affects even memory-safe languages,
  e.g., Jetty leaked passwords (CVE-2015-2080 “JetLeak”).

} Focus of this talk

Leakage of Sensitive Information in Re-Used Buffer (based on a real vulnerability in the Jetty web server)

Buffer contents after first HTTP request:

Buffer contents after second HTTP request (from a different client):



Overview

Problem addressed: Leaks of sensitive stale data from a re-used buffer.

Approach: Heuristic-driven dynamic analysis for detecting reads that may be accessing stale sensitive data.

Results:

- Our dynamic analyses for C and Java can detect & stop Heartbleed (OpenSSL) and JetLeak (Jetty).
- Evidence for attaining reasonably low false-positive rate (currently 0.2 alarms / kLOC for GNU Coreutils on its testsuite).

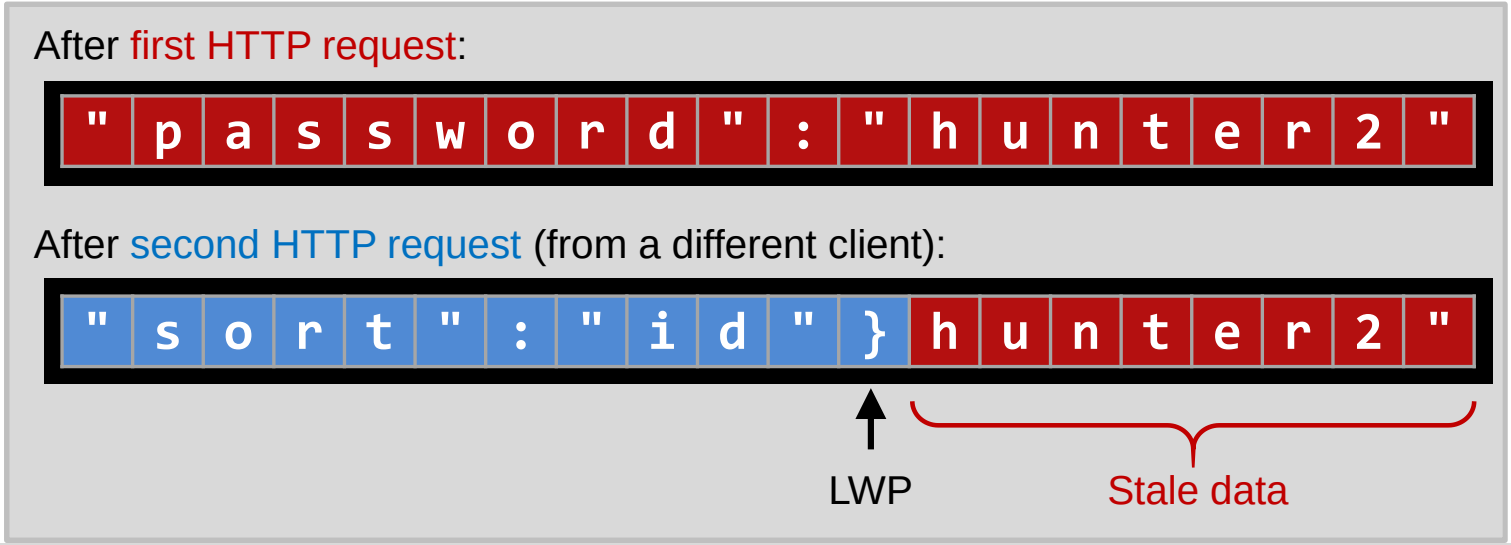
Staleness (unlike out-of-bounds access) is not a mechanically defined property; it refers on developer intent.

Sequential-write heuristic

Definition (*Qualifying*): A buffer *B* is a ***qualifying*** buffer iff every write is either to index 0 or to the successor of the last written position (LWP).

- Note that in addition to writing sequentially, the definition of qualifying allows restarting at the beginning (index 0).

The **sequential-write heuristic** posits that the valid (non-stale) portion of a qualifying array is from index 0 to the LWP.



Runtime monitoring agent

- We've designed a runtime monitoring agent to detect stale reads, as determined by a staleness heuristic.
- We focus on C and Java.
- Intercept primitive memory operations at runtime:
  - allocations of buffers in memory
  - writes to allocated buffers
  - reads of allocated buffers
  - deallocations of buffers
- We use the term "buffer" to refers to any array-like data structure, including simple C arrays as well as the Java `ByteBuffer` class.

Use in developing secure software

Our technique can be used as an aid in testing and fuzzing a program.

It can be used as a dynamic enforcement mechanism in production.

