DTrace for Rust*

*and everything else too

Adam Leventhal, Oxide
@ahl@mastodon.social
@ahl.bsky.social

Level: Intermediate

Some of this will be
“Introductory and overview”

I’m sorry

What is DTrace?

- Dynamic tracing facility
- Built to explore problems ***in production*** on ***mission critical*** systems
- Systemic in scope: kernel, drivers, user-land, dynamic languages, etc.
- Developed at Sun for Solaris; first available in 2003
- Ported to FreeBSD, macOS, Linux, Windows (and others)

The values of DTrace

- The focus on production use requires certain fundamental values
- Safety – a diagnostic system should do no harm
- Availability – you want tools at the ready when you hit problems
- Zero disabled probe-effect – when not in use, DTrace should have no impact to system performance
- These should sound familiar to Rust users!
    - Memory safety; rigorous error handling
    - Production use
    - Zero-cost abstractions

Early DTrace

- DTrace started as a tool for **us** to answer questions about the system
- Added dynamic instrumentation for kernel functions
- No special compilation or wide-spread code changes required

Statically Defined Tracing (SDT)

- Dynamic probes let us see pathologies that had previously been **very** expensive to observe
- Required a lot of familiarity with the implementation
- Fragile release-to-release
- Introduced Statically Defined Tracing to statically mark points of semantic importance
- E.g. I/O, thread scheduling, process lifecycle, locking events

User-land tracing

- There’s a lot more user-land code than kernel code out there so we also wanted to see into process activity
- Added the “pid” provider that can instrument user-land functions
- Again, fully dynamic; no need for special compilation
- Works by replacing instructions with a trap
- The trap handler invokes DTrace’s machinery

User-land Statically Defined Tracing (USDT)

- As before, function tracing requires familiarity with the implementation
- Added USDT to embed probes in user-land programs
- Those probes are part of the compiled binary
- Register with the kernel when the process starts
- Great for exposing higher level semantics (e.g. `postgres:::transaction-start`)
- Turned out to let us inspect dynamic languages such as Java, Ruby, JavaScript, Python, Perl, PHP

DTrace for Rust

- Rust generates binaries that look very much like binaries from C or C++
- We can use the same DTrace facilities for dynamic instrumentation to look at them!

Let’s trace!

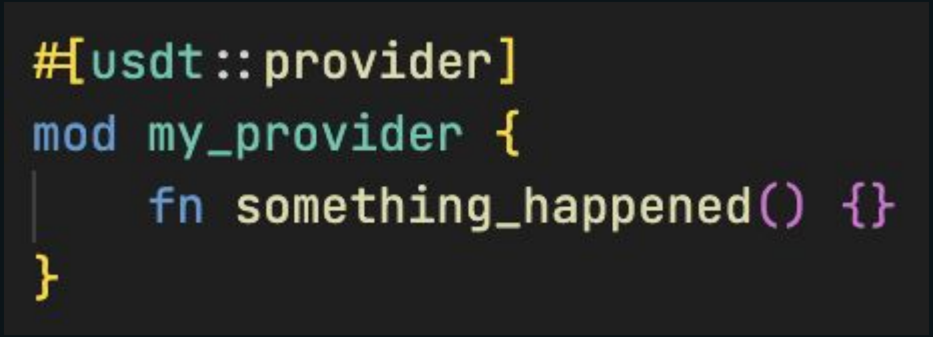
Adding USDT probes to Rust code

- Use the `usdt` crate

```
$ cargo add usdt
```

- Written by me and my Oxide colleague, Ben Naecker
- At Oxide we use this **everywhere**!
- Where to add probes?
  - Where you log
  - Points of interest
  - Before and after actions whose latency you might want to measure
  - …

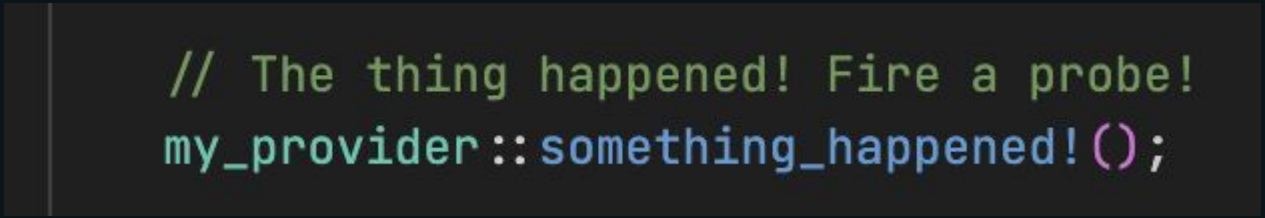
Define your provider



```rust
#[usdt::provider]
mod my_provider {
    fn something_happened() {}
}
```

Note: you'll generally want this at the crate root i.e. main.rs or lib.rs.

