ORACLE

vDSO and BPF – running uprobe BPF programs in userspace

Linux Plumbers 2024

Alan Maguire
Linux Kernel Networking, Oracle
September 2024

The problem

- Many userspace tracing BPF progs read predicates and compare
  - is pid == 1234??
  - is execname == "foo"?
  - etc
- Even when predicate fails, we pay the cost of a trap into kernel to execute the attached BPF program that evaluates that predicate.
- When the probe is placed on a hot codepath, even the predicate failure is expensive.
- Is there an approach that supports BPF execution in userspace such that predicates and simple programs can be run without *repeated* trap cost?

The problem

- If we have to trace system-wide on a hot shared library function, overheads are significant.
- Example: trace malloc()s in libc systemwide (pid == -1)
- Common pattern in tracing; start broadly, before drilling down to specific processes
- When we need to place a trap on all calls systemwide, overheads can be significant

The problem

- Example: with a systemwide uprobe on libc getpid(); time 10,000,000 getpid() calls (https://github.com/alan-maguire/trap-overhead)
- Baseline, no uprobe attached to getpid

```
$ time ./getpid 10000000
real      0m0.946s
user      0m0.341s
sys       0m0.602s
```

- With simple “return 0” uprobe BPF program attached to getpid systemwide(pid == -1)

```
$ time ./getpid 10000000
real      0m11.077s
user      0m2.324s
sys       0m8.740s
```


The goal

- Run a reasonable subset of uprobe/USDT BPF programs in userspace
- As with trap-based uprobes, support attachment to
  - a specific target proccess
  - Systemwide (e.g. all libc malloc()s)
- Make the attachment process as smoothly as possible for tracers, not requiring much additional work to get programs working in userspace
- Lean on existing kernel mechanisms where possible
- As with other probe optimizations, fall back to other methods if it is not feasible to run the program fully in userspace

Health warning

- This is a speculative discussion to try and figure out what is needed to support userspace BPF execution upstream.
- I am not an expert in many of these subsystems, so if you are thinking “there’s an easier way of doing that”, you’re probably right!
- The aim here is just to figure out the best way to solve this problem by working with the relevant subsystems, *if the solution makes sense*.
- The first step though is to figure out if this solution even makes sense from a BPF perspective.

But first… Why?

- There has been a bunch of work done improving u[ret]probe performance in the general case.
- Is running uprobe/USDT BPF programs in userspace enough of a performance win to justify the work required?
- That remains to be seen (still working on a prototype!)
- One interesting side-effect of the existing uprobe optimization work has been that more user-space side involvement has been required, e.g. uretprobe syscall trampoline

