Experience. Outcomes.

Data Watch

Presenter Information
Jason Puncher
Date
Oct 27, 2017

Overview

Goal of the techniques discussed in this presentation is to catch invalid memory accesses (read or write) whilst minimizing the impact on both the CPU and memory.

> “Catching the invalid access in the act.”

Keeping in mind the limitations of our embedded systems:

- Most are PowerPC-based
- All are 32 bit.
- File system is either a small flash disk or a tiny ROM
- RAM and CPU is limited
  - i.e. some targets only have 64 MB of RAM

Problem Summary

- Software may intentionally or unintentionally access memory (e.g. read or write) that it shouldn't.
  - NULL pointer dereference, access beyond array scope, or continued access of released memory.
- Many tools exists to catch these types of accesses. But most catch it after the fact. And even few support latent memory access.
- Some tools exist that catch software in the act.

  - **Valgrind** : This tool simulates every instruction the program executes, adding instrumentation to every memory access, and every value computed. Though it will catch the access, it means simulating every instruction (plus instrumentation) resulting in significant runtime performance impact (10-50 times slower execution).

  - **Electric Fence**: Not so much a tool as a technique which requires rounding the number of bytes allocated to the nearest page size. This would have a significant memory impact in software which allocates many small blocks on a system with large page sizes.

Overview

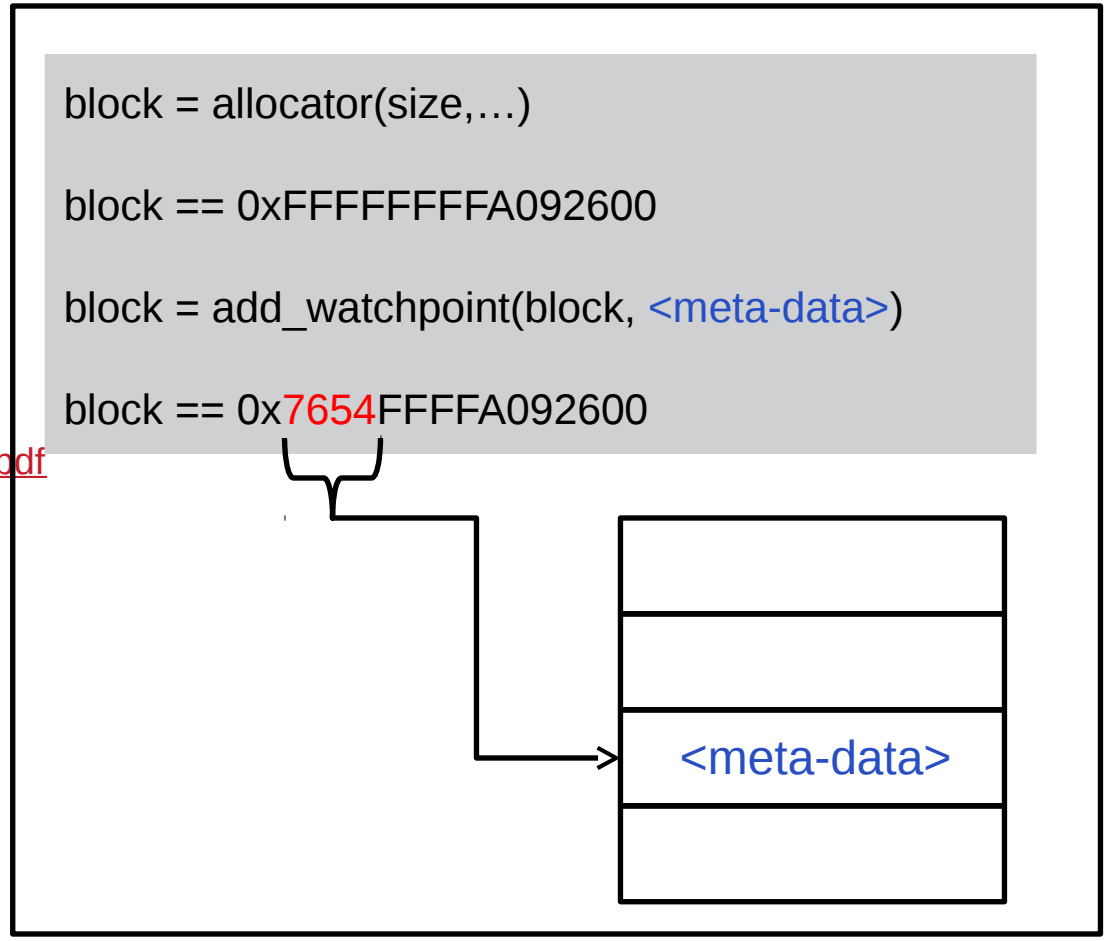
Data Watch will track all accesses to memory allocated from the heap.

