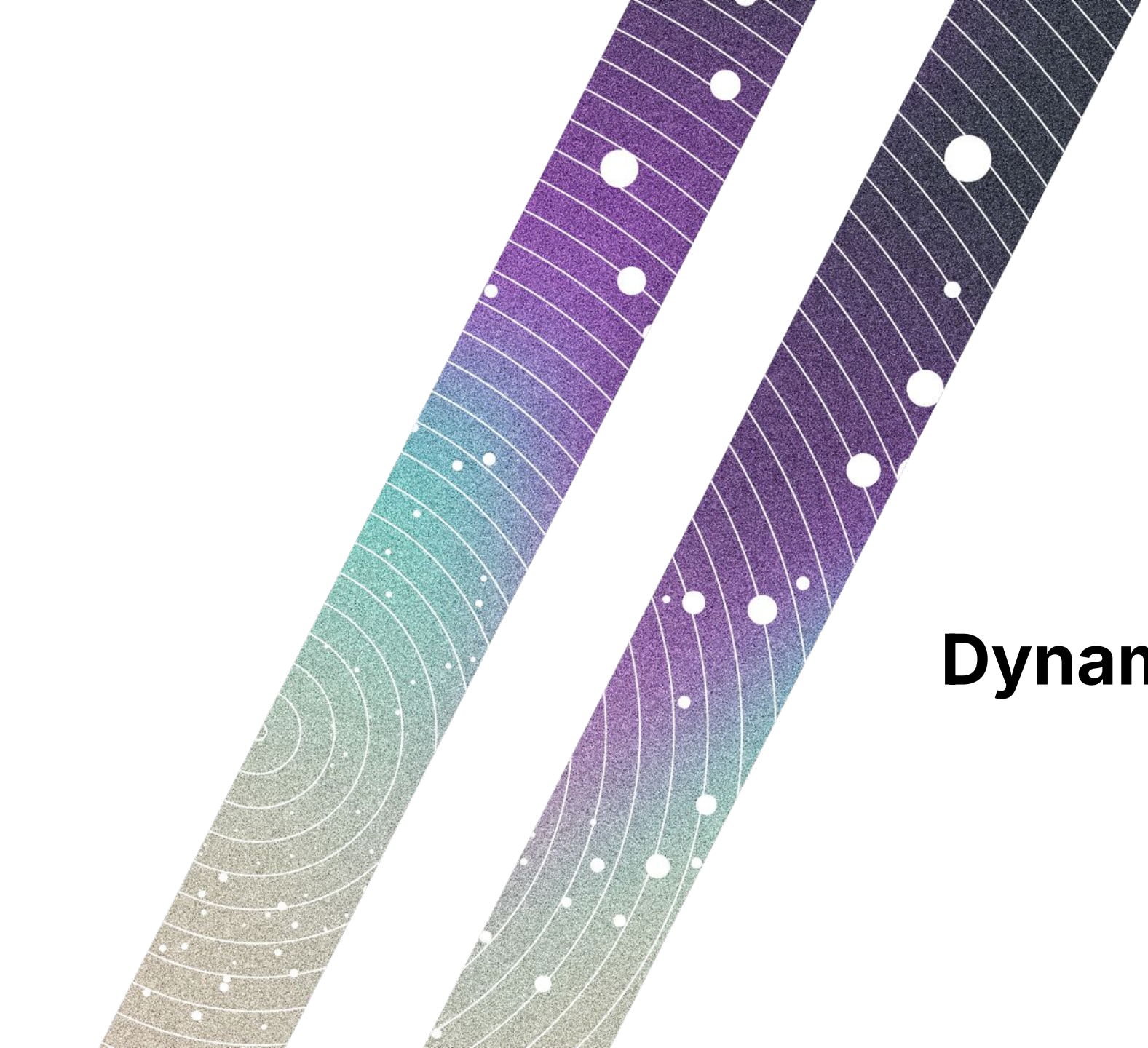


Pyda: Dynamic Binary Analysis Tools in Python

Andrew Haberlandt

whoami

- I play CTF (currently: MMM), mostly pwn/rev
- Currently: Auditor at OtterSec

Discord: ath0

Twitter: @_ndrewh

CTF writeups: https://fastb.in/

Ask me about…

- Symbolic execution
- SMT/SAT solvers
- My favorite undefined behavior



RE//verse

whoami

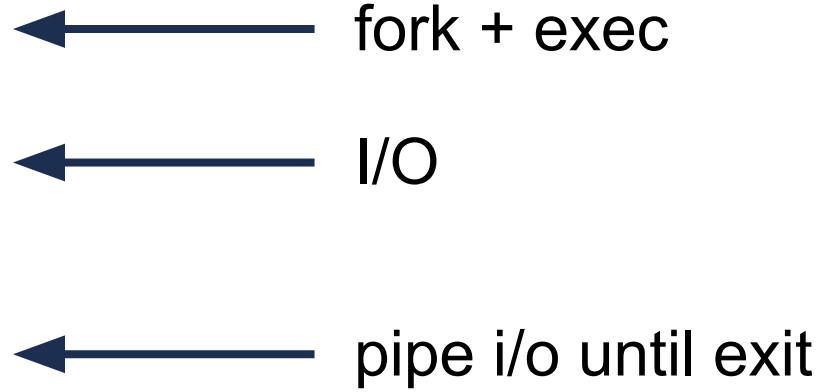
But for the sake of this talk…

I’m just a CTF player.



pwntools

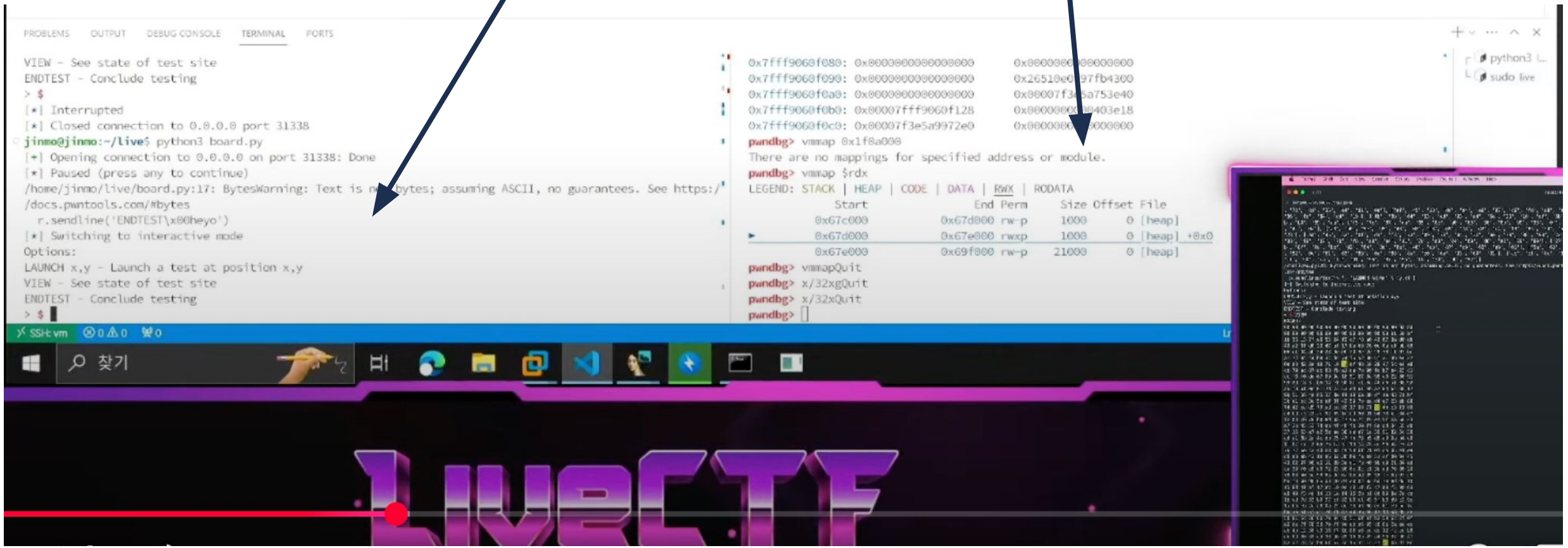
```python
from pwn import *

p = process("./password_checker")

p.recvuntil(b"Enter password:")
p.sendline(b"pass")

p.interactive()
```



Mostly black-box!

python + pwntools

gdb + pwndbg



https://www.youtube.com/watch?v=keowTHP8XRE

:(

```
from pwn import *

p = gdb.debug("./program", gdbscript="""
b *0x1337
c
""")

p.sendline(b"abcd")
p.interactive()
```

How *should* we interact with programs?

Debugger
(GDB)

```
break *0x1337
commands
    p/x $rax
    continue
end
```

```
# gdb -x my_script ./program
```

Hook-based dynamic
instrumentation
(Frida)

```
const F = Process.mainModule;
Interceptor.attach(F.base.add(ptr('0x1337')), {
  onEnter(args) {
    console.log(`RAX = ${this.context.rax}`)
  }
});
```

```
# frida -l script.js ./program
```



Wish list:

- ☐ Concise
- ☐ I/O
- ☐ Reuse pkgs
- ☐ Fast?

