Automated Code Repair to Ensure Memory Safety

Memory-related bugs in C/C++ code are notorious for leading to vulnerabilities. We're developing techniques for automated repair of source code to eliminate such vulnerabilities and enable a proof of memory safety.

What about distinguishing false alarms from true vulnerabilities?

We repair all potential memory-safety vulnerabilities, at a cost of an often small runtime overhead. (Manual tuning might be needed for performance-critical parts.)

Intermediate Representation (IR)

Problem: Static analysis generally works best on a suitable IR, but the repair must be done on the original source code.

Solution: We augment the IR with tags that record how to transform back to source.

- Each abstract syntax tree (AST) node is tagged with a reference to corresponding original (unpreprocessed) source code text.
- We have developed a set of reversible transformations that start with the original AST and transform it to the IR.
- The IR is repaired and then transformed back to source using the tags. If a repair invalidates a tag, then the tag is ignored.
- A macro invocation is preserved if the smallest containing AST node is unchanged; otherwise, it is expanded.
- We consider only a single build config but preserve **`#ifdef`** s where possible.

Our ACR tool takes buggy code (shown here, a format-string vulnerability) and repairs the code to remove the vulnerability while preserving the desired functionality of the code.

Definition of Memory Safety

We say that a program is *memory-safe* if and only if, on every possible execution of the program, every memory access (read or write) is to a location in a currently allocated region.

Possible executions include those where the compiler leaves gaps of unallocated memory between variables on the stack.

Memory safety is often divided into 2 parts:

- Spatial: Writing or reading beyond the bounds of a memory region (FY18+ work)
- Temporal: Writing or reading to a region after it has been deallocated (FY19+ work)

(Dereferencing NULL is technically a mem violation, but low severity, so we ignore.)

Heuristic

If a program performs arithmetic on a memory address *p1* to obtain a new memory address *p2*, and *p2* is later dereferenced, then *p2* should be in the same allocated memory region as *p1*.

The ISO C standard actually requires compliance with this heuristic (on pain of undefined behavior) for arithmetic on values of pointer type.

Static Analysis and Repair

For each memory access $*p$, we generate a precondition ensuring it is within bounds:

$$\mathrm{MemLo}(p) \le p < \mathrm{MemHi}(p)$$

where MemLo and MemHi are functions of the provenance (not value) of p:

If the value of p (at a particular timepoint in an execution trace) was computed by pointer arithmetic on the result of a memory allocation (e.g., malloc), then MemLo(p) and MemHi(p) denote the lower bound (incl.) and upper bound (excl.) of this memory region.

For each precond, do one of the following:

- Prove that it is satisfied.
- Add bounds check with existing variables.
- Modify function signatures and/or structs to include bounds info ("fat pointers").
- Modify program to record info about bounds in global lookup table (as in SoftBound).

In the past, fat pointers were disfavored due to inability to analyze or repair libraries available only in binary form. However, new developments in SEI's Pharos platform will allow us to overcome this limitation by tackling binaries.

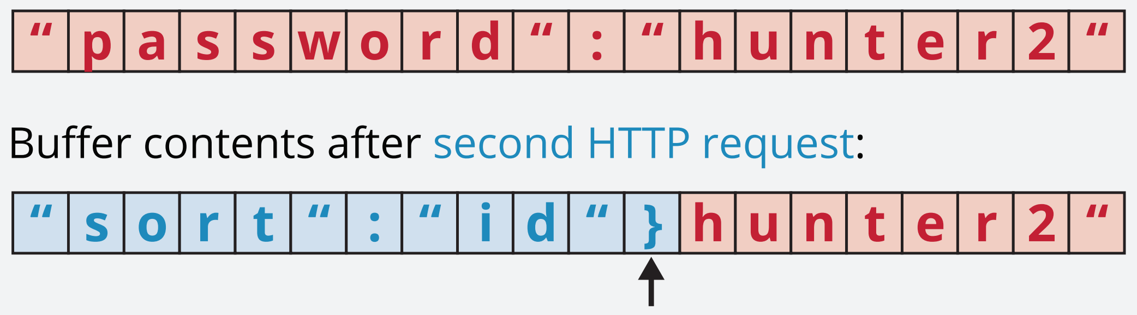
Leaks of Sensitive Data via Stale Reads

Consider a web server that stores a received request in a reusable buffer. Once the server is done with the request, the buffer holds *stale* data, which can later be (partially) overwritten by a new request.

