TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Preventing Memory Error Exploits with WIT (P. Aktridis et al.)

presented by Bjoern Doebel

Dresden, 2008-07-15

Motivation

- Memory errors
  - Buffer overflows
  - Dangling pointers
  - Multiple frees
- Lots of tools
- Still account for nearly half of all security vulnerabilities.
- Crappy tools?
    - Usability
    - Performance Overhead
    - Bug coverage

Reading Group History

- Vigilante [Costa05] & Bouncer [Costa07]
  - Detect and propagate exploits through the internet (self-certifying alerts)
  - Aim: identify malicious input and automatically generate filters
- Data-Flow Integrity Checking [Costa06]
  - Static checking: for each memory location determine the write locations that are allowed to modify it
  - Runtime instrumentation: for each write operation check, whether memory target is in set of allowed writes
  - Up to 100% CPU and 50% memory overhead

Write Integrity Testing (WIT)

- Static Analysis:
  - Compute CFG
  - Determine objects that can be written by each instruction (safe vs. unsafe operations and objects)
- Compiler Extension
  - Insert runtime checking and guard objects
- Runtime Checking
  - Check all unsafe operations using object “colors”

Statically analyzing a memory exploit

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:    int i=0;
6:    while (i < sz) {
7:      cgiCommand[i] = msg[i];
8:      i++;
9:    }
10:
11:   ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Safe vs. unsafe variables

Safe vs. unsafe instructions

Adding instrumentation

- For each unsafe variable p:
  - Calculate p's points-to set
  - Assign individual color {2..256}
  - Assign same color to unsafe operations writing to p
- Merge colors for overlapping points-to-sets
- Generate code checking colors for each unsafe instruction
- Guards between unsafe objects (may still have the same color)

Instrumentation (2)

- Checks only for unsafe write operations
  - Explicitly do not catch out-of-bounds reads
- Wrap heap malloc/free routines to update color table
- Instrument functions to update stack frames before and after function execution
- Separate version of WIT to catch libc exploits
- Similar handling of function pointers

Performance

- SPEC and Olden benchmarks
- 256 colors seem to be enough

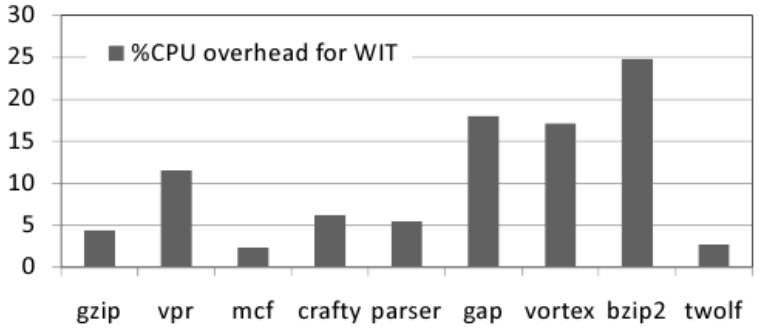

Figure 6. CPU overhead on SPEC benchmarks.

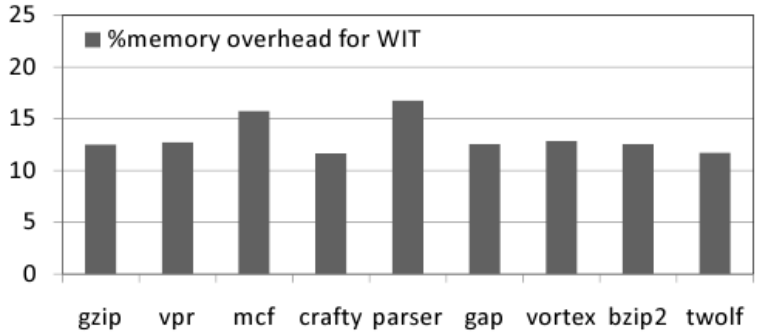

Figure 8. Memory overhead on SPEC benchmarks.

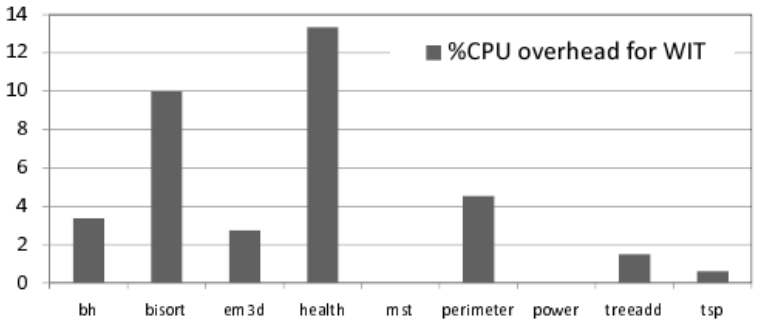

Figure 7. CPU overhead on Olden benchmarks.

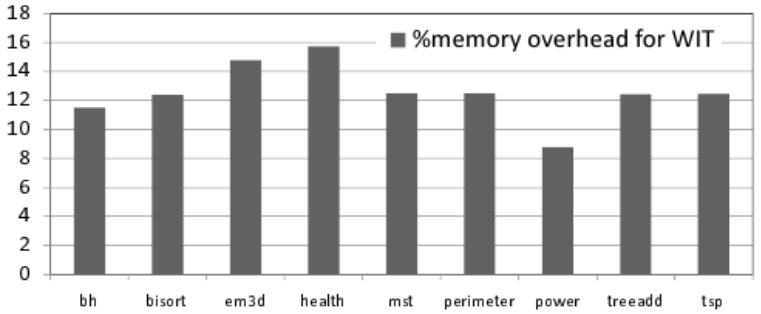

Figure 9. Memory overhead on Olden benchmarks.

Remarks / Questions

- No support for user-defined memory allocators (but easily solved).
- Can we think of an application where 256 colors are insufficient?
  - Need many dynamic allocations of small, independent objects
- Given the fact, that we find unsafe objects during SA, why not just smash the programmer's head with a large wooden hammer? (thx, Marcus)