Research Review 2017

Inference of Memory Bounds

Will Klieber, software security researcher

Joint work with Will Snavely

Inference of Memory Bounds

(One-year project, extended to December 2017)

Goal: Detect the intended bounds of memory.

Problem 1: Repair buffer security vuls. Both out-of-bounds WRITEs and READs.

Leakage of sensitive info (out-of-bounds reads):

- HeartBleed vulnerability.
- Unaffected by mitigations such as ASLR and DEP.
- Re-usable buffer with stale data: bounded to valid portion of buffer.
- Affects even memory-safe languages: e.g., Jetty leaked passwords (CVE-2015-2080).

Leakage of Sensitive Info in Re-Used Buffer

Buffer contents after first HTTP request:

"	p	a	s	s	w	o	r	d	"	:	"	h	u	n	t	e	r	2	"

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

"	s	o	r	t	"	:	"	i	d	"	}	h	u	n	t	e	r	2	"

Upper bound for reading:
most recently written location

Inference of Memory Bounds

(One-year project, extended to December 2017)

Goal: Detect the intended bounds of memory.

Problem 1: Buffer Security vuls. Both out-of-bounds WRITEs and READs.

Leakage of sensitive info (out-of-bounds reads):

- HeartBleed vulnerability.
- Unaffected by mitigations such as ASLR and DEP.
- Re-usable buffer with stale data: bounded to valid portion of buffer.
- Affects even memory-safe languages: e.g., Jetty leaked passwords (CVE-2015-2080).

Problem 2: Decompilation of binaries. We will reconstruct information of the form “the bounds of pointer *p* is the interval [*n*, *m*]”.

Solution & Approach: Static analysis to find and evaluate likely bounds.

For decompilation: Report these bounds, use when naming variables.

For repair: Test with dynamic analysis. Repair code to check bounds.

Overall Approach for Candidate Bounds Checks

Static Analysis

Identify qualifying arrays

Static Analysis

Identify candidate bounds

Dynamic Analysis

Confirm or reject candidate bounds

Program Transformation

Repair program to abort if bounds check fails

For leakage of sensitive information

Static Analysis – Strategies to Propose Candidate Bounds

1. (For reads) The most recently written position in the buffer.
2. Bounds of region allocated by `malloc`.
3. Pointer arithmetic with constant offset (e.g., field of a struct) – for decompilation.
4. Analysis of memory accesses within loops and limits of the loop.
   - Exact if the number of iterations is known at start of loop.
   - Only a candidate bound if it is possible to break out of the loop early.
5. Invariants for `structs` (by typename or by allocation site)
   - Suppose that we discover that, in most of the program, one field of a struct supplies the bounds of another field of the struct.
   - Then we guess that this is an invariant and violations of it are errors.
6. If in most callsites of a function `foo(int n, char *p, ...)`,
   the bounds on p is the closed interval `[p, p+n-1]`,
   then propose that in the other callsites, the same bounds should apply.

Dynamic Analysis

(Only applies to repair, not to decompilation of malicious binaries.)

Instrument program to write to log file.

In particular, record which checks are violated, as well as statistics on checks that succeed.

Run the instrumented program

to collect presumed-good traces.

Divide the candidate bounds into three categories:

1. Strongly supported: Many traces where the bounds check succeeded, with values near the bounds, and no failed checks.
2. Likely incorrect: Some traces where the bounds check failed.
3. Indeterminate: Insufficient log data about the check.

Repair the program

by inserting missing bounds checks.

Reading Outside the Valid Portion of an Array

How do we determine which arrays should be subject to this analysis?

- We consider an array to be a ***qualifying array*** if every write to the array is at either index 0 or at the successor of the last written position (LWP).

How do we identify what the ***valid portion*** of the array is?