What about false positives (i.e., qualifying buffers with valid data beyond LWP)?

- First run the dynamic analysis on a representative set of ‘good’ traces, recording the allocation site of buffers with flagged reads.
- The set of ‘good’ traces can come from a test suite, from logs of running it in production, or otherwise exercising the program in a controlled manner.
- Adjust instrumentation to exclude allocation sites flagged on the ‘good’ traces.

Data structures used in our analysis

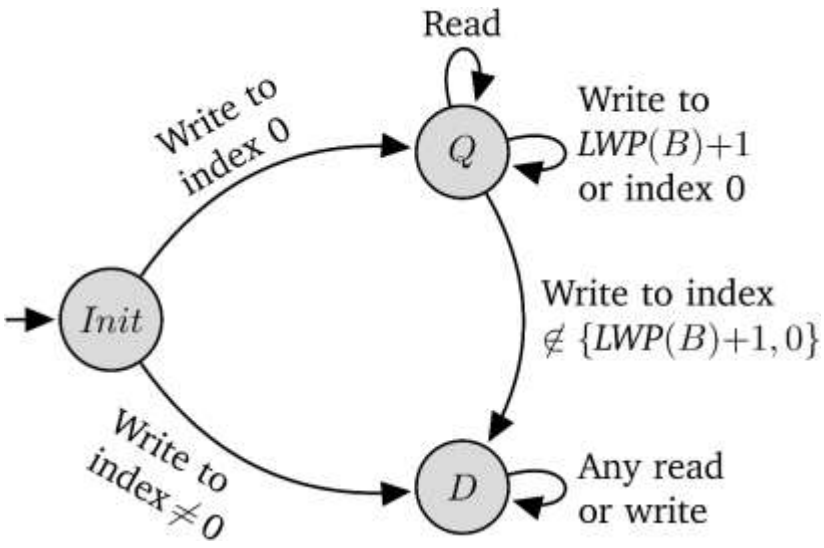
We maintain metadata for every currently allocated buffer *B:*

- *State(B)* ∈ {`Init`, `Qualified`, `Disqualified`}.
- *LWP(B)* is the last written position of *B*.

The sequential-write heuristic posits that, if *State(B)* is `Qualified`, then:

- Every offset i where $0 \leq i \leq \text{LWP}(B)$ holds valid (non-stale) data.
- Every offset i where $\text{LWP}(B) < i < \text{len}(B)$ holds stale or uninitialized data.

Algorithm



Algorithm 1 Dynamic Analysis State Machine

```
1:  procedure ALLOC(B)
2:      State(B) ← Init
3:      LWP(B) ← −1
4:  end procedure

5:  procedure WRITE(B, off, size)
6:      if off = LWP(B) + 1 or off = 0 then
7:          LWP(B) ← off + size − 1
8:          if State(B) = Init then
9:              State(B) ← Qualified
10:         end if
11:     else
12:         State(B) ← Disqualified
13:     end if
14: end procedure

15: procedure READ(B, off, size)
16:     if State(B) = Qualified and off > LWP(B) then
17:         ISSUESTALEREADWARNING(B, off, size)
18:     end if
19: end procedure
```

Java implementation

The Java implementation of this dynamic analysis has two components:

- a Java agent plugin, which modifies Java class files to insert callbacks, and
- a runtime, which provides implementations of the inserted callbacks.

We focused on java.nio.ByteBuffer. However, the technique applies to any array-like classes as well as native Java arrays.

The ByteBuffer class implements an array-like data structure:

- Data is inserted with **`put`** methods and retrieved with **`get`** methods.

We instrumented the **`put`** methods to update the LWP and qualification state.
We instrumented the **`get`** methods to flag beyond-LWP reads if qualifying.

Our metadata for each buffer (i.e., its state and its last written position) is maintained in a dictionary data structure within the `Runtime` class, keyed off a unique identifier associated with each `ByteBuffer` instance.

Java bytecode rewriting

```
class ByteBuffer /*...*/ {
  public byte get(int index) {
    Runtime.captureRead(this, index, 1);
    // Implementation of get() starts here...
  }

  public void put(int index, byte b) {
    Runtime.captureWrite(this, index, 1);
    // Implementation of put() starts here...
  }
}
```

Listing 1. Inserting callbacks with bytecode rewriting

Running the Java analysis

- To run the analysis on a Java program, the JVM is invoked with the "`-javaagent`" argument, and the agent JAR file passed as an argument.
- The agent plugin runs before the application is launched. The first thing it does is add our runtime JAR to the bootstrap class loader.
- Next, the desired bytecode modifications are performed.
- Finally, the Java application is launched.
- No source code modifications to the original Java application are required.