Invoke the probe in your code

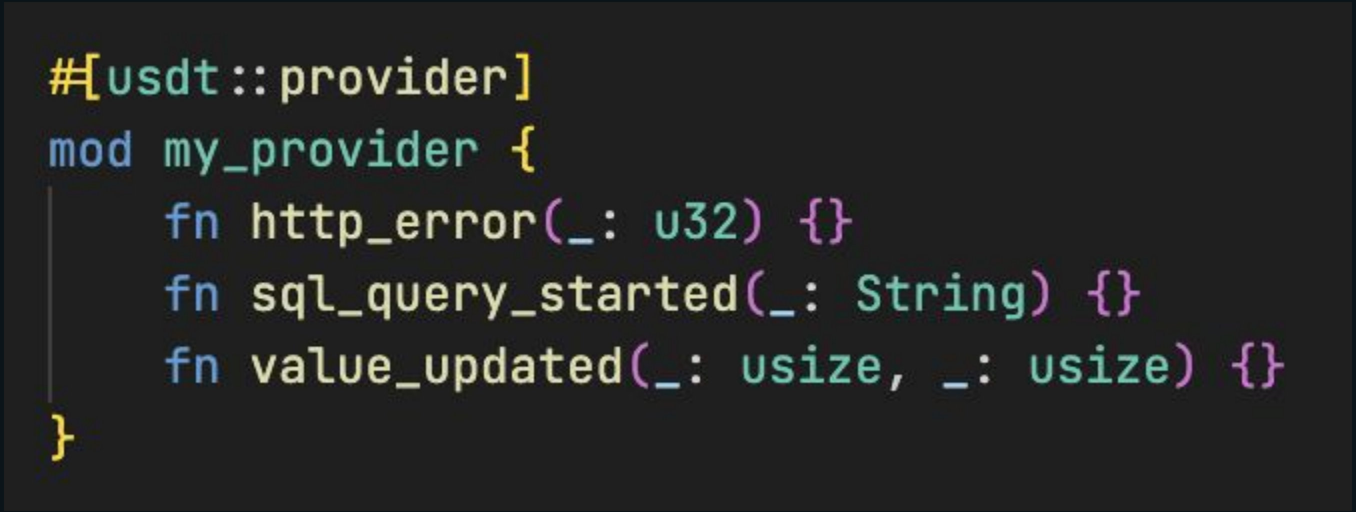


```
// The thing happened! Fire a probe!
my_provider::something_happened!();
```

Neat! But only starting to be useful…

ADVANCED CONTENT:
A macro just output another macro and no one even blinked

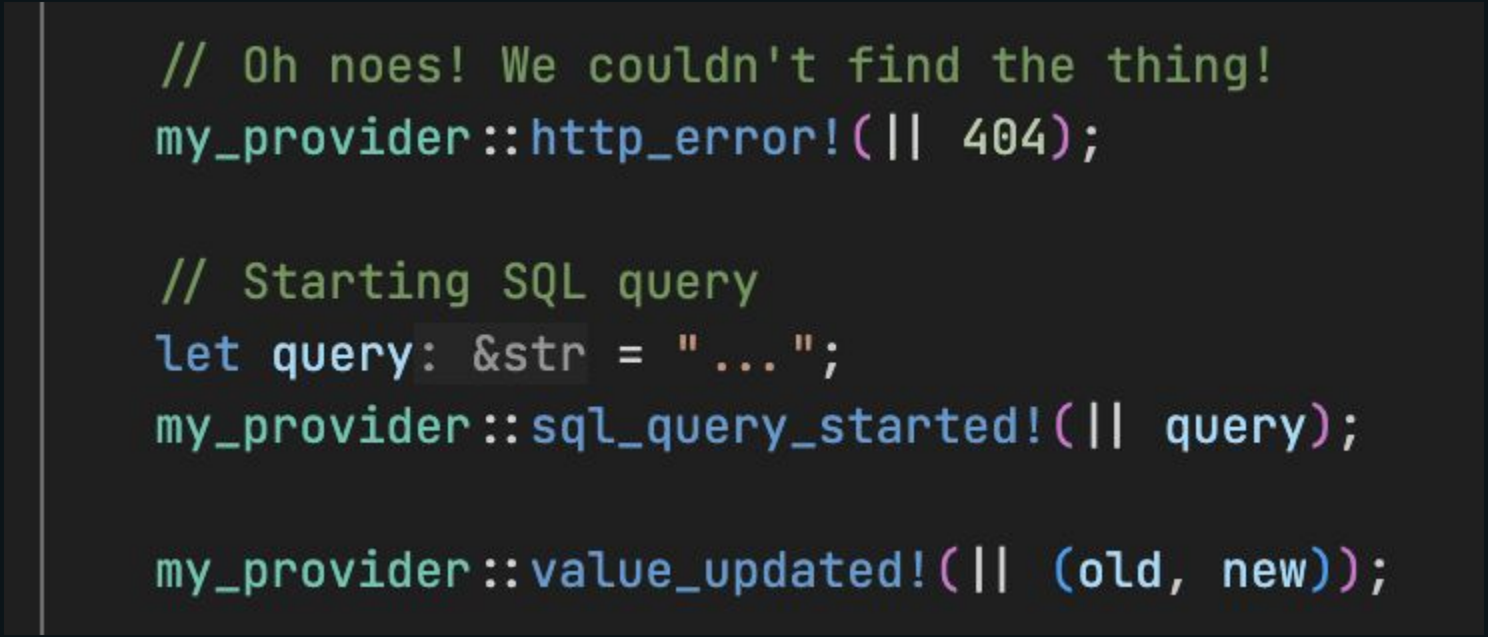
Adding arguments



```
#[usdt::provider]
mod my_provider {
    fn http_error(_: u32) {}
    fn sql_query_started(_: String) {}
    fn value_updated(_: usize, _: usize) {}
}
```