Uprobes and user-space execution

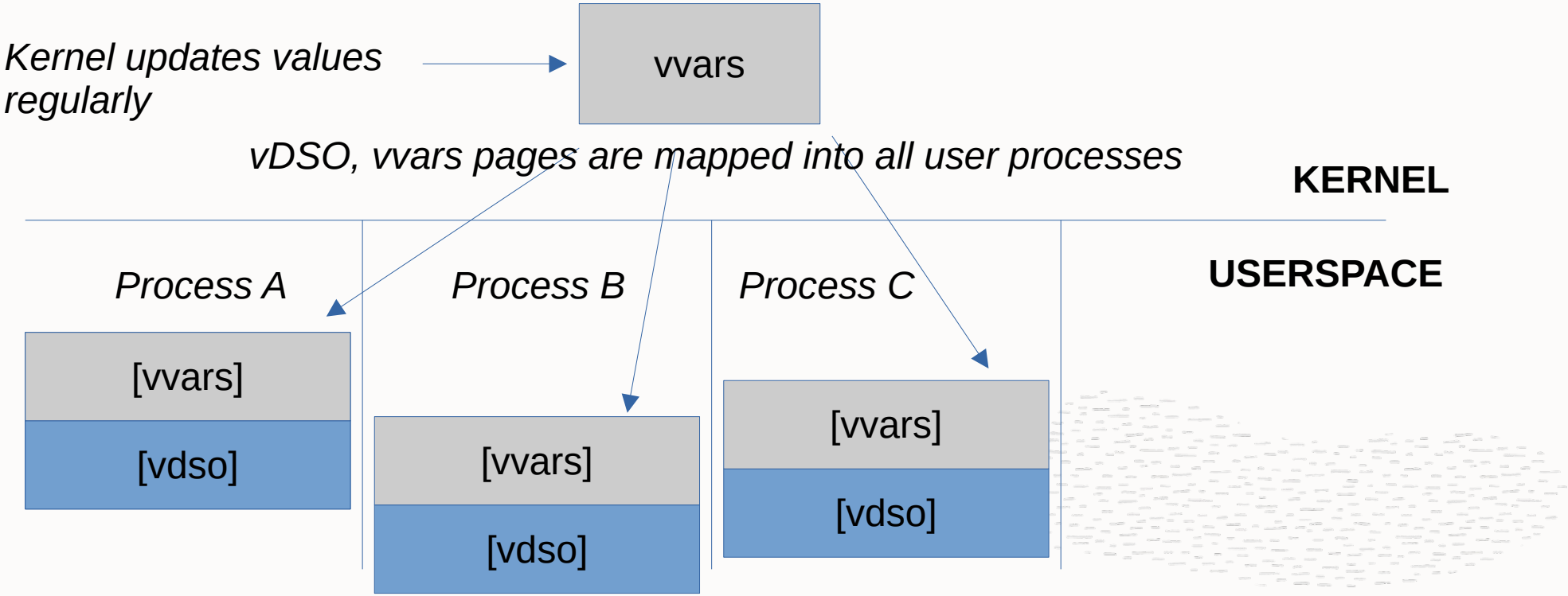
- As part of uprobe setup, we install a Virtual Memory Area (VMA) in userspace to host the out-of-line (xol) instructions we instrumented with a breakpoint
- …and more recently, the uretprobe syscall code.
- Future work envisions a uprobe system call also to speed up uprobe execution.
- We are already inserting userspace code into traced processes; how far can we take this concept?
- vDSO provides a potential model for the possibilities here

Quick vDSO primer

- vDSO – virtual Dynamic Shared Object provides a means to reduce system call overheads by eliminating them where possible.
- It is a shared library .so object built by the kernel and delivered with the kernel. It is added into the address space of processes via two virtual memory areas (VMAs)
- - [vdso] for executable code associated with retrieving syscall-related data; i.e. the library code itself
- - [vvars] for variables used by the kernel to write data and by vdso functions to read that data

```
$ cat /proc/self/maps
5638cd000000-5638cd008000 r-xp 00000000 fc:01 67197593                   /usr/bin/cat
5638cd207000-5638cd208000 r--p 00007000 fc:01 67197593                   /usr/bin/cat
5638cd208000-5638cd209000 rw-p 00008000 fc:01 67197593                   /usr/bin/cat
5638ce311000-5638ce332000 rw-p 00000000 00:00 0                          [heap]
….
7ffdb58a5000-7ffdb58a9000 r--p 00000000 00:00 0                          [vvar]
7ffdb58a9000-7ffdb58ab000 r-xp 00000000 00:00 0                          [vdso]
```

Quick vDSO primer



Quick vDSO primer

- glibc calls getauxval(AT_SYSINFO) to get vDSO base address (setup_vdso())
- Dynamically links in vDSO functions assigning function pointers (setup_vdso_pointers())
- Calls the functions for the appropriate syscall wrapper when available
- vDSO supports
  - Timekeeping (via updated values in vvars)
  - Retrieving CPU id (via register)
  - Random number generation

What can we learn from vDSO?

- Mechanism to deliver shared library-like functionality
  - A place to host BPF helper functions (either in [vdso] or in a similarly-created .so)
  - Some overlap between BPF helpers and vDSO functions already (timekeeping, random number generation)
  - Delivered along with the kernel (as BPF helper support is)
- Uses VMAs for code [vdso] and data [vvars]
  - For BPF progs, we would need a rwx VMA to hold mmap()ed BPF maps
  - VDSO installs vvars directly adjacent to vDSO code that uses it
  - Can use relative addressing with such an arrangement
- vDSO saves system calls; can we avoid syscalls when running in userspace?
  - Not much point jumping through all these hoops just to execute a syscall anyway!

