Programmable dynamic tracing on Linux with DTrace using BPF

dr. Kris Van Hees
Consulting Engineer, Languages and Tools
Linux Engineering
September 18, 2023

Overview

1. Short overview of DTrace on Linux
2. Programmable dynamic tracing:
   - D scripts compiled into BPF programs
3. The ‘joy’ of product status…
4. .. and other challenges
5. Features we need but do not have in upstream kernels (yet)
6. More information… Get involved...

Short overview of DTrace on Linux

- DTrace for Linux started in 2010
- Between 2018 and 2020, DTrace transitioned from an invasive kernel/userspace implementation to a pure userspace implementation (based on kernel tracing features incl. eBPF).

- DTrace provides:
  - Combined kernel space and userspace tracing
  - C-style scripting language
  - Higher level data structures (strings, arrays, associative arrays, aggregations)
  - Scripted actions associated with probes
  - Speculative tracing
  - ...

Programmable dynamic tracing

- DTrace provides **programmable** tracing:
  - Code is written in clauses
  - Clauses are associated with probes and act like functions, executed when the probe fires
  - Predicates provide conditional clauses
  - Any number of clauses can be associated with a probe
  - Any number of probes can be associated with a clause
- DTrace provides **dynamic** tracing:
  - Scripts written in D
  - Can adapt to trace data

D scripts are compiled into BPF programs

```
fbt::wake_up_new_task:entry
{
    self->p = (struct task_struct *)arg0;
    euids[self->p->tgid] = self->p->cred->euid.val;
    comms[self->->tgid] = (string)self->p->name;
}

sched_process_exit
{ euids[pid] = 0; }

proc:::exit
{ comms[pid] = 0; }

execve:entry
{
    this->in_execve = 1;
    this->uid = 0;
}
```

```
execve:return
{
    this->in_execve = 0;
}

path_openat:return
/this->in_execve && arg1 > 0 && arg1 < 4096/
{
    this->uid = ((struct file
*)arg1)->f_inode->i_uid.val;
}

path_openat:return
/this->execve && arg1 && comms[ppid] != 0 &&
 this->uid != 0 && this->uid != euids[ppid]/
{
    printf("…");
}
```

D scripts are compiled into BPF programs

```
fbt::wake_up_new_task:entry
{
    self->p = (struct task_struct *)arg0;
    euids[self->p->tgid] = self->p->cred->euid.val;
    comms[self->->tgid] = (string)self->p->name;
}

sched_process_exit
{ euids[pid] = 0; }

proc:::exit
{ comms[pid] = 0; }

execve:entry
{
    this->in_execve = 1;
    this->uid = 0;
}
```

FBT providet

```
execve:return
{
    this->in_execve = 0;
}

path_openat:return
/this->in_execve && arg1 > 0 && arg1 < 4096/
{
    this->uid = ((struct file
*)arg1)->f_inode->i_uid.val;
}
```

FBT providet

```
path_openat:return
/this->execve && arg1 && comms[ppid] != 0 &&
 this->uid != 0 && this->uid != euids[ppid]/
{
    printf("…");
}
```

FBT providet

D scripts are compiled into BPF programs

```
fbt::wake_up_new_task:entry
{
    self->p = (struct task_struct *)arg0;
    euids[self->p->tgid] = self->p->cred->euid.val;
    comms[self->->tgid] = (string)self->p->name;
}
```

FBT providet

```
sched_process_exit
{ euids[pid] = 0; }
```

tracepoint

```
proc:::exit
{ comms[pid] = 0; }

execve:entry
{
    this->in_execve = 1;
    this->uid = 0;
}

execve:return
{
    this->in_execve = 0;
}

path_openat:return
/this->in_execve && arg1 > 0 && arg1 < 4096/
{
    this->uid = ((struct file
*)arg1)->f_inode->i_uid.val;
}
```

FBT providet

```
path_openat:return
/this->execve && arg1 && comms[ppid] != 0 &&
 this->uid != 0 && this->uid != euids[ppid]/
{
    printf("…");
}
```

FBT providet

D scripts are compiled into BPF programs

```
fbt::wake_up_new_task:entry
{
    self->p = (struct task_struct *)arg0;
    euids[self->p->tgid] = self->p->cred->euid.val;
    comms[self->->tgid] = (string)self->p->name;
}
```

FBT providet

```
sched_process_exit
{ euids[pid] = 0; }
```

tracepoint

```
proc:::exit
{ comms[pid] = 0; }
```

```
execve:entry
{
    this->in_execve = 1;
    this->uid = 0;
}
```