Example: Reused buffer with stale data

Buffer contents after first HTTP request:

Buffer contents after second HTTP request:



“	p	a	s	s	w	o	r	d	“	:	“	h	u	n	t	e	r	2	“
“	s	o	r	t	“	:	“	i	d	“	}	h	u	n	t	e	r	2	“

The upper bound for reading is the last (most recent) written location

We developed a heuristic for identifying such a buffer and what part of it is valid.

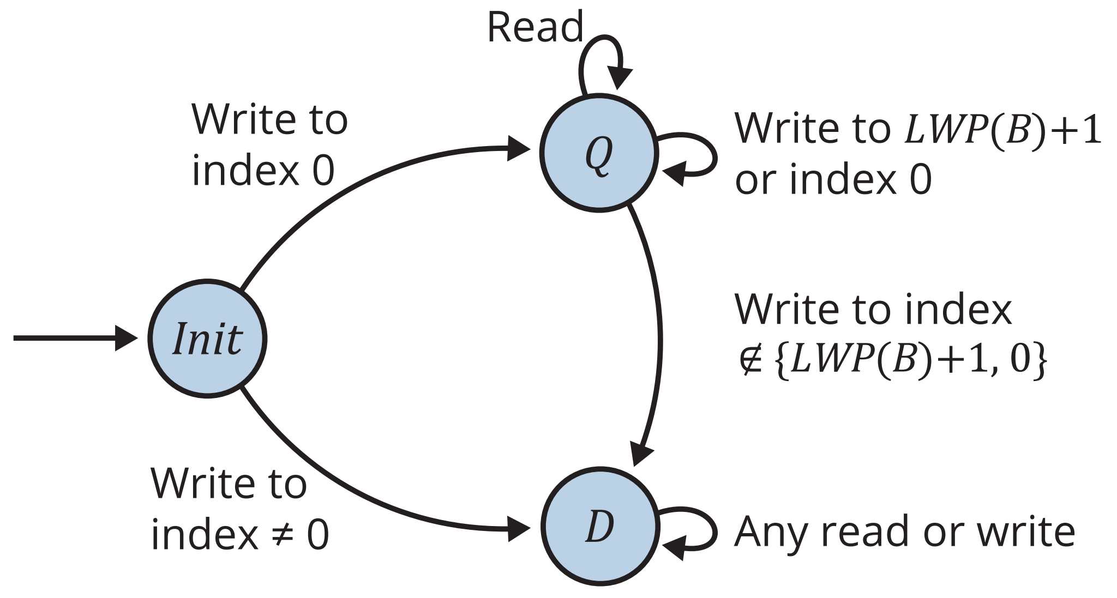
Definition: A buffer B is ***qualifying*** if and only if every write is to either index 0 or the successor of the last written position (LWP).

Our *sequential write* heuristic posits that a qualifying array contains valid (non-stale) data up to and including the LWP.

We implemented a dynamic analysis based on this heuristic, targeting C and Java. Our analysis detects JetLeak (CVE-2015-2080) in Jetty and Heartbleed in OpenSSL.

In analyzing GNU coreutils (80k LOC, plus 486k LOC of library), there were 17 alarms (all suspected/confirmed false positives).

Developing a static analysis is future work.



Carnegie Mellon University
Software Engineering Institute

Dr. Will Klieber | weklieber@sei.cmu.edu

Copyright 2018 Carnegie Mellon University. All Rights Reserved.

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM18-1130