How *should* we interact with programs?

“Blocking”

pwntools

```python
from pwn import *

p = process("./password_checker")

p.recvuntil(b"Enter password:")
p.sendline(b"pass")

p.interactive()
```

GDB

```
b *0x1337
c
ni
b *0x1234
```

blocking

“Event-based”

Frida

```
const F = Process.mainModule;
Interceptor.attach(F.base.add(ptr('0x1337')), {
  onEnter(args) {
    console.log(`RAX = ${this.context.rax}`)
  }
});
```

RE//verse

why not both?

Event-based API

```
from pyda import *

p = process()
base = p.maps[p.exe_path].base

def hook(p):
    p.regs.rax += 1

p.hook(base + 0x1337, hook)
p.run()
```

```
# pyda script.py -- ./program
```

← ‘handle’ to the current process

← Read/modify regs

← Hook any instruction

Debugger-style API

```python
from pyda import *

p = process()
base = p.maps[p.exe_path].base

p.recvuntil("Enter password:")

# ... stopped here! ...

p.sendline("hunter1")
p.run_until(base + 0x1337)

# ... stopped here! ...

p.run()
```

process() returns a pwntools tube!

pwntools-style I/O

run_until resumes process until pc reached

You can modify regs/mem while stopped!

run() runs until completion

Interoperable w/ Event API!

Where should our “instrumentation” run?

Out-of-process

Examples: GDB, bpftrace, libdebug

+ Debug “anything with a GDB stub”
- Instrumentation limited: set breakpoints, handle interrupts
- Performance limited by interrupt handling + context switching overhead

In-process

Examples: Frida, Intel PIN, DynamoRIO

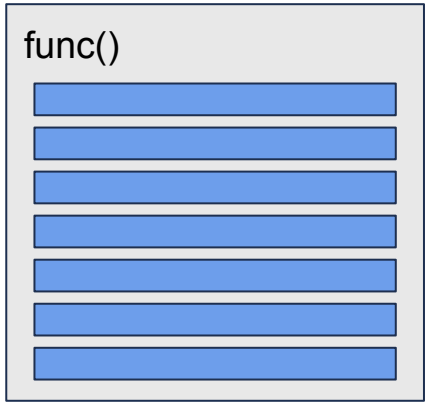
+ Instrumentation can directly write to memory w/o ptrace
+ Application can jump directly to instrumentation
- Application and instrumentation must share OS resources and (sometimes) shared libraries



What is a “dynamic instrumentation framework”?

Provides an interface to ***observe*** and ***modify*** the behavior of a program ***while it is executing***.

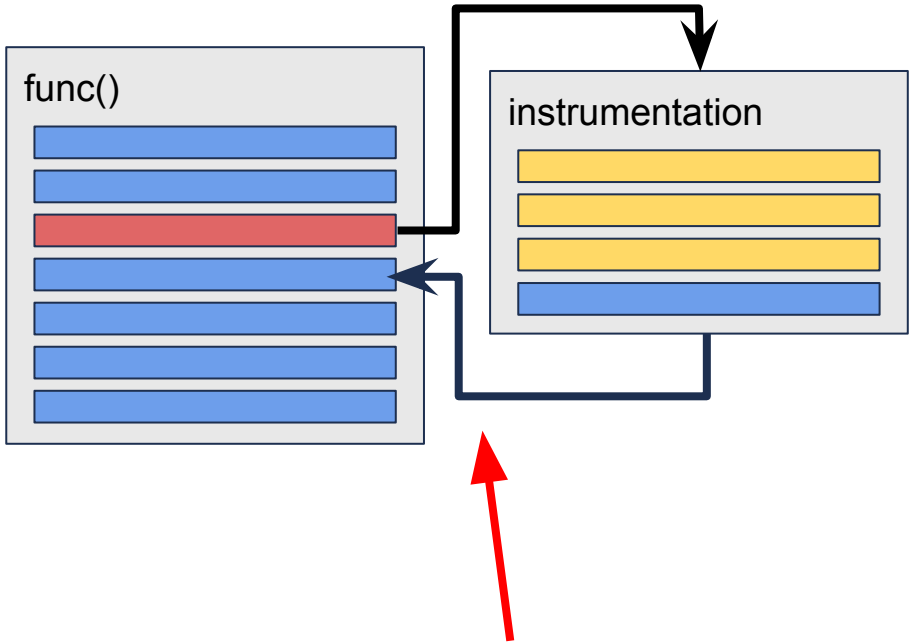
“Probe-based”