C implementation

The C implementation of the dynamic analysis has two principal components:

- a set of compiler instrumentations, added with a slightly modified AddressSanitizer pass in the Clang compiler; and
- a runtime agent that intercepts allocations, reads, and write.

The runtime is built on top of Intel Pin, a dynamic binary instrumentation tool.

We are concerned with 3 types of allocation: stack, heap (“dynamic”), global.

Compiler instrumentation

Here we see the instrumentation code (highlighted in gray) added during compilation.

Heap allocations do not receive any special instrumentation during compilation; neither do reads or writes.

Instead, we use Intel Pin to dynamically instrument the binary executable to handle those cases.

```
static char global[1024];

void premain() {
  capture_global_alloc(global, 1024);
}

void func(size_t size) {
  char stack[1024];
  capture_stack_alloc(stack, 1024);

  // Heap allocations don't receive
  // additional instrumentation
  char *heap = (char*)malloc(size);
  free(heap);

  capture_stack_free(stack);
}

void postmain() {
  capture_global_free(global);
}
```

C implementation (2)

Why use dynamic instrumentation instead of doing it all during compilation?

- The deciding factor was that AddressSanitizer can miss some reads/writes, for example if they occur in a non-instrumented third-party library, or if they occur within inline assembly.
- To avoid this possibility, we decided to use Pin to capture reads and writes. We wanted to get as close to the ground truth as possible, and Pin lets us to that better.

When an allocation is detected, a record holding the state of the allocation is inserted into a balanced binary search tree.

- The records in this tree are sorted by the starting address of the allocation.
- When the program performs a read or write, our instrumentation finds the allocation record in the tree with the starting address closest to the read/write address, without exceeding it. This is a O(log *n*) operation, where *n* is the number of allocations currently tracked.

Adjustments to C analysis

Some implementations of the `memcpy` function write data starting from the end of the destination array and ending at the start.

- As a workaround, we treat `memcpy` as a special case: we treat it as a single monolithic read followed by a single monolithic write.

Some buffers in OpenSSL are sequentially written except for a small header.

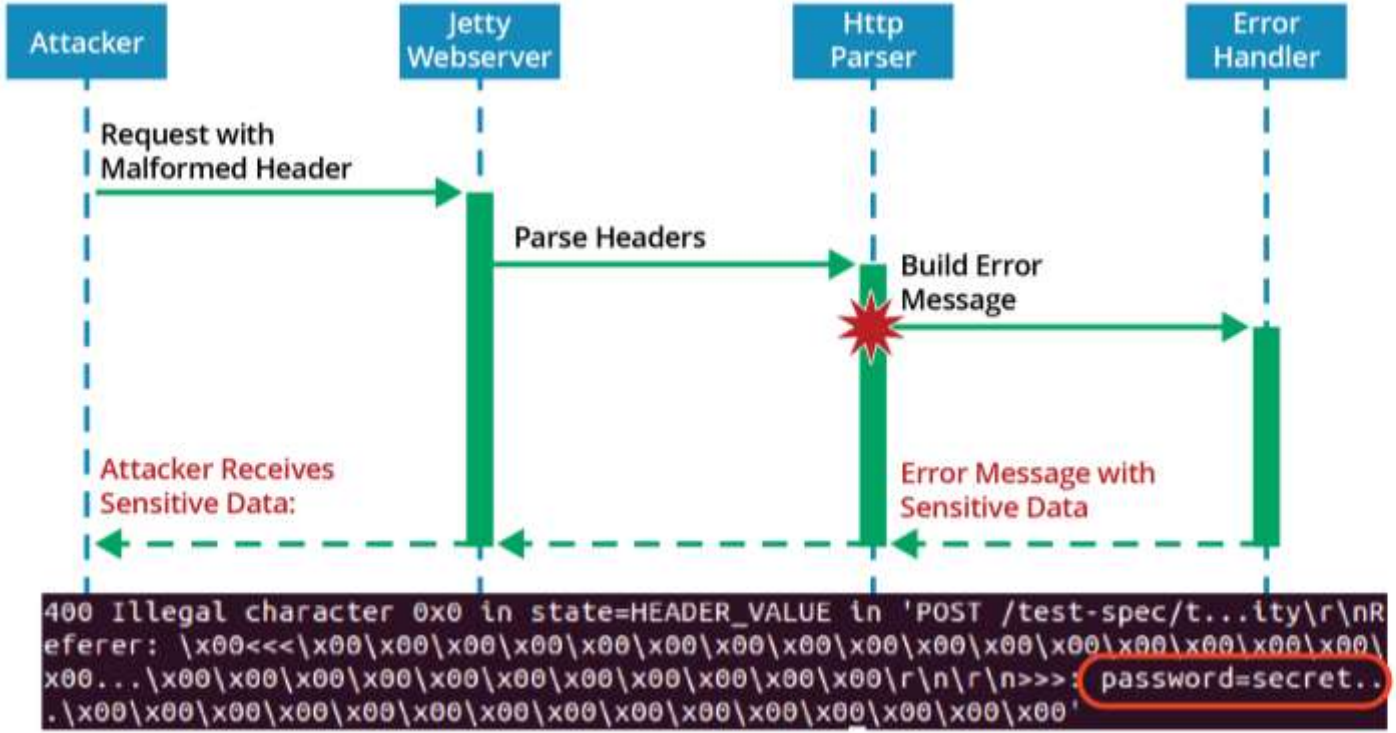
- We relax the constraint that new generations of data must begin at offset 0, and instead allow new generations to begin in the range [0, ε], where ε is some small user-defined constant (we used ε=4).