- It can detect some stack overwrite conditions, memory block "out-of-bounds" access and latent read-write to previously freed memory.

Violations such as these often result in abnormal system behaviors that can be very hard to debug because the effects are unpredictable and can manifest in ways completely unrelated to the affected sub-systems or code.

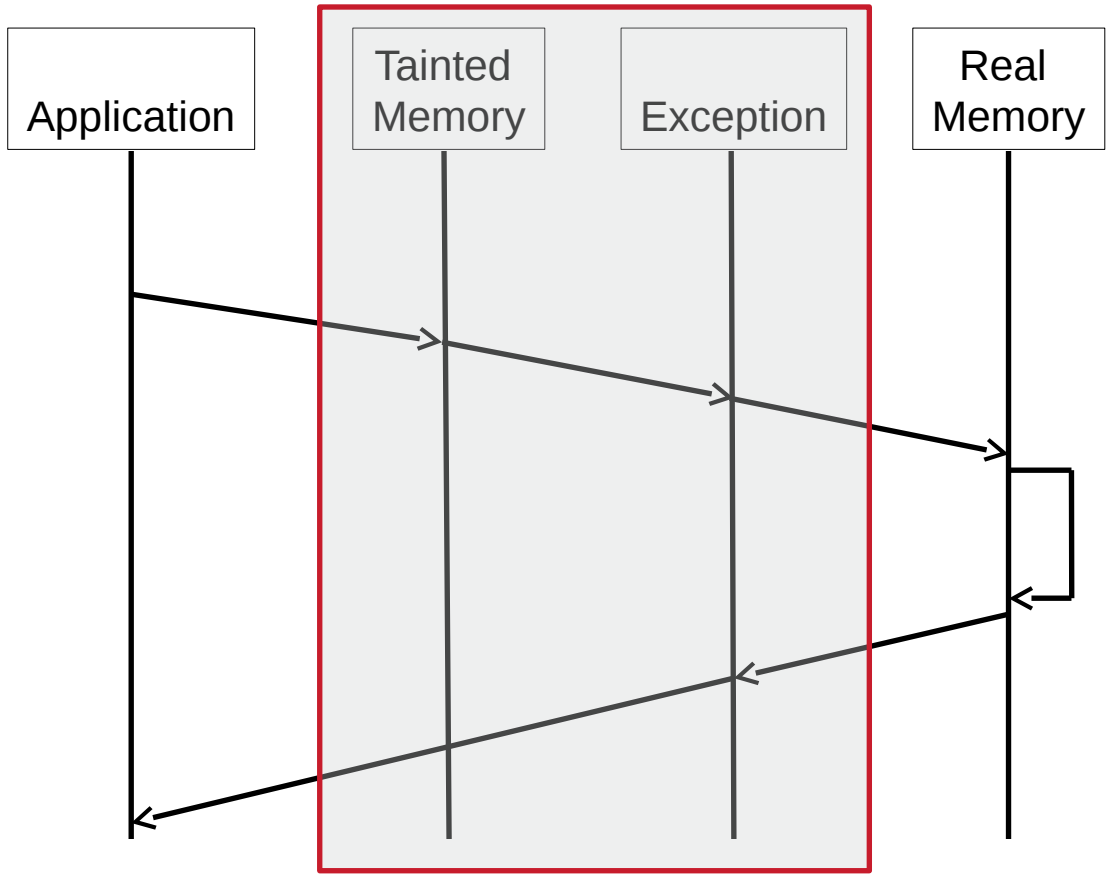
Address Watchpoints (inspiration…)

- Inspired by presentation at L'Ecole Polytechnique (in May 2014).
- Reference material can be found here:
  https://ahls.dorsal.polymtl.ca/system/files/AHLS%20-%20Address%20Watchpoints-%20Instrument%20Data%2C%20Not%20Code.pdf
- Taint object addresses so that accesses to "interesting" objects always raise a fault.
  - "Address watchpoints"
- Relies on x86-64 48-bit address implementation in which 16 bits are "free" to be changed.