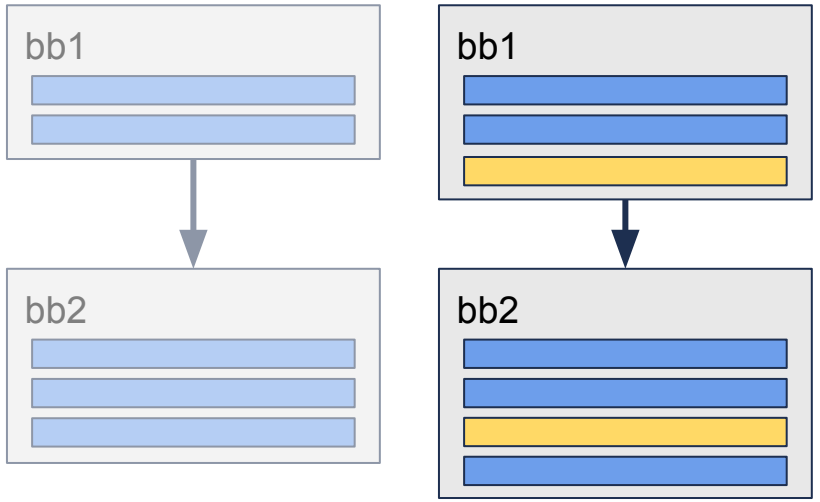
What is a “dynamic instrumentation framework”?

Provides an interface to ***observe*** and ***modify*** the behavior of a program ***while it is executing***.

“Probe-based”