What are we trying to “vDSO”?

- vDSO is about avoiding system calls to minimize overhead where possible
- Here we are trying to avoid trapping into the kernel to fire a uprobe
- We are in a sense trying to vDSO *uprobe BPF execution itself* , *avoiding trap/syscall overhead*.

Steps to consider

- How do we trigger the program?
- Once triggered how do we run the BPF program safely in userspace?
- What limitations apply to executing BPF programs in userspace?
- We will *not* reinvent BPF verification!
  - BPF verification as usual
  - Relocations should be used to handle JITing for execution in userspace
  - Some programs will just not be eligible, and that’s okay
  - A subset of helpers, mmap()able maps required
  - No kfuncs
  - This would allow simple programs/predicates, such as “count all malloc()s system-wide”

uprobe firing – from trap to trampoline

- Currently, we replace the instruction at the instrumentation point by a trap
- This needs to become a *call* to a userspace trampoline instead
- Need >= 5 bytes instead of 1 byte for trap
- Likely spanning multiple instructions
- For 32-bit relative calls, target needs to be within 32-bit address range of probe point

uprobe firing – a vDSO trampoline

- Today we create 1-page [uprobe] VMA in __create_xol_area()
- Called as part of prepare_uretprobe()
- Created as high as possible in 48-byte address space

```
7fbdb924c000-7fbdb9251000 rw-p 00000000 00:00 0
7fff96439000-7fff9645b000 rw-p 00000000 00:00 0                          [stack]
7fff965d6000-7fff965da000 r--p 00000000 00:00 0                          [vvar]
7fff965da000-7fff965dc000 r-xp 00000000 00:00 0                          [vdso]
7fffffffe000-7ffffffff000 --xp 00000000 00:00 0                          [uprobes]
```


uprobe firing – a vDSO trampoline

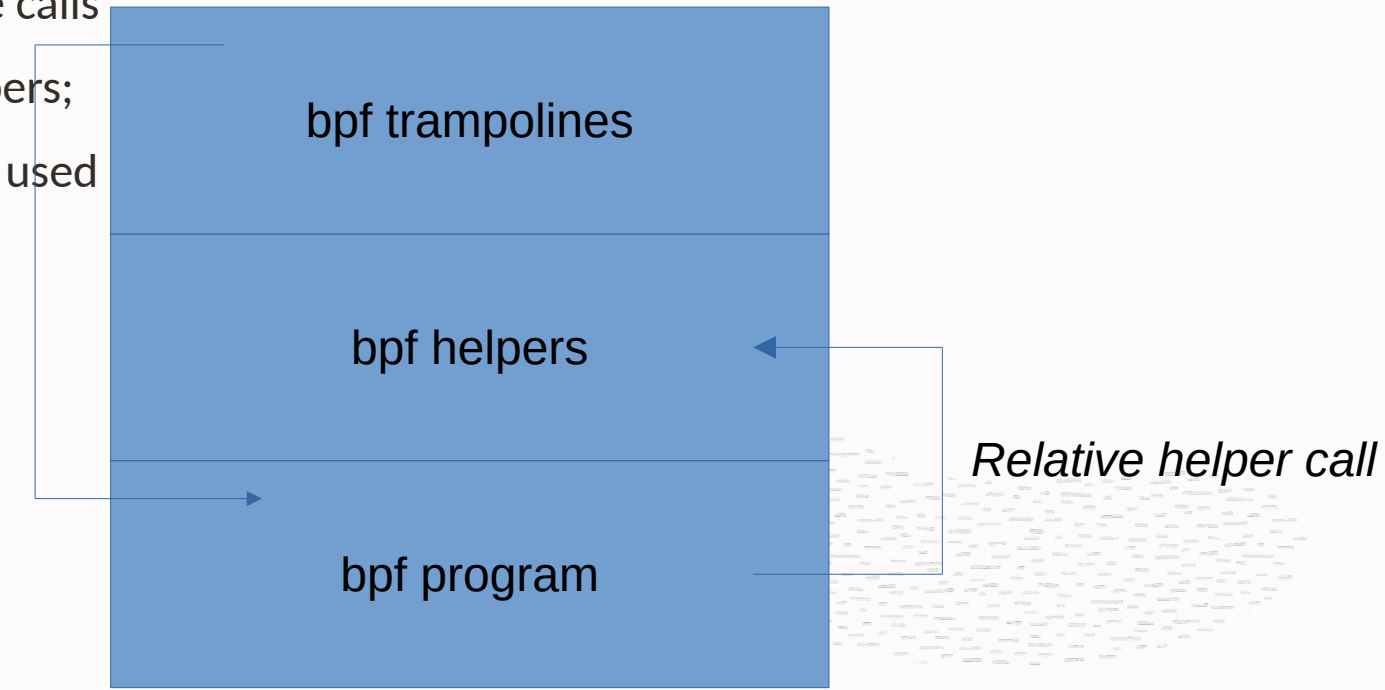
- Modify this to create as high as possible, *but within 32-bit relative address call range of probe site*
- What if we cannot get address range? Graceful degradation; drop back to using trap/syscall
- Have original in-kernel JITed program available for fallback cases
- [uprobe] area could contain trampoline, xol area, and **BPF helpers/programs**