Syscall provider

```
execve:return
{
    this->in_execve = 0;
}
```

Syscall provider

```
path_openat:return
/this->in_execve && arg1 > 0 && arg1 < 4096/
{
    this->uid = ((struct file
*)arg1)->f_inode->i_uid.val;
}
```

FBT providet

```
path_openat:return
/this->execve && arg1 && comms[ppid] != 0 &&
 this->uid != 0 && this->uid != euids[ppid]/
{
    printf("…");
}
```

FBT providet

D scripts are compiled into BPF programs

```
fbt::wake_up_new_task:entry
{
    self->p = (struct task_struct *)arg0;
    euids[self->p->tgid] = self->p->cred->euid.val;
    comms[self->->tgid] = (string)self->p->name;
}
```

FBT providet

```
sched_process_exit
{ euids[pid] = 0; }
```

tracepoint

```
proc:::exit
{ comms[pid] = 0; }
```

proc-provider

```
execve:entry
{
    this->in_execve = 1;
    this->uid = 0;
}
```

Syscall provider

```
execve:return
{
    this->in_execve = 0;
}
```

Syscall provider

```
path_openat:return
/this->in_execve && arg1 > 0 && arg1 < 4096/
{
    this->uid = ((struct file
*)arg1)->f_inode->i_uid.val;
}
```

FBT providet

```
path_openat:return
/this->execve && arg1 && comms[ppid] != 0 &&
 this->uid != 0 && this->uid != euids[ppid]/
{
    printf("…");
}
```

FBT providet

D scripts are compiled into BPF programs (cont.)

- Each clause is compiled into a BPF function
- DTrace probes are mapped to kernel-level probes:
  - FBT probes are mapped to kprobes
  - Syscall probes are mapped to syscall entry and return tracepoints
  - Profile probes are mapped to perf timer events
  - USDT and pid probes are mapped to uprobes
  - SDT probes (proc, sched, lockstat, …) are mapped to any other probe
    - Sometimes a single probe, sometimes multiple probes
    - Sometimes multiple probes working together (e.g. one does setup, the other reports the firing)
- Common subroutines are implemented using pre-compiled BPF code
  - Leveraging BPF support in GCC and binutils

D scripts are compiled into BPF programs (cont.)

- A BPF program is generated for each kernel-level probe:
  - BPF program types vary across different kernel-level probes
    - BPF programs are specific to a certain program type
  - DTrace considers all probes to be essentially the same
    - Differences are reflected in naming (irrelevant) and probe arguments
  - A **single clause** associated with **two probes of different BPF program type** requires **two BPF programs**.
  - DTrace probes implemented on top of other DTrace probes need to appear to the consumer as distinct probes.

D scripts are compiled into BPF programs (cont.)

- The BPF program for a specific BPF probe is generated as a trampoline:
  - Exit immediately if the consumer has not started yet **(global on/off switch)**
  - Create a DTrace context:
    - Populate probe arguments based on the BPF context (program type specific)
    - Set up the DTrace context based on the DTrace probe information
    - Set up the output buffer and other internal pointers and data structures
  - Generate calls to all clause BPF functions for this kernel-level probe, checking before each call whether tracing is still active **(global on/off switch)**
  - For each (if any) (dependent) DTrace probe implemented based on this (underlying) probe:
    - Save the probe arguments
    - Morph the DTrace context into the dependent DTrace probe and call its clauses
    - Restore the probe arguments

D scripts are compiled into BPF programs (cont.)

- The final (loadable) BPF program is then constructed using a custom linker:
  - Recursively resolve all function call references by appending the generated code for the function (clause BPF function or pre-compiled BPF function) to the program and patching the call target offset
  - Resolve all symbolic references to constants that are compilation-specific

D scripts are compiled into BPF programs (cont.)

It all sounds so easy…

D scripts are compiled into BPF programs (cont.)

It all sounds so easy…

too easy...

The ‘joy’ of product status…

- Since 2020, DTrace based on BPF is supported as a product on various kernel releases:
  - 5.4.x-based kernels
  - 5.15.x-based kernels
- Most development is done on newer kernels:
  - 6.1.x
  - 6.5.x
  - bpf-next

… and that has consequences!