Definition (*ε-qualifying*): A buffer *B* is an *ε-qualifying* buffer iff every write is either to an index *i* where $0 \leq i \leq \varepsilon$ or to the successor of the last written position (LWP).

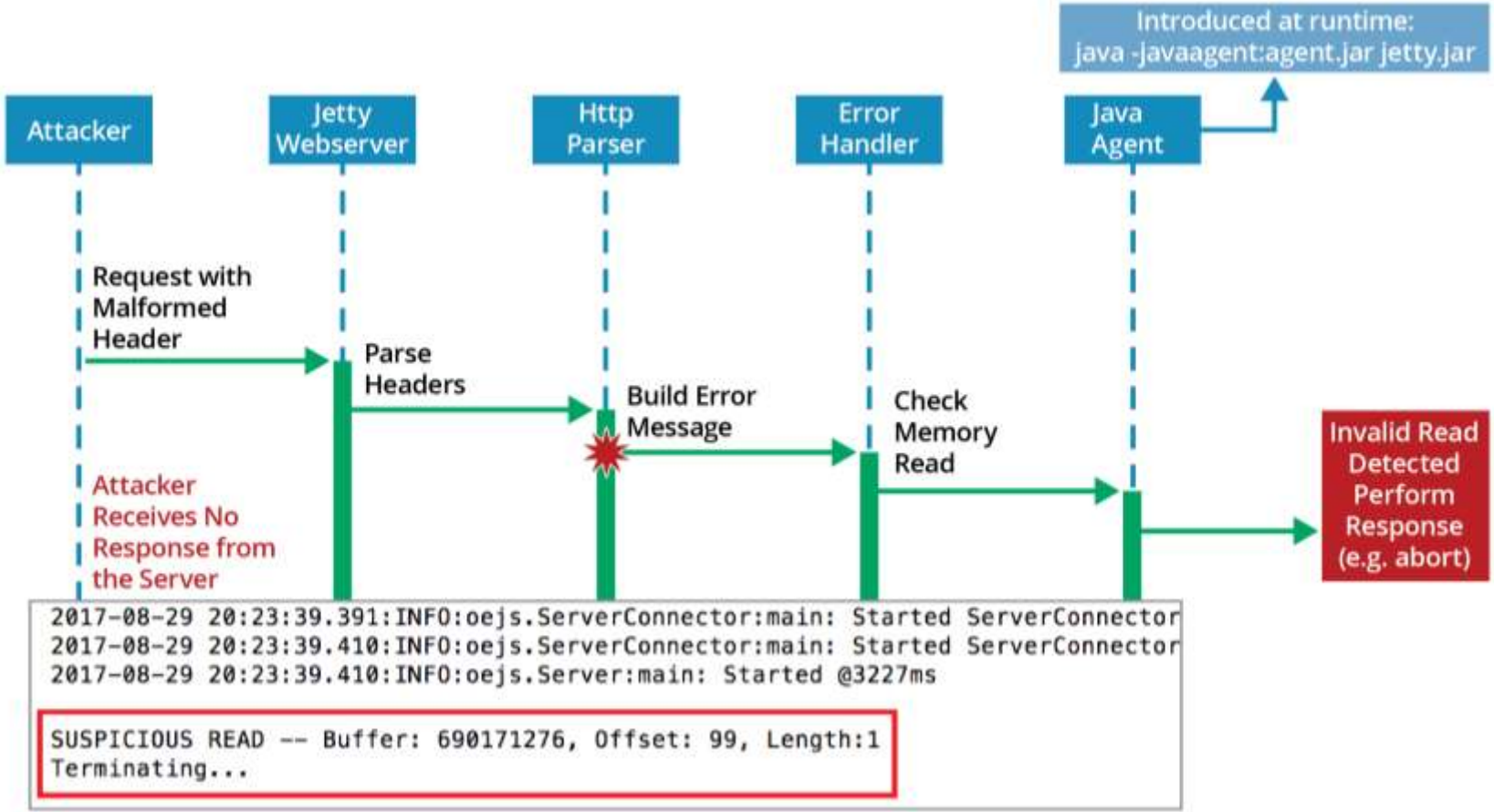
Results

Unpatched Jetty

Using JetLeak testing script provided by Gotham Digital Science:



Jetty with Runtime Bounds Enforcement



Heartbleed

We constructed a standalone test application using version 1.0.1a of the OpenSSL library, which is susceptible to Heartbleed. This application allows the user to send customized heartbeat messages to a local OpenSSL server.

The client can control the length of the heartbeat payload *(L_actual)*, as well as the value of payload length field in the request (*L_user*). The Heartbleed bug is triggered by setting *L_user* to a value larger than *L_actual*.

We first confirmed that Heartbleed exhibits reads of stale data within the bounds of an array, in addition to out-of-bounds reads.

We fixed the value of *L_actual* to some small integer, and fed increasingly large values of *L_user*. We did not receive an out-of-bounds access error from AddressSanitizer until *L_user* exceeded 17725. This bound was observed when we varied the value of *L_actual* as well.

These observations suggest that when *L_actual* < *L_user* <= 17725, the server will leak stale or uninitialized bytes from the in-bound portion of the shared buffer.

Heartbleed log

```
Send heartbeat request 1 (valid):
    Payload length = 500, Length field = 500
Allocation of size 17736 at address 0x629000005200
Stored 5 bytes at offset 3
Stored 519 bytes at offset 8
Loaded 500 bytes at offset 11

Send heartbeat request 2 (tampered):
    Payload length = 100, Length field = 200
Stored 5 bytes at offset 3
Stored 119 bytes at offset 8
Loaded 200 bytes at offset 11:  Suspicious Read
```

Listing 5. Heartbleed detector log summary

GNU Core Utilities

We ran the tool on the GNU Coreutils using the test suite included in the coreutils repository. (This codebase consists of 80,000 lines of source code, plus another 486,000 lines from included libraries.)

Running the test suite produced a total of 34,000 traces (executions).