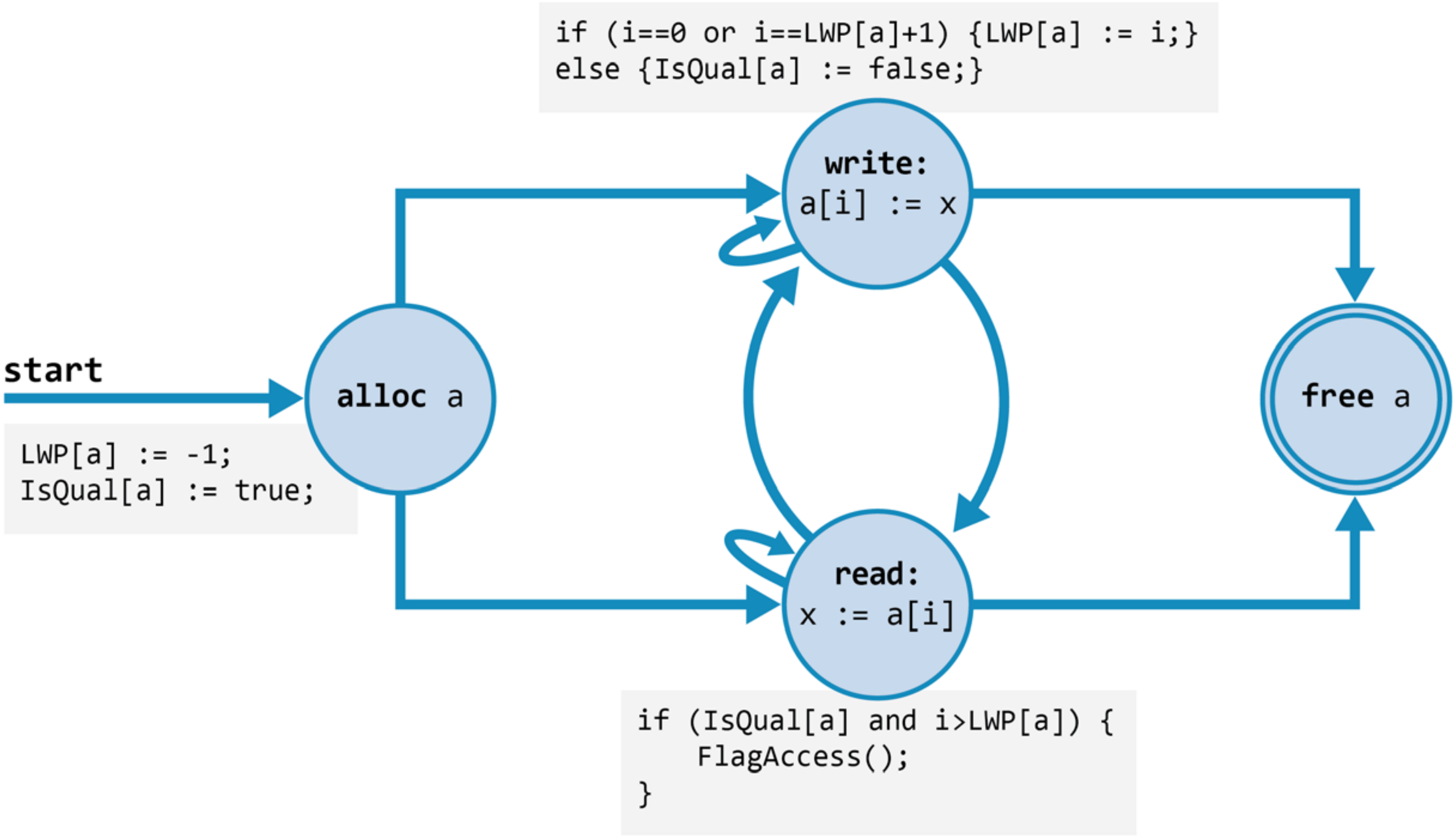
- Heuristic: It is from the start of the array up to and including the last written position of the array.

How often do qualifying arrays occur in real-world programs?

- Imprecision in our static analysis might cause false negatives.
- To establish ground truth, we do a separate dynamic analysis (next slide).