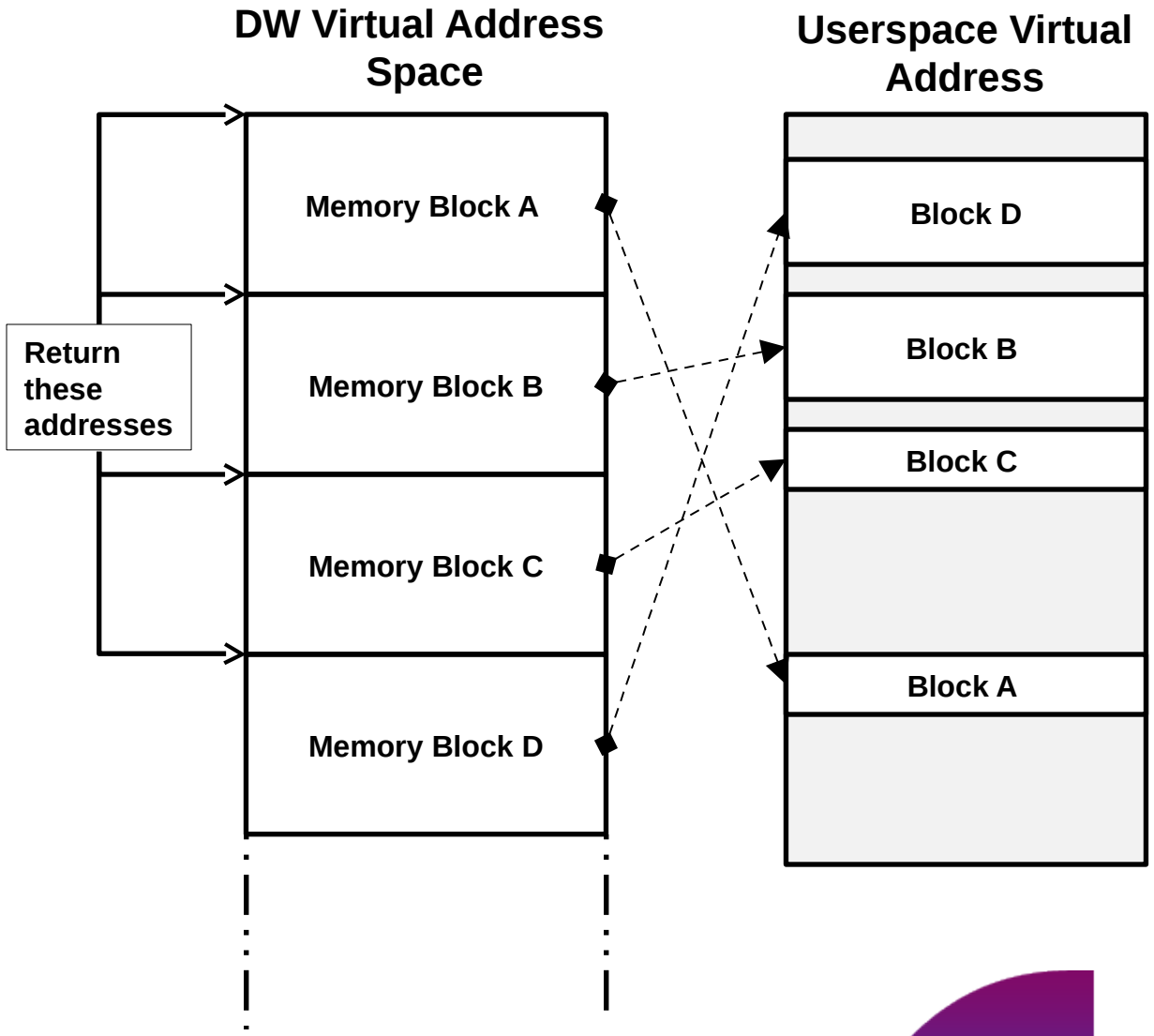
Address Watchpoints (continued…)

- Access to tainted memory will generate an exception.
- The exception handler would then strip the 16 bit taint from the address line and follow through with the access to heap memory.
- Problem: How do we taint the address on a 32 bit system?
  - We do so by mapping the heap memory address to a virtual memory region not mapped by the hardware MMU.
  - In essence, we are creating a software MMU. However, we are not limited by the page size, or having to align the heap address to a page boundary.