Integral and string types are straightforward

Probes with arguments


```rust
// Oh noes! We couldn't find the thing!
my_provider::http_error!(|| 404);

// Starting SQL query
let query: &str = "...";
my_provider::sql_query_started!(|| query);

my_provider::value_updated!(|| (old, new));
```

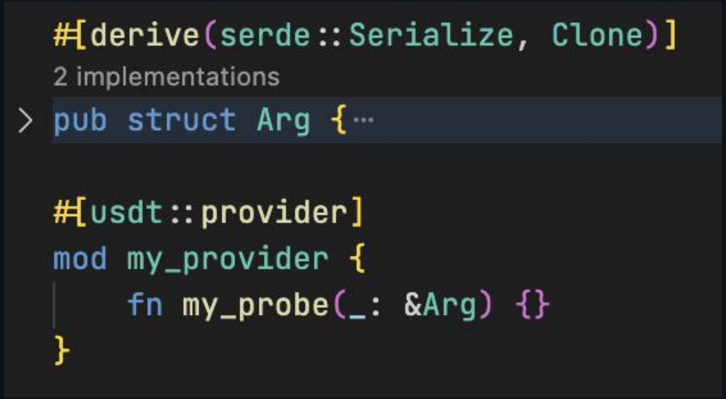
Note: invoke with a closure that returns the values to be traced.

More arguments

- Wait, why do probes take closures rather than the actual arguments?
- Good question! First, let’s look at arguments that aren’t just strings and numbers
- We can pass in value of any type that implements `serde::Serialize`

- Why `Serialize`? Hold that thought…

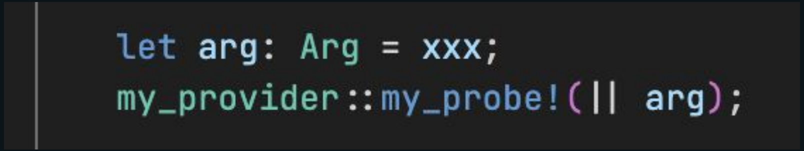
Probes with more complex arguments



```
#[derive(serde::Serialize, Clone)]
2 implementations
> pub struct Arg {…

#[usdt::provider]
mod my_provider {
    fn my_probe(_: &Arg) {}
}
```

Invoke the probe as before:



```
    let arg: Arg = xxx;
    my_provider::my_probe!(|| arg);
```

Consuming USDT probes in DTrace

- DTrace probes have arguments named arg0, arg1, arg2, …

```
# dtrace -n 'my_provider*:::http_error{ trace(arg0); }'
dtrace: description 'my_provider*:::http_error' matched 0 probes
CPU     ID                    FUNCTION:NAME
  1 521333 _ZN9test_usdt4main17h4f231f0987428b4bE:http_error        404
```

Strings are in userland so we need to copy them in

```
# cat sql.d
my_provider*:::sql_query_started
{
    trace(copyinstr(arg0));
}

# dtrace -s sql.d
dtrace: script 'sql.d' matched 0 probes
CPU     ID                    FUNCTION:NAME
  1   3106 _ZN9test_usdt4main17h4f231f0987428b4bE:sql_query_started
                   SELECT * FROM data
```

Tracing complex types

- Recall that more complex types need to impl serde::Serialize
- The probe invocation serializes the value to JSON
- DTrace's built-in `json()` function lets us navigate the serialized structure
- Serialization is fallible so there are top-level properties `ok` or `err`