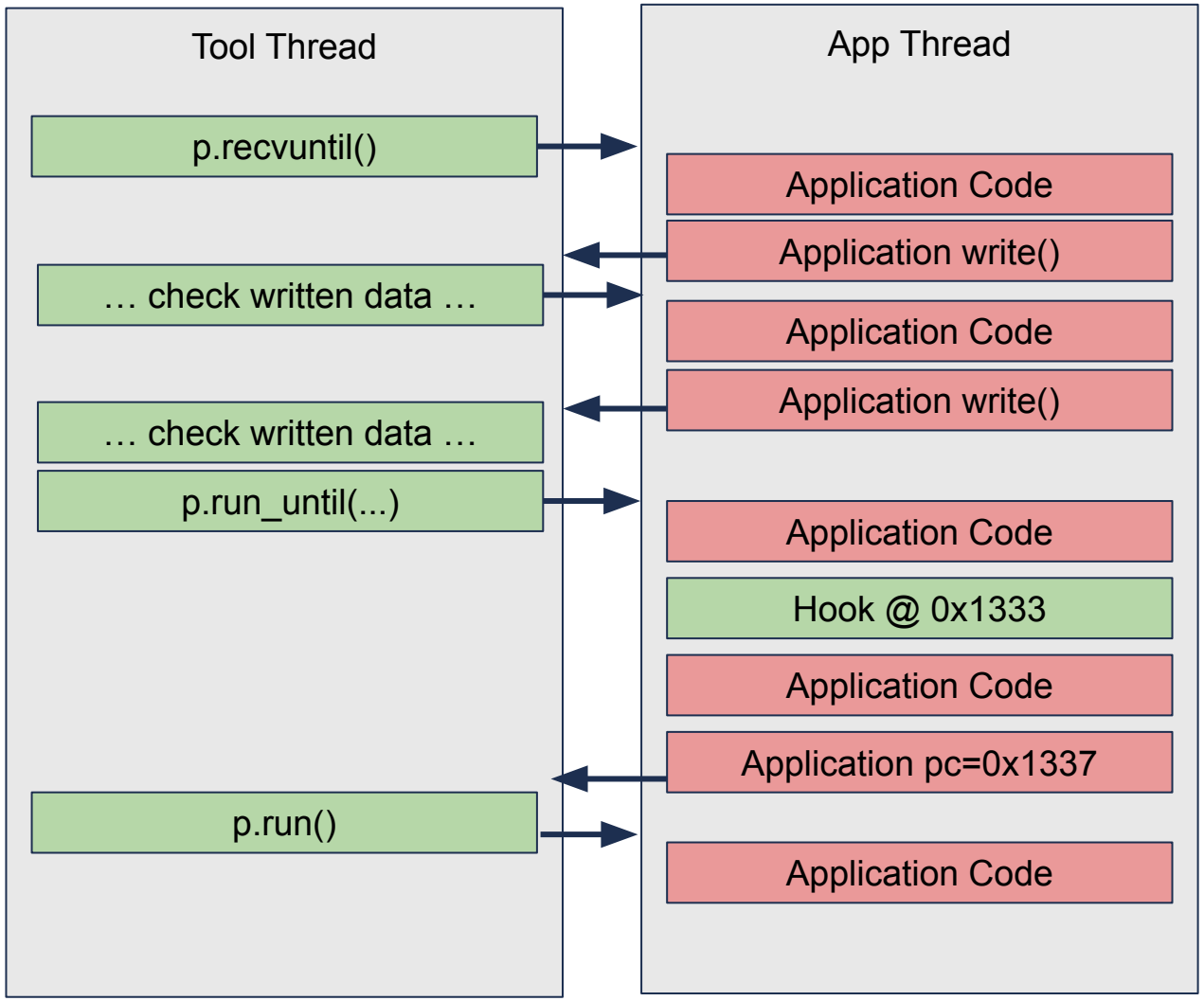


Dynamic recompilation



Embedding CPython inside a DynamoRIO tool

```python
from pyda import *

p = process()
base = p.maps[p.exe_path].base

p.recvuntil("Enter password:")

# ... stopped here! ...

p.sendline("hunter1")
p.run_until(base + 0x1337)

# ... stopped here! ...

p.run()
```