uprobe firing – challenges

- Today we just modify one byte to insert a trap
- Adding a 5 byte “call” will span multiple instructions on x86_64
- Incomplete state could cause problems
- Solution: we place trap, update the other 4 bytes, finally replace the trap with the call instruction
- Means we trap into kernel if we hit the probe while setting things up/tearing down
- We have a pair of progs; a JIT-ed for kernel and a JIT-ed for userspace version, equivalent in operation
- This is required for fallback case too, i.e. where we cannot create VMA in address range

uprobe BPF execution – executable VMA

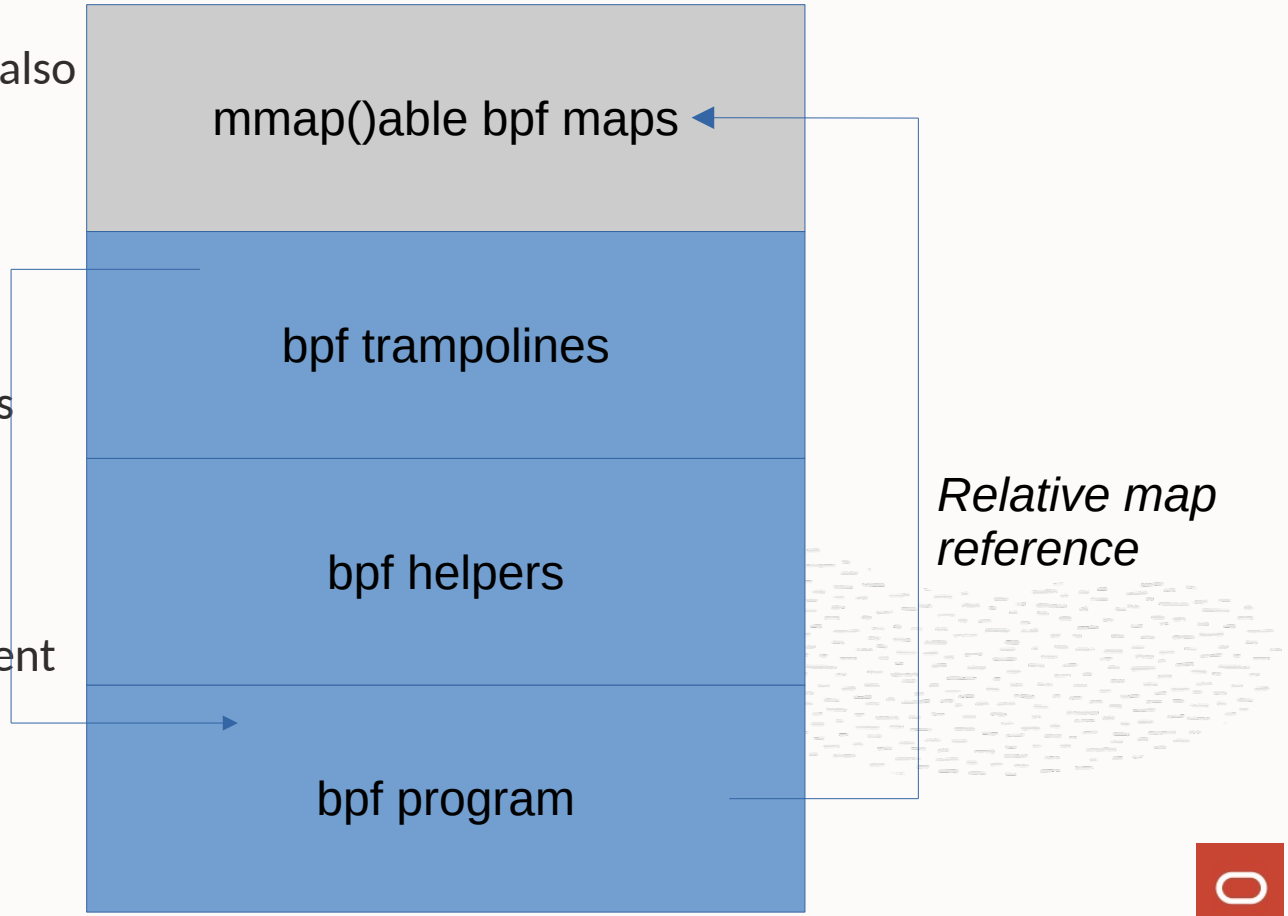
1) Trampoline prepares struct pt_regs, jumps to BPF program
2) Helper calls become relative calls
3) Program(s) added after helpers; trampoline/program can be used in multiple process address spaces unchanged as all memory references are relative