Dynamic Analysis with SAFECode

- SAFECode builds on the LLVM compilation process to:
  - Maintain a side table recording the size and location of allocated memory regions.
  - Check bounds when doing pointer arithmetic and prevent invalid mem accesses.
- **We have extended SAFECode as follows:**
  - Record the allocation site and the last written position (LWP) of each allocated array.
  - Check whether each write to the array is consistent with def'n of *qualifying array*.
  - If all the writes have been qualifying, we flag any reads beyond LWP.
- Note that this dynamic analysis is different than the earlier-described dynamic validation of statically inferred candidate bounds.
- Purpose: (1) Validate static analysis (for project internal), and
  (2) source-to-binary repair (beyond the project).


if (i==0 or i==LWP[a]+1) {LWP[a] := i;}
else {IsQual[a] := false;}
write:
a[i] := x
start
alloc a
LWP[a] := -1;
IsQual[a] := true;
free a
read:
x := a[i]
if (IsQual[a] and i>LWP[a]) {
FlagAccess();
}

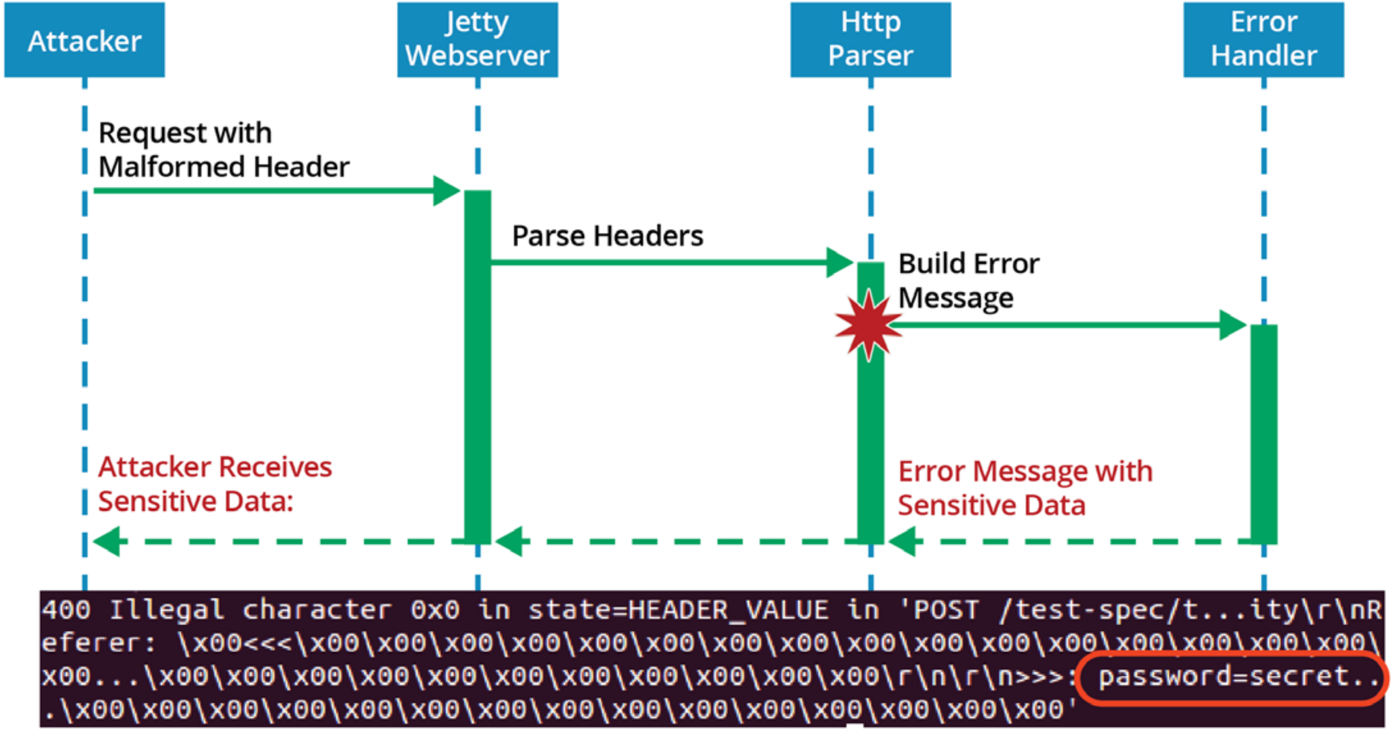
Example with Jetty

As mentioned earlier, Jetty is a web server implemented in Java that leaked passwords (and other sensitive stale information) from a re-used buffer.