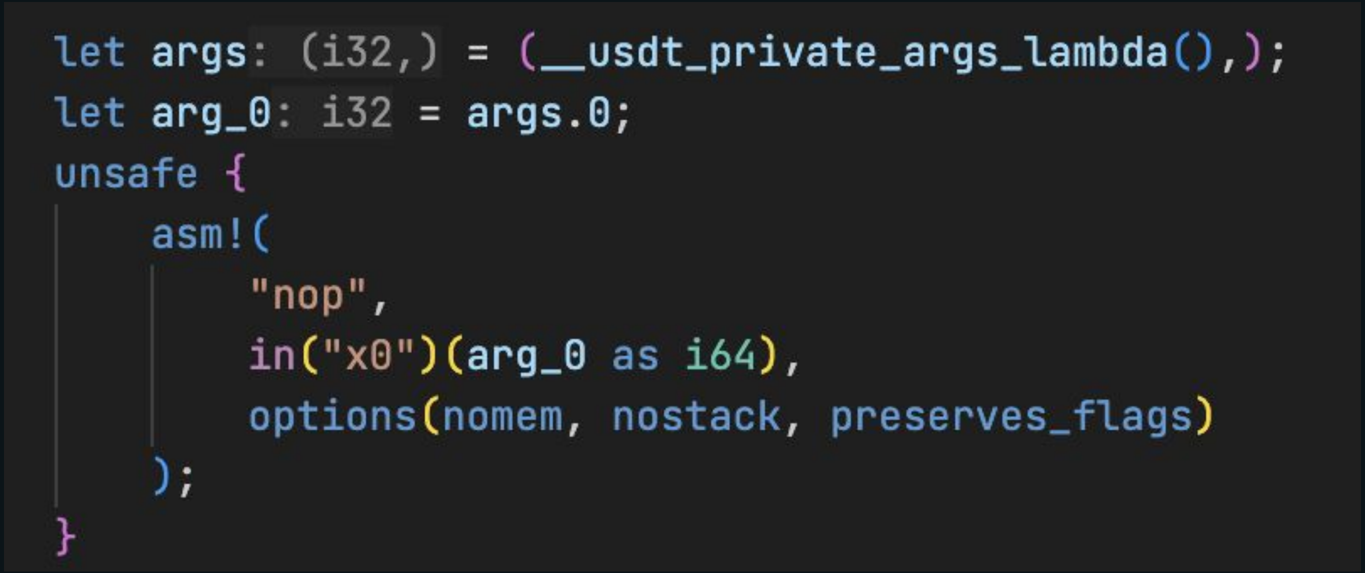
```
# cat complex.d
my_provider*:::my_probe
{
    trace(json(copyinstr(arg0), "ok.val"));
}
# dtrace -s complex.d
dtrace: script 'complex.d' matched 0 probes
CPU     ID                    FUNCTION:NAME
 13   4693 _ZN9test_usdt4main17h4f231f0987428b4bE:my_probe   7
```

Ew… gross.

- Hard otherwise to convey complex structure into DTrace
- JSON is, broadly, the interchange we’ve settled on (for good or ill)
- The probe macros (generated by the provider macro) encapsulate quite a bit of complexity
- Tricky to make these probes have zero disabled probe-effect (zero-cost abstractions)

ADVANCED CONTENT

Expanding a probe macro (simple)