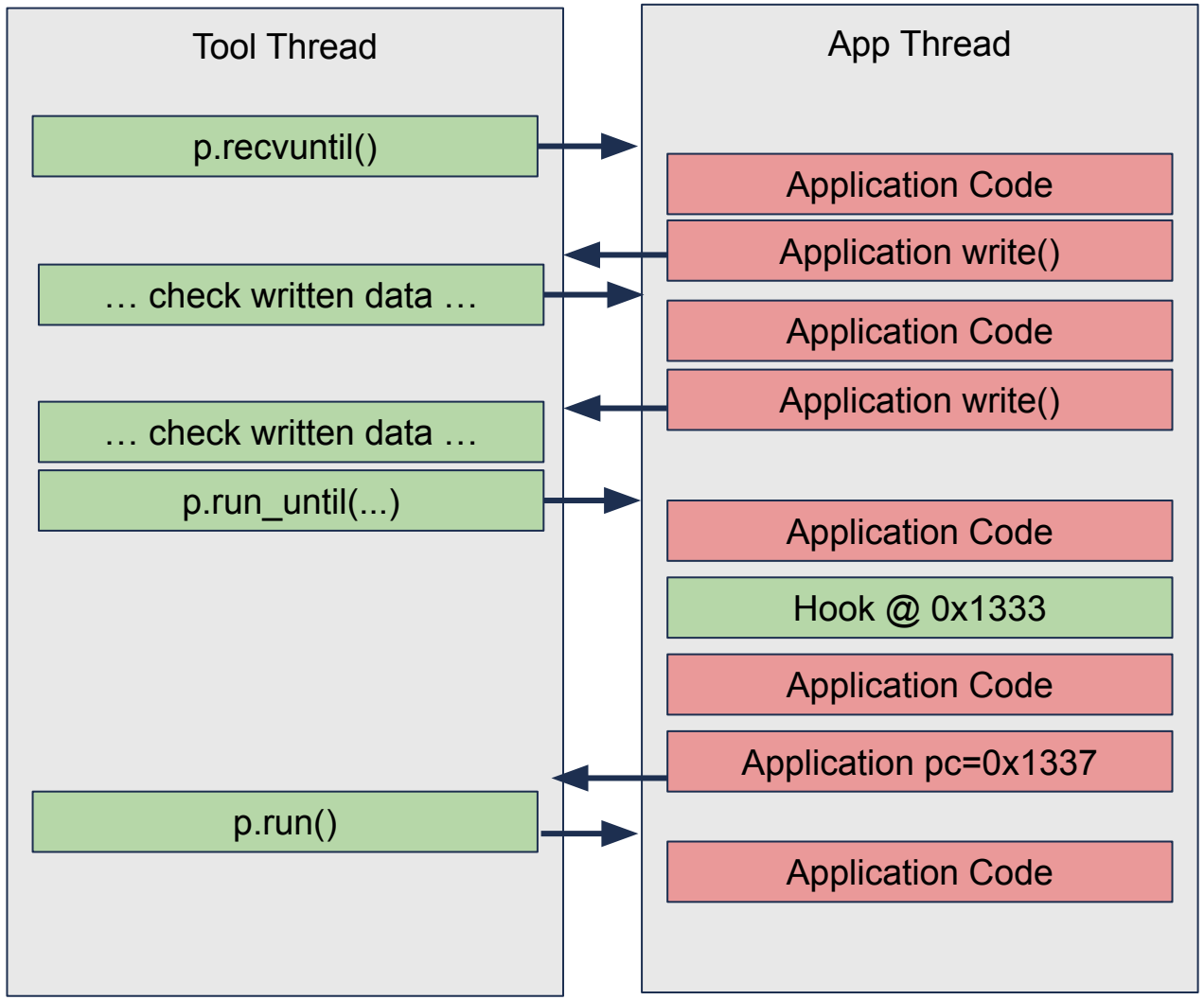
RE//verse

Embedding CPython inside a DynamoRIO tool

At every exit from Python (in either thread), we allow you to:

- Modify registers
- Add/Remove hooks (hooks are thread-global!)
- Redirect execution (i.e. modify p.regs.rip)

We also handle: thread creation/exit, fork, (fatal) signals, I/O capture



Example: `ltrace` in ~20 lines

Use pwntools ELF parser →

Enumerate all imports →

```
% pyda ltrace.py -- /bin/ls
strrchr(rdi=0x7ffcc2ad2db0, rsi=0x2f, rdx=0x7ffcc2ad10d8, rcx=0x7fc472
setlocale(rdi=0x6, rsi=0x7fc472aaebbd, rdx=0x80, rcx=0x80)
bindtextdomain(rdi=0x7fc472aae9a6, rsi=0x7fc472aaec79, rdx=0x1, rcx=0x
textdomain(rdi=0x7fc472aae9a6, rsi=0x7fc47602b590, rdx=0x1, rcx=0x0)
__cxa_atexit(rdi=0x7fc472aa0a80, rsi=0x0, rdx=0x7fc472ab7008, rcx=0x0)
isatty(rdi=0x1, rsi=0x7fc476065f00, rdx=0x20, rcx=0x0)
getenv(rdi=0x7fc472aaec8b, rsi=0x5401, rdx=0x0, rcx=0x7fc432ac1f42)
getenv(rdi=0x7fc472aaec99, rsi=0x5401, rdx=0x0, rcx=0x7fc432ac1f42)
ioctl(rdi=0x1, rsi=0x5413, rdx=0x7ffcc2ad0f60, rcx=0x7fc432ac1f42)
getenv(rdi=0x7fc472aaeca1, rsi=0x5413, rdx=0x7ffcc2ad0f60, rcx=0x7fc43
getopt_long(rdi=0x1, rsi=0x7ffcc2ad10c8, rdx=0x7fc472ab1380, rcx=0x7fc
...
```

```python
from pyda import *
from pwnlib.elf.elf import ELF

p = process()

e = ELF(p.exe_path)
e.address = p.maps[p.exe_path].base

plt_map = { e.plt[x]: x for x in e.plt }

def lib_hook(p):
    name = plt_map[p.regs.pc]
    print(f"{name}(" + ", ".join([
        f"rdi={hex(p.regs.arg1)}",
        f"rsi={hex(p.regs.arg2)}",
        f"rdx={hex(p.regs.arg3)}",
        f"rcx={hex(p.regs.arg4)}",
    ]) + ")")

for x in e.plt:
    p.hook(e.plt[x], lib_hook)

p.run()
```

Example: Record indirect jump targets

No need to learn new APIs

```
p = process()
BASE = p.maps[p.exe_path].base

objdump = subprocess.run(
  f"objdump -M intel -d {proc.exe_path} | grep call",
  shell=True,
  capture_output=True
)

#... a bunch of string parsing...
call_locs = parse_calls(objdump.stdout)
call_map = defaultdict(set)
def call_reg(p):
    src = p.regs.rip - BASE
    reg = call_locs[src]
    tgt = p.regs[reg] - BASE

    call_map[src].add(tgt)

for c in call_locs:
    p.hook(BASE + c, call_reg)

p.run()
print(dict(call_map))
```

```
targets = {
  46808: {44864},
  46938: {44864},
  46658: {44864}
}
for (s, dests) in targets.items():
    for d in dests:
        func = bv.get_functions_containing(s)[0]
        func.add_user_code_ref(s, d)
```

Example: Examine tcache state on ‘free’

Use (some) pwndbg APIs directly*



*we emulate some gdb apis used by pwndbg

```
from pyda import *
from pwnlib.elf import ELF
import pwndbg

p = process()

e = ELF(p.exe_path)
e.address = p.maps[p.exe_path].base

def hook(p):
    tcache = pwndbg.heap.current.tcachebins()
    if tcache is not None:
        for (s, b) in tcache.bins.items():
            if len(b.fd_chain) < 2:
```