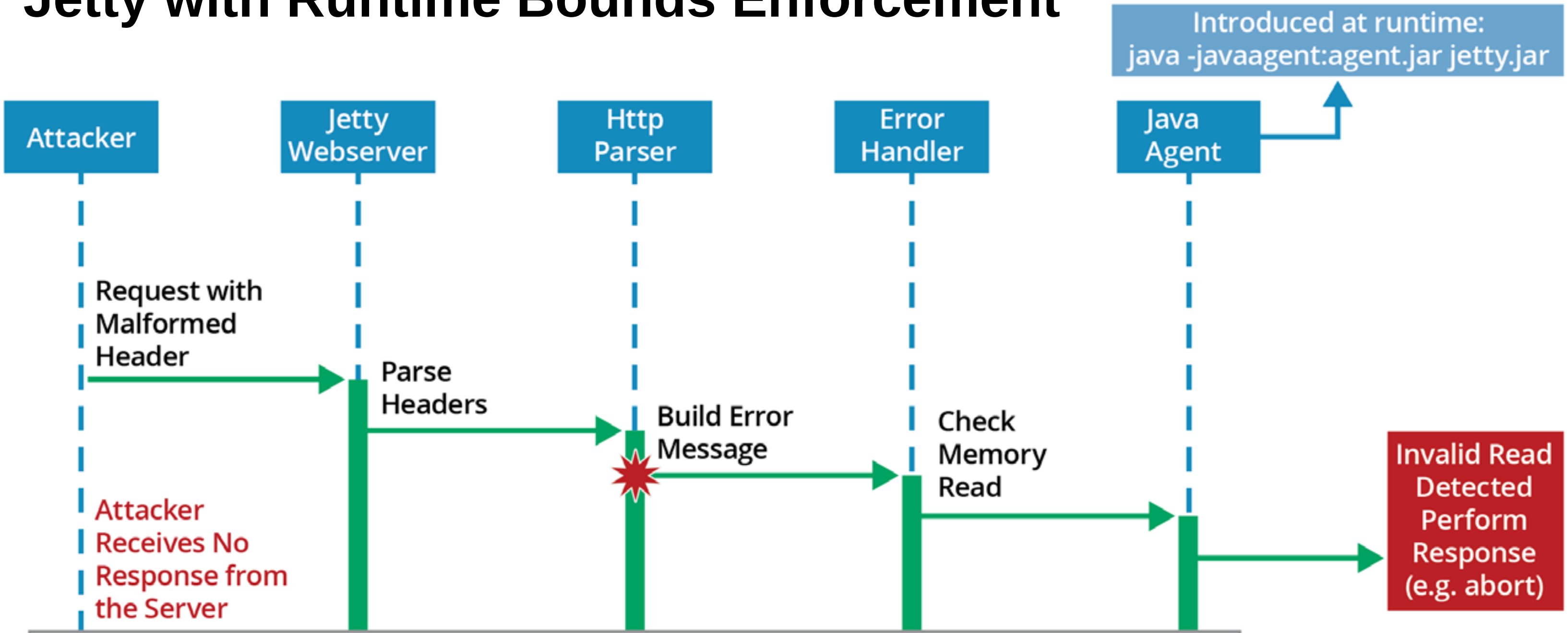
We have also implemented the dynamic analysis (from the previous slide) for Java programs with **`ByteBuffer`**s. (This is implemented via a Java agent.)

With this tool, we dynamically patch Jetty to prevent leakage of sensitive information. (See next slide.)

Unpatched Jetty



Jetty with Runtime Bounds Enforcement




```
2017-08-29 20:23:39.391:INFO:oejs.ServerConnector:main: Started ServerConnector
2017-08-29 20:23:39.410:INFO:oejs.ServerConnector:main: Started ServerConnector
2017-08-29 20:23:39.410:INFO:oejs.Server:main: Started @3227ms

SUSPICIOUS READ -- Buffer: 690171276, Offset: 99, Length:1
Terminating...
```

Conclusion and Future Work

Repair spatial memory violations, with a focus on out-of-bounds READs that leak sensitive information.

This project is the first part of a four-year project on automated repair to enable a **proof** of memory safety.

Proving memory safety is part of a larger thrust in automated code repair. The ultimate goal is enable cost-effective remediation of defects in large DoD codebases.

Contact Information

Presenter / Point of Contact

Will Klieber <weklieber@cert.org>

Software Security Researcher

Contributors

Will Snavely <wsnavely@cert.org>