The joy of ‘product’ status… (cont.) … and other challenges

- Kernel helpers differ between kernel versions (usually more, never less)
- BPF verifier behaviour differs between kernel helpers
- Kernel implementation of target areas for tracing change (less common)

- And there is an expectation of retaining documented behaviour

Some BPF helpers are only available in newer kernels

- bpf_probe_read() and bpf_probe_read_str() can be used for kernel and userspace addresses on most architectures (but not all)
  - bpf_probe_read_kernel(), bpf_probe_read_user(), bpf_probe_read_kernel_str(), bpf_probe_read_user_str() were introduced later to resolve this
  - Some kernels versions had a confusing mix of what worked and what didn’t
- bpf_get_current_task_btf() and bpf_task_pt_regs() were introduced in later kernels
  - But we still need to get to task CPU registers on older kernels also
  - We wrote some (semi-convoluted) BPF code to mimic the kernel code to determine the location of the saved userspace registers for the current task, using bpf_probe_kernel_read()s to chase pointer chains to get to our target.

Creative programming to work around BPF verifier limitations

- BPF verifier is meant to guarantee safety of BPF programs being loaded into the kernel
- BPF verifier allows programs to pass that it deems safe
  - But it may reject programs that are actually perfectly safe
  - It is impossible to get it right all the time
- Compiler-generated code may be safe because of how the code is generated, but the BPF verifier may not be able to ascertain that
- DTrace provides programmable dynamic tracing, so it **needs** to be able to generate programs

- We strike a balance between generating efficient code and code that is constructed to ensure it can pass the BPF verifier.

Creative programming to work around BPF verifier limitations (cont.)

- Remember: BPF verifier implementations differ between kernel versions
- We need to be able to pass the BPF verifier on all supported kernel versions
  - Sadly, this is often a process of trial and error:
    - Analyze a rejection
    - Find a solution
    - Ensure the solution is valid on all supported kernel versions

Creative programming to work around BPF verifier limitations (cont.)

- The BPF verifier uses static evaluation of instruction sequences to ‘prove’ safety
- There is a limit on how many instructions the BPF verifier will evaluate: **1 million**
  - That is a pretty low limit, because...

Creative programming to work around BPF verifier limitations (cont.)

- The BPF verifier uses static evaluation of instruction sequences to ‘prove’ safety
- There is a limit on how many instructions the BPF verifier will evaluate: **1 million**
  - That is a pretty low limit, because…
    - The BPF verifier has limited state saving capabilities
    - Loops often need to evaluated for every possible input value
    - Code after a function call return may need to be evaluated for every possible return value
  - Adding in (pointless) branches can give the BPF verifier hints about value range boundaries
  - But we need to be careful – no dead code allowed!
    - This is also a challenge in view of program linking… resolving symbolic constants could render a code block dead code.

Creative programming to work around BPF verifier limitations (cont.)

- The BPF verifier’s static evaluation of instruction sequences is complex
  - Predicting how the BPF verifier will evaluate your code is extremely difficult
    - And can change depending on the kernel version
  - Understanding a failure is not always enough to figure out a solution

Tracing infrastructure performance with MANY probes

- Creating a large amount of kprobes (or uprobes) is pretty slow
- Removing a large amount of kprobes (or uprobes) is very slow
  - 51351 probes took 58.37s!
- Problem seems to be located in the management of data structures at the kernel level
  - Possible solution for removals: lazy removal
    - Mark probe for removal but don’t remove it from the list
    - At regular intervals, do a batch removal of “stale” probes

Features we need but do not have in upstream kernels (yet)

- DTrace probe naming is expected to be ‘stable’: provider:module:function:name
  - Probes in code that can be compiled as a kernel module are expected to be grouped under the module name
  - Whether the module is compiled as a loadable module or compiled into the kernel should not affect the probe naming
  - Patch submitted to upstream kernel: **kallmodsyms**
- DTrace makes extensive use of datatype information
  - Depending on debuginfo is unacceptable (too large)
  - CTF (Compact C Type Format) was developed for this in the early days of DTrace
  - Support is now in GCC and binutils
  - Patch to be submitted to upstream kernel: **CTF**

Features we need but do not have in upstream kernels yet (cont.)

- DTrace needs to be able to listen for various events using a poll interface
  - Notification of available data in perf output buffers
  - Notification of state changes of processes (pid)
  - Existing mechanisms are not adequate
  - Patch to be submitted to upstream kernel: **waitfd()**

More information… Get involved…

- Source code:
  - https://github.com/oracle/dtrace-utils (dev branch)
- Mailing list:
  - dtrace-devel@oss.oracle.com

Thank you!

ORACLE

Our mission is to help people see data in new ways, discover insights, unlock endless possibilities.