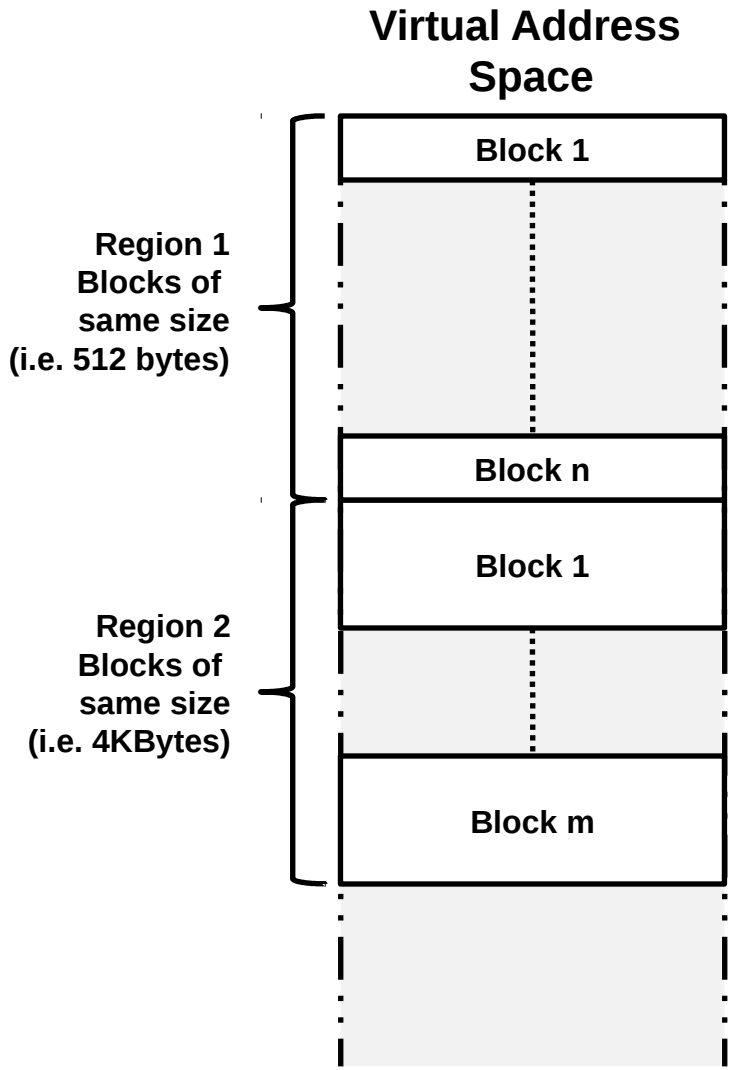
Working with 32-bit applications

- Carve up a large region of unmapped memory into equal sized blocks.
- The memory allocator will :
  - allocate both a block of memory from the heap as well as one of these virtual blocks.
  - update a table mapping the virtual block address to its heap memory address. The table will also include the allocation size.
  - return the pointer to the virtual address.
- The pointer is now tainted. All accesses through this pointer will cause an exception, allowing the tool to sanitize the access.
- If sane, the tool will map the virtual address back to the heap memory address and perform the read/write.



Working with 32-bit applications (continued)

- The limitation to this approach is that the tool can only track allocations whose size is smaller than the virtual block size.
- For this reason we create multiple regions of varying block sizes.
- In our system unmapped memory far exceeds the amount of physical memory. As such, with this approach we are able to create enough regions to cover all memory allocations.



Tool Memory Allocation and Management

- A table is maintained to map the block index back to its userspace virtual address.
- Each entry having :
  - Index of next free block pointer. Value of -1 indicates an allocated block or last free block.
  - Heap address for each block (after allocation).
  - The size of the memory allocation.
- Elements are pulled from the free list using FIFO. It's a best effort way to ensure recently freed blocks are not re-used immediately.
- A mapping table is created for each region.