uprobe BPF execution – R/W VMA (maps)

4) Map references can be relative also so that the identical program is relocatable to other processes.

Only difference between processes is relative call at probe site (since each process can map to a different VMA address). Only requirement is adjacent VMAs for maps/progs



uprobe BPF execution – BPF maps

```
$ cat /proc/self/maps
7f99167cf000-7f99167d3000 rw-p 00000000 00:00 0                          [uprobe_maps]
7f99167d3000-7f99167db000 r-xp 00000000 00:00 0                          [uprobe]
```

uprobe BPF execution – BPF program

- Requires a process of JITing to a user-space capable version of the program
- At attach time, evaluate if program is eligible
- Relocate calls to BPF helpers to relative calls to userspace equivalents
- Relocate map accesses to relative address of map(s)
- Make use of vDSO for BPF helpers if possible; crossover between vDSO-ed – and vDSO-able helpers
  - bpf_get_smp_processor_id() -> __vdso_getcpu()
  - bpf_ktime_get_ns() -> __cvdso_clock_gettime()
  - bpf_get_prandom_32() - __cvdso_getrandom()
  - Map helpers: accessors for BPF maps

Summary

- Uprobe setup already adds a VMA to userspace processes already to aid tracing
- We will likely need to move to using call anyway to facilitate uprobe syscall trampoline
- Taking this process further, we can perhaps learn from the vDSO approach
- Both as an analogy, and potentially directly use vDSO functions to support traps/emulate helper execution in userspace

References

- Demonstrating trap overheads for uprobes https://github.com/alan-maguire/trap-overhead
- Original uprobe cover letter which nicely explains the mechanisms https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=654443e20dfc0617231f28a07c96a979ee1a0239
- What are vDSO and vsyscall in Linux? https://www.baeldung.com/linux/vdso-vsyscall-memory-regions
- Implementing virtual system calls https://lwn.net/Articles/615809/
- bysyscall; bypassing system calls via eBPF. From eBPF summit 2024 https://youtu.be/DdhGhvYr9sA?si=lOPWmfB_XFJemxW5


ORACLE