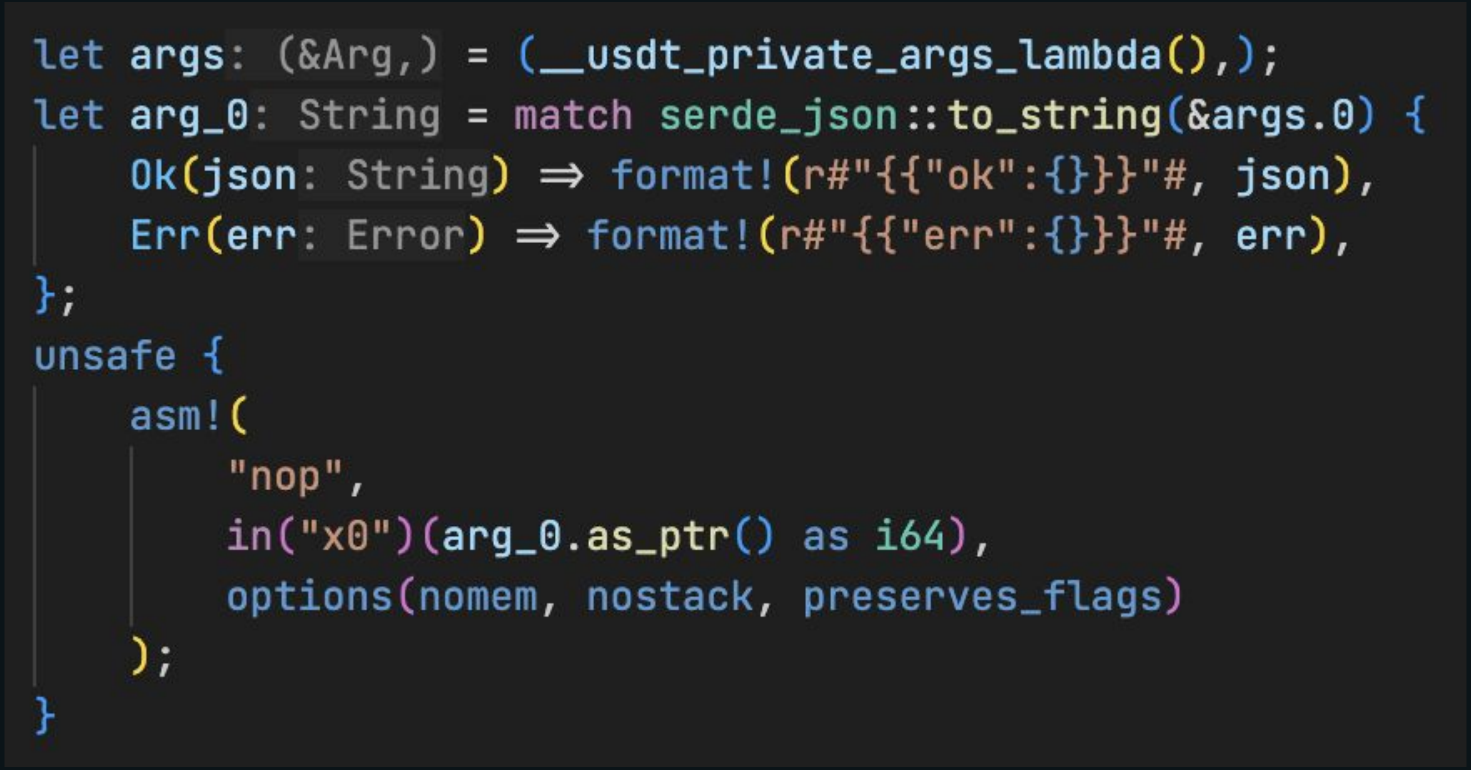


```
let args: (i32,) = (__usdt_private_args_lambda(),);
let arg_0: i32 = args.0;
unsafe {
    asm!(
        "nop",
        in("x0")(arg_0 as i64),
        options(nomem, nostack, preserves_flags)
    );
}
```