	Heap Address	Size	Next Free
1	0x1245b248	2944	-1
2	0x1345fe10	0	5
3	0x12560d08	0	-1
4	0x1254ed58	0	142
5	0x123f0a50	3094	-1
256	0x1136ea80	0	96

Free List [head]

Free List [tail]

Table for region 1 (for blk of size X)

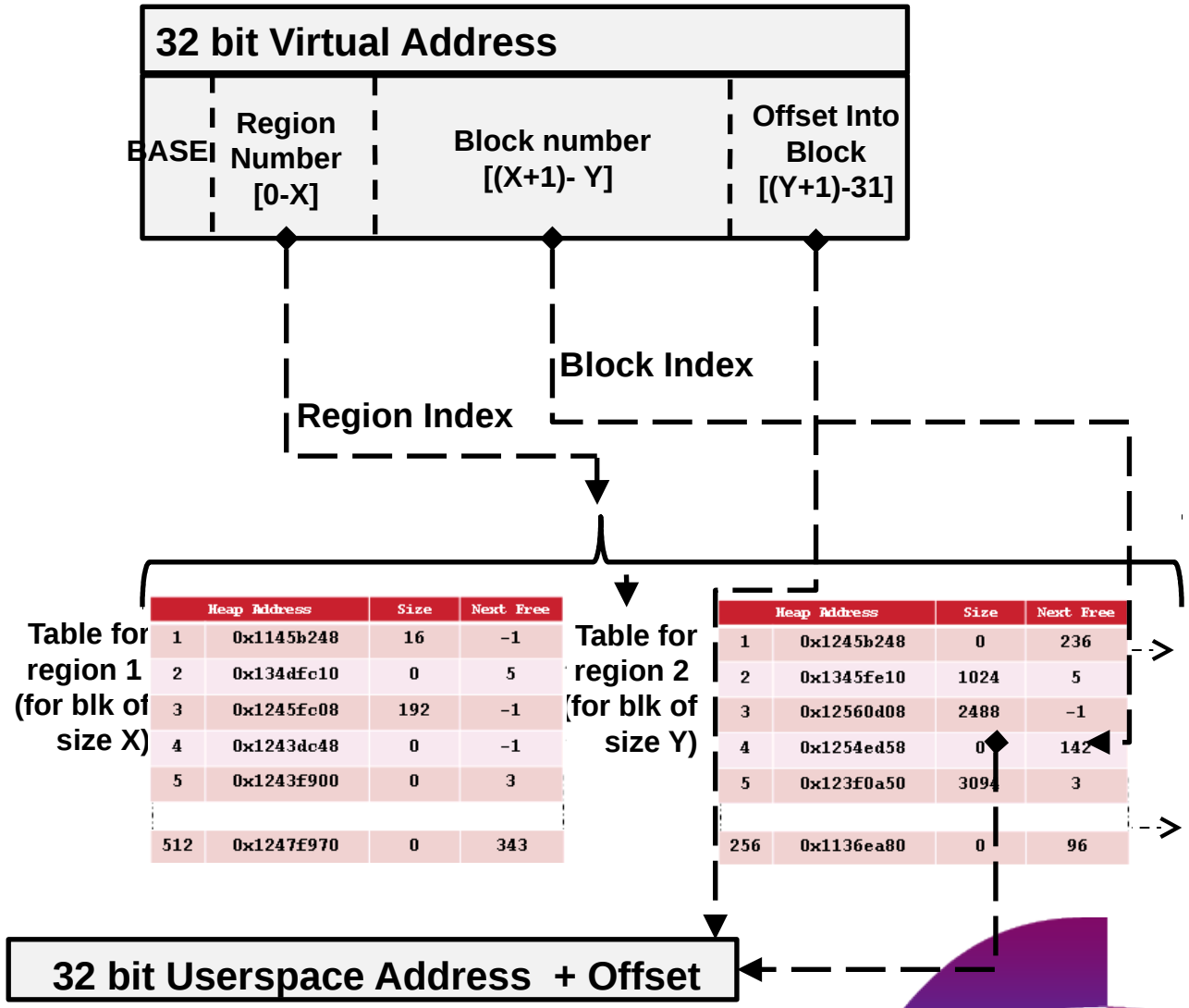
	Heap Address	Size	Next Free
1	0x1145b248	16	-1
2	0x134dfc10	0	5
3	0x1245fc08	192	-1
4	0x1243dc48	0	-1
5	0x1243f900	0	3
512	0x1247f970	0	343