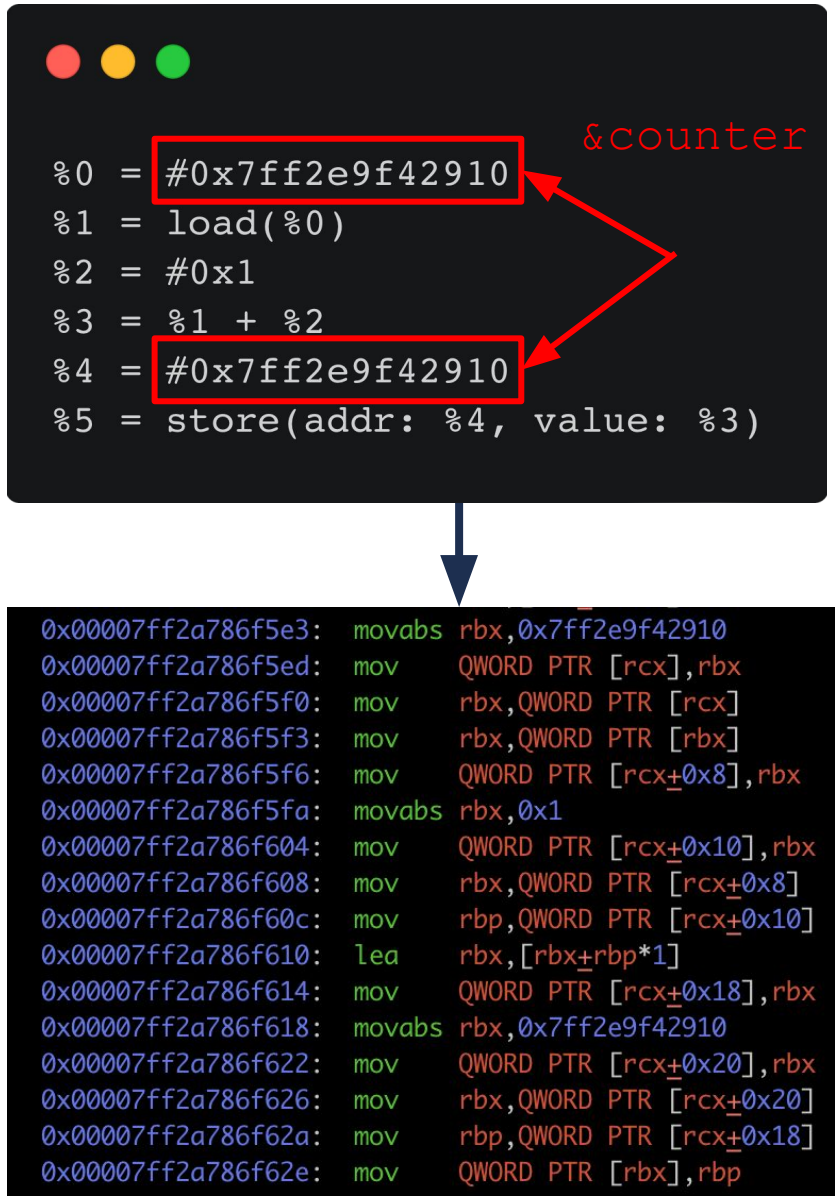
```
tcache 0x20: 0x7fd6eb6df300 -> 0x7fd6eb6df2e0 -> 0x7fd6eb6df2c0 -> 0x7fd6eb6df2a0 -> 0x0
tcache 0x30: 0x7fd6eb6df320 -> 0x0
```

```
p.run()
```