DTrace replaces the `nop` with a trap when enabled

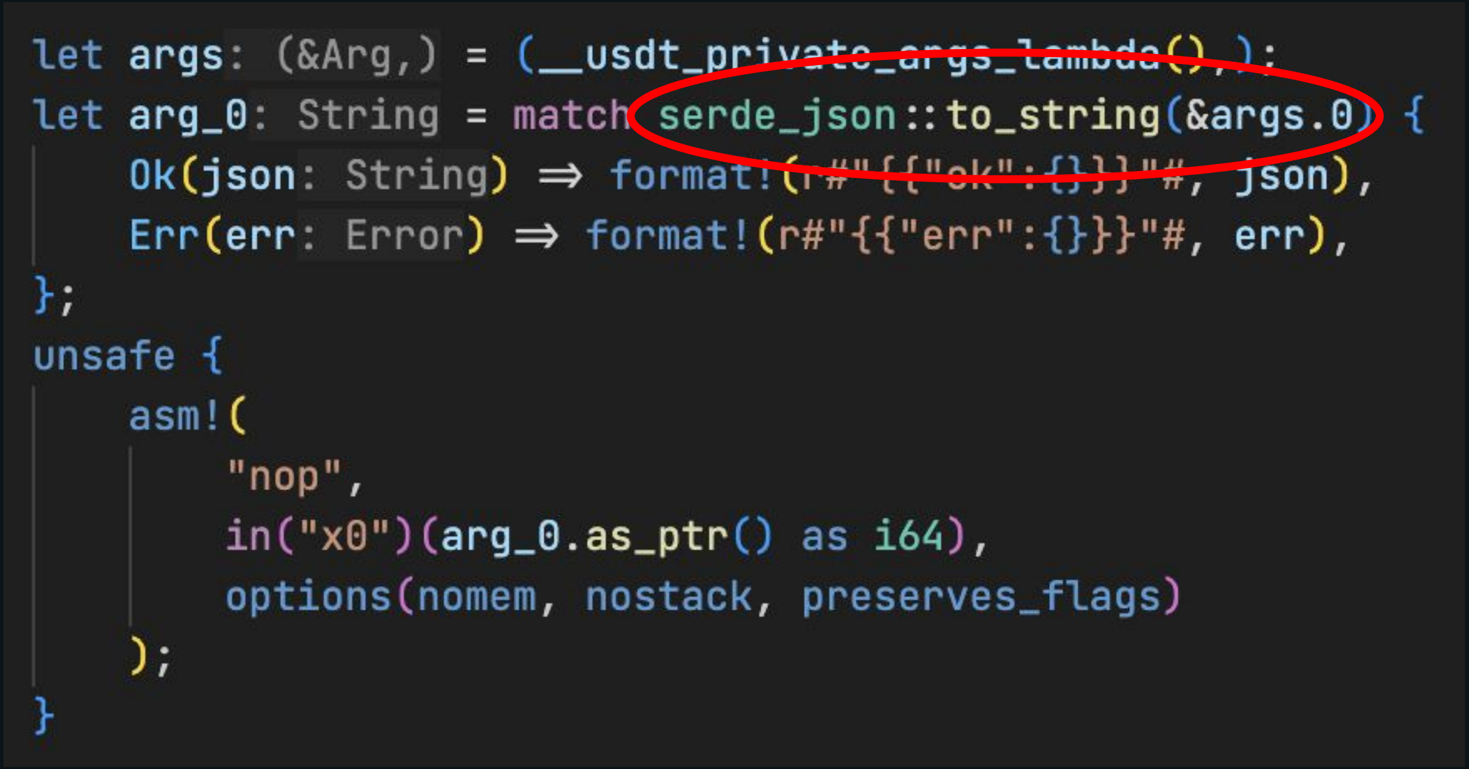
* Some non-code details elided

Expanding a probe macro (complex)



```rust
let args: (&Arg,) = (__usdt_private_args_lambda(),);
let arg_0: String = match serde_json::to_string(&args.0) {
    Ok(json: String) => format!(r#"{{"ok":{}}}"#, json),
    Err(err: Error) => format!(r#"{{"err":{}}}"#, err),
};
unsafe {
    asm!(
        "nop",
        in("x0")(arg_0.as_ptr() as i64),
        options(nomem, nostack, preserves_flags)
    );
}
```

Expanding a probe macro (complex)



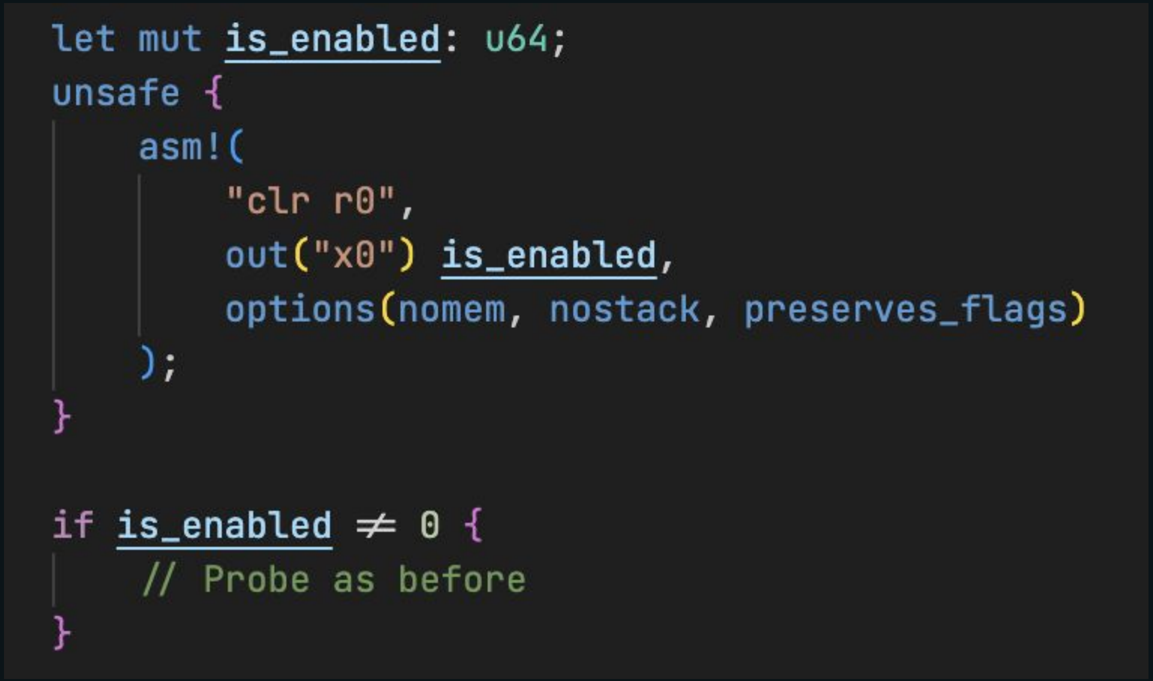
```rust
let args: (&Arg,) = (__usdt_private_args_lambda(),);
let arg_0: String = match serde_json::to_string(&args.0) {
    Ok(json: String) ⇒ format!(r#"{{"ok":{}}}"#, json),
    Err(err: Error) ⇒ format!(r#"{{"err":{}}}"#, err),
};
unsafe {
    asm!(
        "nop",
        in("x0")(arg_0.as_ptr() as i64),
        options(nomem, nostack, preserves_flags)
    );
}
```