Table for region 2 (for blk of size Y)

	Heap Address	Size	Next Free
1	0x1245b248	0	236
2	0x1345fe10	1024	5
3	0x12560d08	2488	-1
4	0x1254ed58	0	142
5	0x123f0a50	3094	3
256	0x1136ea80	0	96

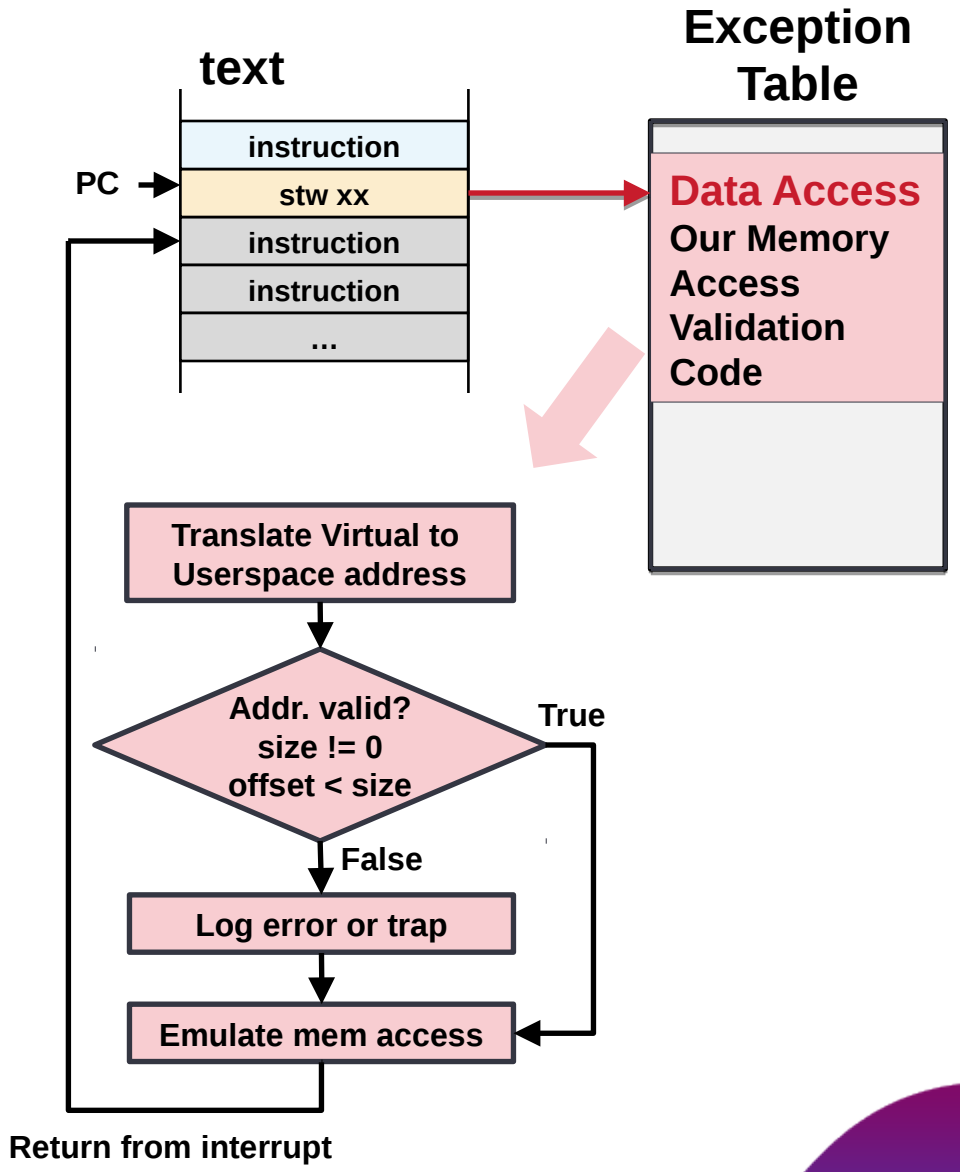
Tool Memory Allocation and Management (continued…)

- In our system it's possible to carve off a 128/256 MB (aligned) blocks of virtual memory. The starting address of this block is our base.
- In it we create equal sized regions (the number of which must be a power of 2).
- We then carve each region into blocks (the size of which must also be a power of 2).
- This allows us to quickly determine which region we are accessing and which block (within the region) based on the bits in the address.