Compiler for inlining instrumentation

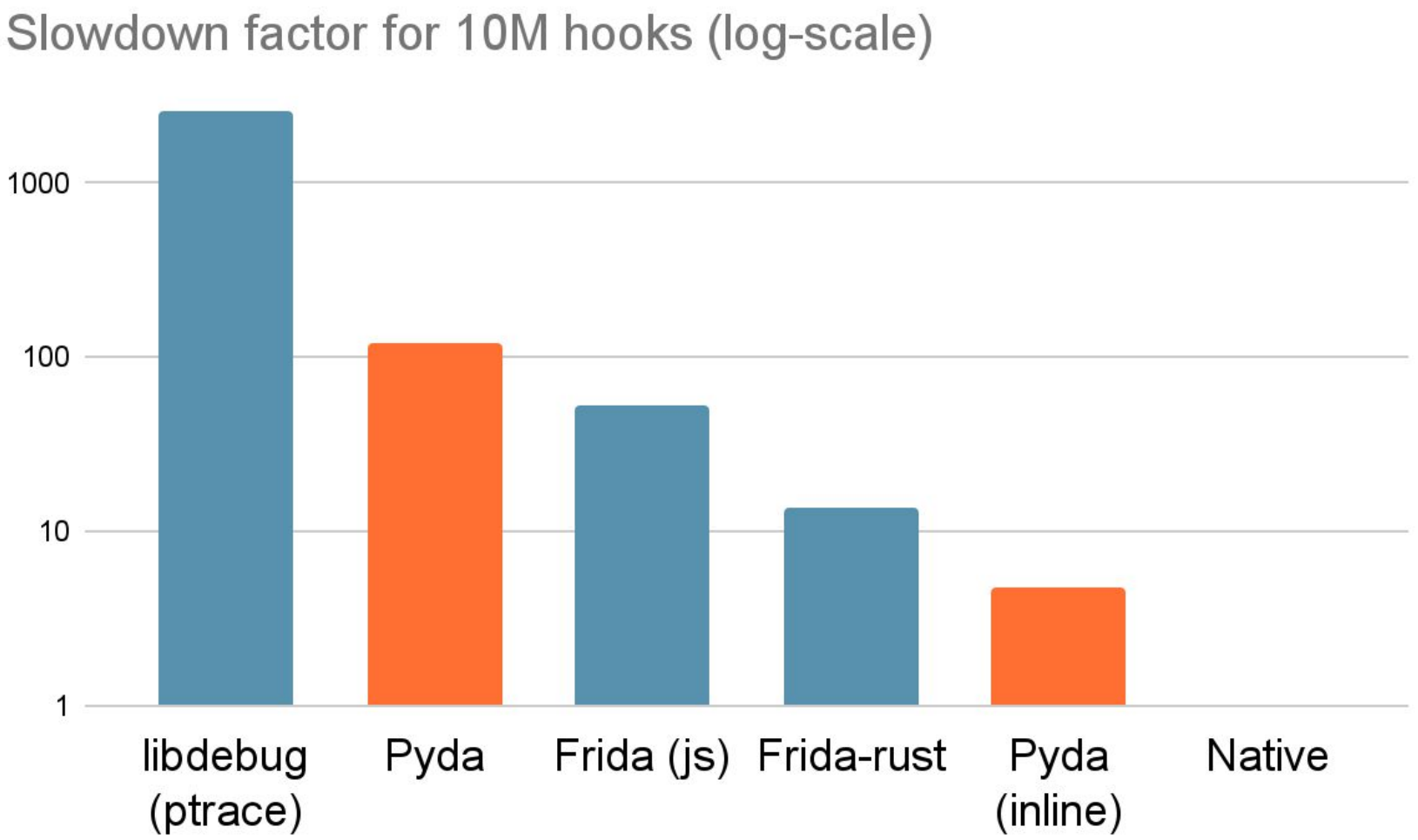
Called only once!

```python
from pyda import *

p = process()
base = p.maps[p.exe_path].base

counter = Global(ctypes.c_long(0))

def lib_hook(b):
    counter.val += 1

p.builder_hook(base + 0x1056, lib_hook)
p.run()
print(counter)
```



RE//verse

Inline hooks are ~10x faster



Inlining complex instrumentation: ringbuffer

```python
from pyda import *

p = process()
base = p.maps[p.exe_path].base

counter = Global(ctypes.c_long(0))
buf = Global((ctypes.c_long * 1000)())

def log_tmp(b):
    counter.val += 1
    buf[counter % 1000] = b.regs.rdi

p.builder_hook(e.address + 0x1056, lib_hook)
p.run()

print(buf.cval)
```



Global also supports ctypes arrays



Mod operator enables ringbuffer accesses

Demo



Summary

- Hooks inspect/modify registers and memory at any instruction
- Full dynamic instrumentation w/ DynamoRIO + custom compiler = faster instrumentation
- Currently supports X86/ARM64 Linux (WIP: macOS)

```python
from pyda import *

p = process()
base = p.maps[p.exe_path].base

def hook(p):
    p.regs.rax = 0

def sys_hook(p, sys_num):
    print(hex(p.regs.rip))

p.hook(base + 0x1337, hook)
p.syscall_pre(0, sys_hook)
p.set_thread_entry(lambda p: print(f"tid {p.tid} started")
p.run()
```

https://github.com/ndrewh/pyda

RE//verse

Github

Slides