In the table, row 1 shows the number of allocation sites that are disqualified (D), qualified (Q), and flagged (F) on at least one trace.

Results on GNU Coreutils

Configuration	D	Q	F
1. Base configuation	1852	1409	278
2. LoadOverlap	1849	1412	59
3. LoadOverlap, MergeDQ	1849	1310	38
4. LoadOverlap, MergeDQ, MinSize=32	1515	291	26
5. LoadOverlap, MergeDQ, MinSize=64	1178	186	17

Fig. 3. Number of allocation sites (out of 3160) that are disqualified (D), qualified (Q), and flagged (F) on at least one trace

Conclusion and future work (1/2)

Our current implementation uses a combination of compile-time instrumentation and dynamic binary instrumentation. This has the advantage of catching all memory accesses, even those in code that were not instrumented at compile-time.

However, it has disadvantages: (1) it is unsuitable for deployment in production use, and (2) it does not play nice with white-box fuzzing tools. In the future we plan to perform all instrumentation as part of compilation. Then we would be able to use white-box fuzzing tools such as AFL [15] to attempt to find inputs that trigger flagged reads. There would still be some manual analysis necessary to determine whether the flagged read is a true positive or a false positive.

Conclusion and future work (2/2): fuzzing and beyond

The manual effort can be minimized by employing a slight variation on existing fuzzing techniques. Given the original codebase C, we create a variation C0 that zeroes out any data that the staleness heuristic posits to be stale. We then use fuzzing techniques to try to find an input vector that not only triggers a flagged read but also causes C0 to produce different output than C. In that way, manual code inspection can be avoided if the judgment of true/false positive can be made simply on the basis of which of the two outputs (those of C and C0) is correct for the input vector.

Our ultimate goal with this work is to develop techniques for repairing code. There are two directions we are considering to get closer to this goal: developing more efficient runtime detection mechanisms, or developing techniques for directly repairing the source code.