Runtime Execution

- Access to an unmapped virtual address generates a Data Access Exception (DSI).
- Exception handler calculates block index, and offset into block.
- Block index is used to locate metadata. With it,
  - it can determine if the offset is greater than the allocated size.
  - or if the block is still allocated.
  - optionally, the memory could be pre-filled with a known pattern to see if we have a read before write access.
  - map the block + offset to the heap address
- Log Error or trap if address is invalid.
- The exception handler must then emulate the load/store using the heap address.



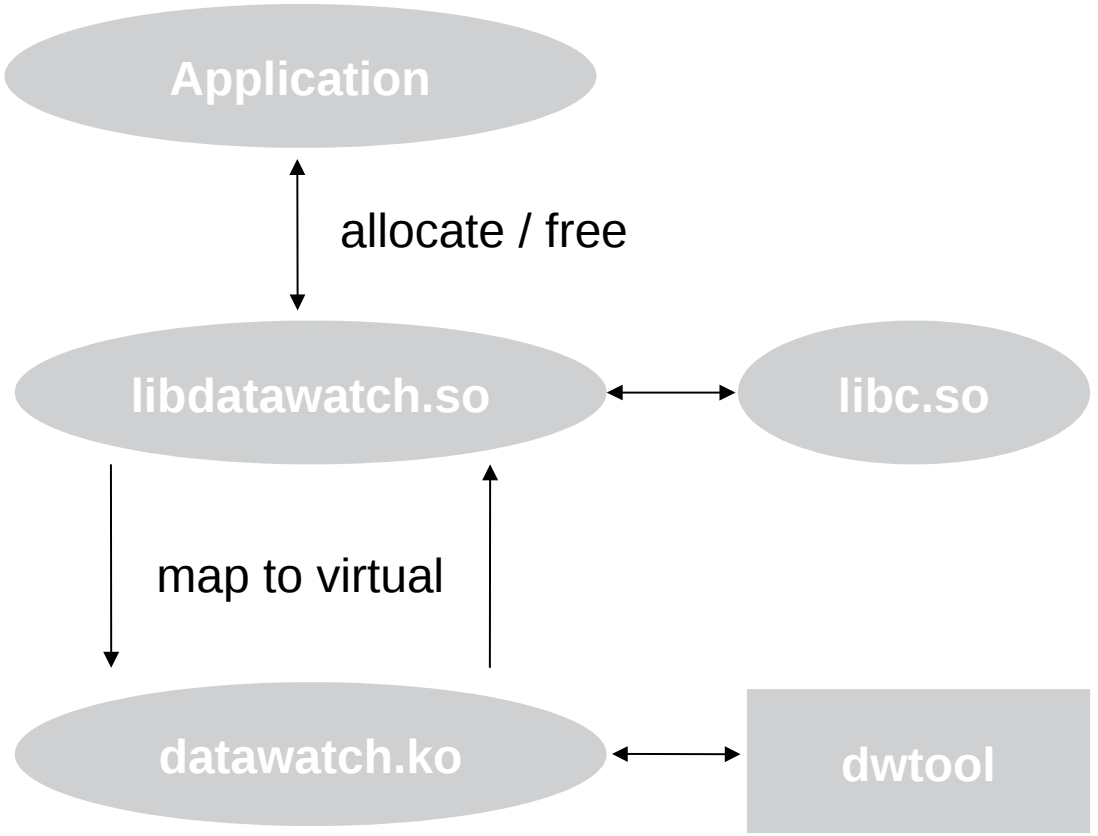
Linux Implementation

Comprised of three separate components:

1. Share library which, using LD_PRELOAD, replaces all the memory allocation/free routines.

2. Kernel driver which maintains the mapping table, handles the exception, and emulates the load/store instruction.

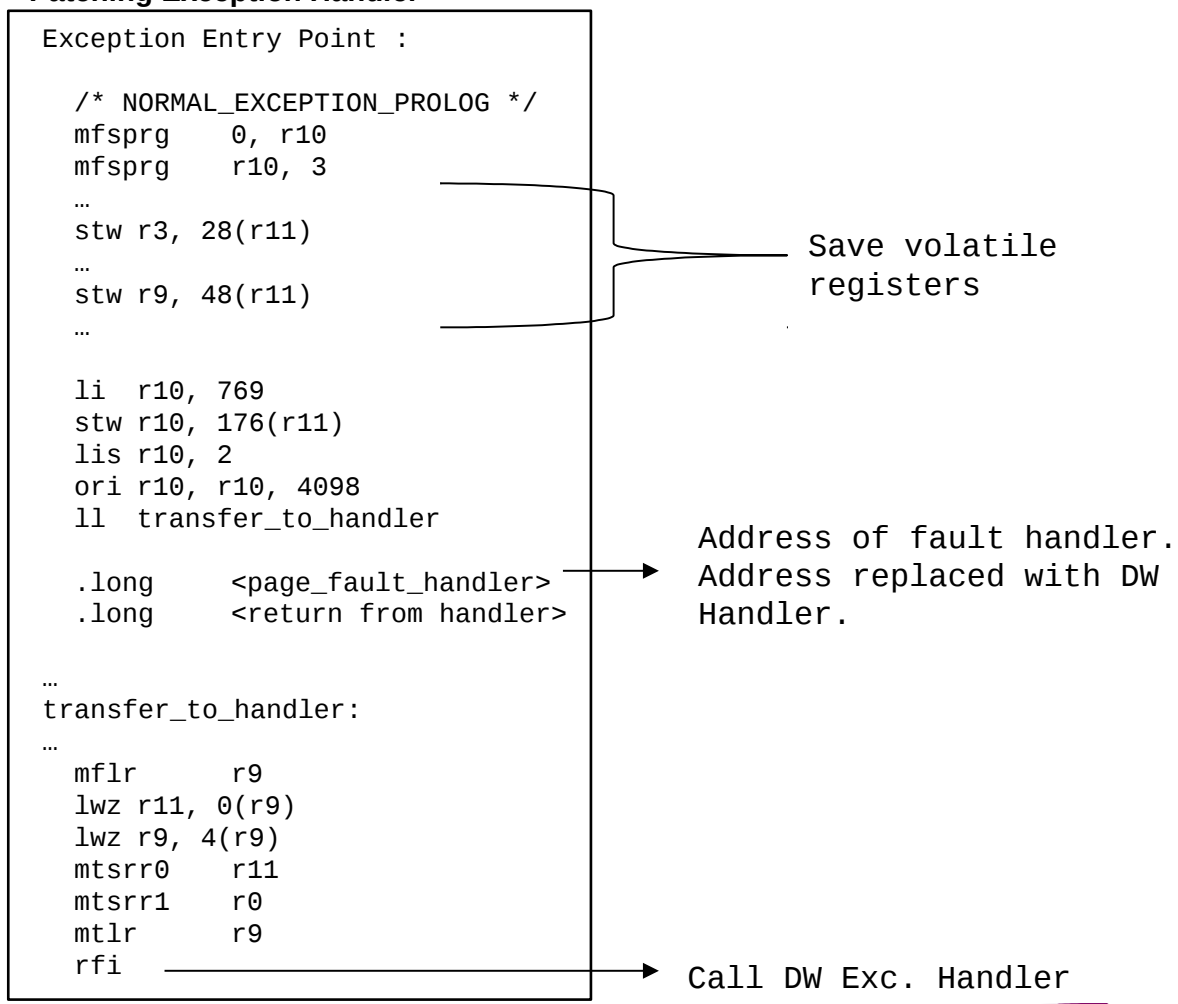
3. A utility to interact with the statistics and logs updated and maintained by the kernel driver.



Linux Implementation – Kernel Module – Patching on PowerPC e500(mc)

- Kernel exception handler is patched at the location where the address of the fault handler is stored.
- The fault handler must be re-entrant, as access to the user space address may cause a page fault.
- Handler must check if secondary exceptions occurred inside DW exception handler and fetch the page itself. Otherwise the default handler will trap if the initial exception happened while in the kernel context.

Patching Exception Handler



Other applications

- This tool has been expanded to support:
  - Stack linkage scribbler detection
  - Watch specific memory address range
  - Read before Write detection
  - Memory Utilization Tracking

Thank You