Is-enabled probes

- Back in time, back in C, we had a similar problem
- What if the arguments to probes were expensive to compute?
- We came up with a new kind of dynamic instrumentation
- Rather than trap into the kernel, change the flow of code

```
if my_probe_is_enabled() {
    // calculate expensive arguments
    fire_my_probe(expensive arguments)
}
```

In Rust



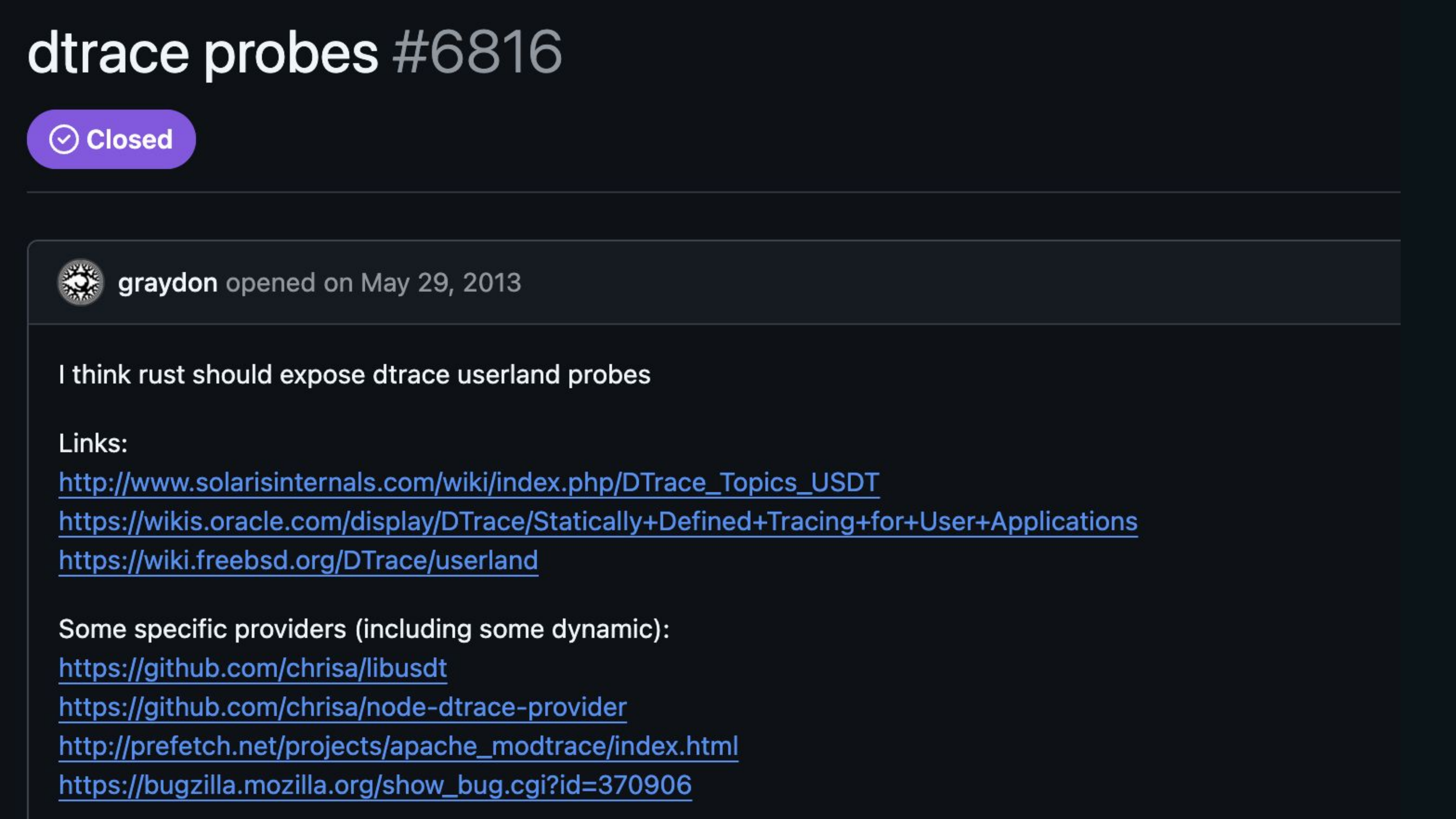
```rust
let mut is_enabled: u64;
unsafe {
    asm!(
        "clr r0",
        out("x0") is_enabled,
        options(nomem, nostack, preserves_flags)
    );
}

if is_enabled ≠ 0 {
    // Probe as before
}
```

This time, DTrace replaces the `clr` with a instruction that sets the register to 1

Good news!

