Targeted Memory Runtime Analysis

David Piché

June 1st, 2023

Polytechnique Montreal

DORSAL Laboratory

Agenda

1. Introduction
2. General approach
3. Our approach using Ptrace
4. Our approach using Libpatch
5. Results
6. Future Work

Introduction

- Memory issues in C/C++ are still prevalent
  - Use-after-free
  - Memory leaks
  - Out-of-bound writes
  - And much more…

The general approach

- We want to verify accesses to dynamically allocated objects

- This means, for the library:
  1. Get control before the access
  2. Verify a valid access
  3. Unprotect the object
  4. Perform the access
  5. Re-protect the object

The general approach : Getting control before access

- By protecting dynamically allocated objects, accesses trigger a SIGSEGV
- We can then handle that signal with a custom signal handler
- Override *malloc/realloc/free* functions to add/remove protection

The general approach : Getting control before access

To protect dynamically allocated objects, we have implemented two methods:

- Pointer tainting using bits 47 to 63
    - System call arguments may be tainted!
        - Requires a kernel patch
- Use mmap() with PROT_NONE flag
    - Currently we allocate an entire page per object

The general approach : Bounds checking

In order to verify the access, use bounds checking

- We need information regarding the memory access:
  - Which register contains the tainted address
  - Information on base, index, scale, offset to compute address for bounds checking

- Use **Capstone** to disassemble instruction and retrieve relevant information

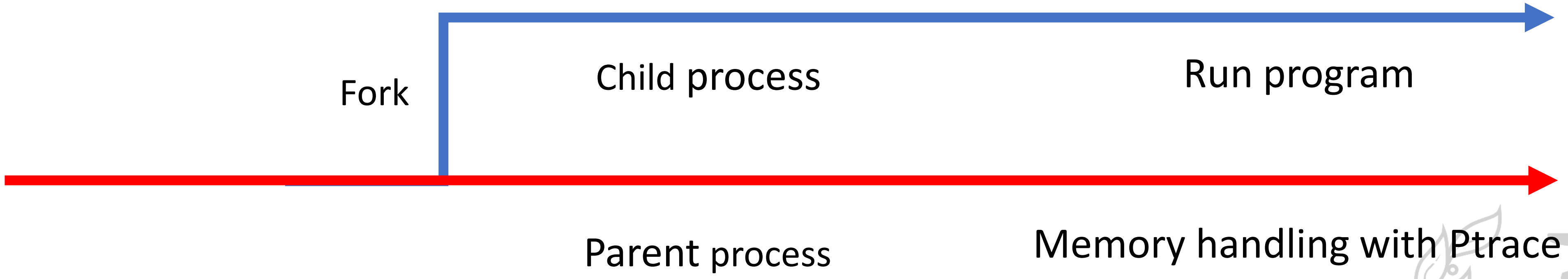
The general approach : Challenges

We still have some remaining challenges:

- How can we re-protect the object after the instruction?
- When using a custom signal handler, what restrictions apply for disassembling code (capstone)?

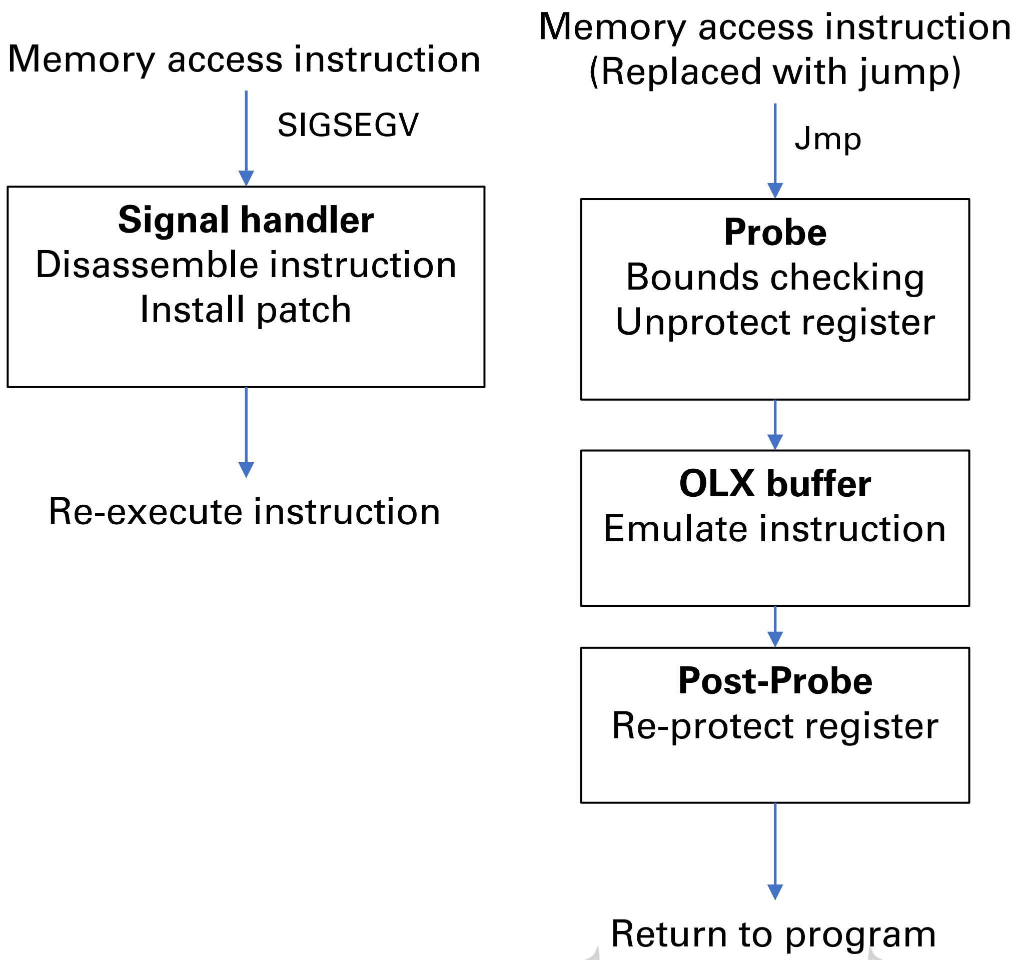
Our Ptrace approach

- Use Ptrace with 2 different processes
  - The child process runs the program with the special allocators
  - The parent process takes care of memory handling
  - Ptrace used for communication between processes and single-step
  - Using the CLONE_VM flag with clone() to make communication between the two threads easier



Our Libpatch approach

- The **Libpatch** library from Olivier Dion specializes in inserting probes at runtime
- Install patch at first encounter of instruction
- OLX buffer emulates instruction
- Post-probe allows us to re-protect address



Our Libpatch approach

- With the patch installed, no need to disassemble the same instruction multiple times
- For programs with repeated instructions with memory accesses, significant performance gain
- Prototype ready, ongoing development
- However, we need to install the patch in the signal handler

Result

We use the SPEC CPU 2017 benchmarks and micro-benchmarks:

- For the 505_mcf benchmark, 11 million tainted memory accesses for only 11k unique heap memory access instructions

- Majority of tainted objects used in those memory accesses are very small in size (< 127)

Future Work

- Finish implementation of our approach using Libpatch
  - Get a clear idea of its overhead

- **Targeted** memory analysis
  - Taint some memory allocations based on parameters (size, origin, …)