- Fortunately, we don’t have to worry about that!
- That’s the power of zero-cost abstractions
- Probe arguments are closures to remind us that they might not be invoked!
- Rust lets us encapsulate that complexity…
- (at least, mostly …)
- … and add probes liberally



dtrace probes #6816

Closed

graydon opened on May 29, 2013

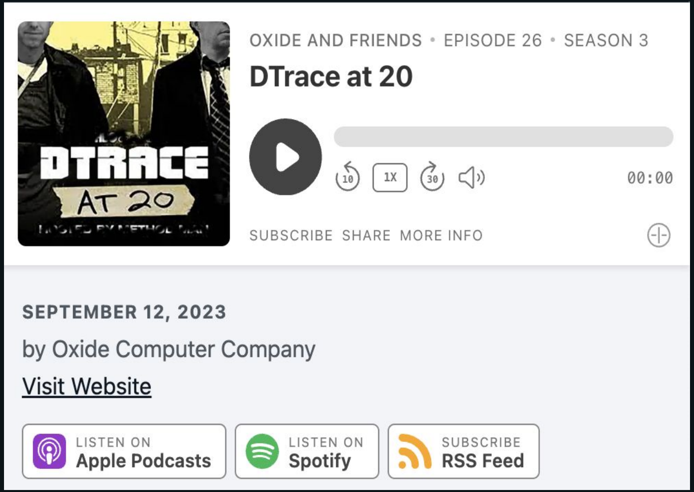
I think rust should expose dtrace userland probes

Links:
http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_USDT
https://wikis.oracle.com/display/DTrace/Statically+Defined+Tracing+for+User+Applications
https://wiki.freebsd.org/DTrace/userland

Some specific providers (including some dynamic):
https://github.com/chrisa/libusdt
https://github.com/chrisa/node-dtrace-provider
http://prefetch.net/projects/apache_modtrace/index.html
https://bugzilla.mozilla.org/show_bug.cgi?id=370906

Go forth, and DTrace

- Running on a system with DTrace?
- Think about adding probes with the usdt crate
- Next time you want to inspect a system at runtime, think about dynamic tracing rather than kill/println!/build/deploy


OXIDE AND FRIENDS • EPISODE 26 • SEASON 3

DTrace at 20

00:00

SUBSCRIBE SHARE MORE INFO

SEPTEMBER 12, 2023

by Oxide Computer Company

Visit Website

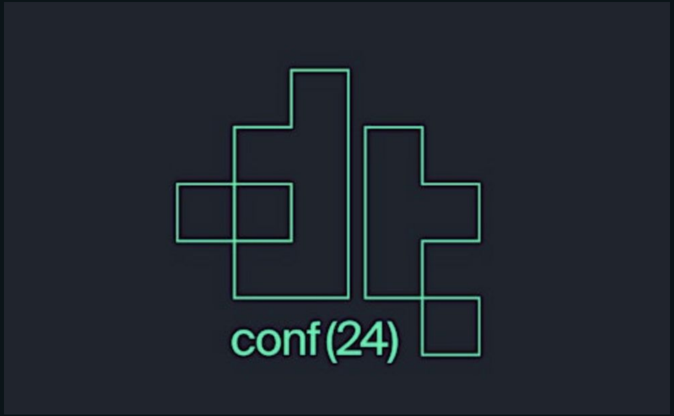
LISTEN ON Apple Podcasts | LISTEN ON Spotify | SUBSCRIBE RSS Feed

YouTube

P99 CONF 2024 | DTrace at 21: Reflections on Fully-grown Software by Bryan Cantrill

USDT for Rust

OxCon 2023
Benjamin Naecker
Adam Leventhal

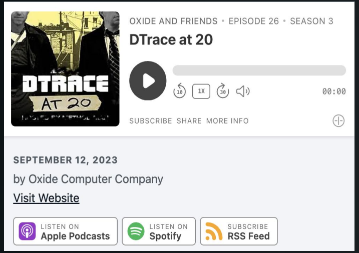


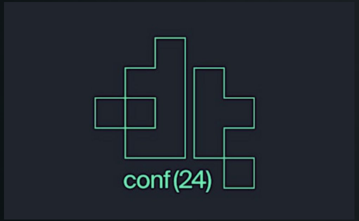
Thanks!

Adam Leventhal, Oxide
@ahl@mastodon.social
@ahl.bsky.social

0x

YouTube
P99 CONF 2024 | DTrace at 21: Reflections on Fully-grown Software by Bryan Cantrill


OXIDE AND FRIENDS • EPISODE 26 • SEASON 3
DTrace at 20
00:00
SUBSCRIBE SHARE MORE INFO
SEPTEMBER 12, 2023
by Oxide Computer Company
Visit Website
LISTEN ON Apple Podcasts
LISTEN ON Spotify
SUBSCRIBE RSS Feed



USDT for Rust
OxCon 2023
Benjamin Naecker
Adam Leventhal