libdebug

A Python library to automate the debugging of binary executables

19 April 2024

POLITECNICO
MILANO 1863

Development Team

@JinBlack

@Frank01001

@MrIndeciso

@lo_no

@MarcoDige

Feel free to ask any of us for help.

Additional developers are welcome too.

Why

- GDB is complex, slow, and can break easily (e.g., corrupted ELF)
- Many rev challenges require scripted debugging
- Nobody really knows gdbscript
- Python is cool and everyone knows and uses Python

How it works

- libdebug uses the same backend as GDB: ptrace (which is ugly, unreliable, undocumented, obscure, nonsensical, and many other things)
- libdebug provides a small set of commands (continue, breakpoint, step, backtrace…) which behave mostly like in GDB
- libdebug automatically handles communication with the process over stdin/stdout, just like pwntools
- Additionally, libdebug provides advanced features such as asynchronous breakpoints, syscall hooking and more

libdebug for dummies

A simple how-to guide to scripted debugging

Installation

```
python3 -m pip install git+https://github.com/libdebug/libdebug.git
```

We are still waiting for the organization to be approved on PyPI…

Installation requirements:

- `libdwarf-dev, libelf-dev`
- `libiberty-dev`
- `linux-headers-generic`
- `python3-dev`

Your first script

```
from libdebug import debugger


d = debugger("/bin/ls")
d.run()
print(hex(d.rip))
d.kill()
```

```
from libdebug import debugger


d = debugger()
d.attach(pid=12345)
print(hex(d.rip))
d.kill()
```


Register Access

```
val = d.rax
d.rax += 1
d.bh = 3
d.rcx = d.ebp
```

Anything you can do in assembly, you can do in the script

Register Access

```
val = d.rax
d.rax += 1
d.bh = 3
d.rcx = d.ebp
```

Anything you can do in assembly, you can do in the script

What about floating point, AVX and all of that?

```
print(hex(d.ymm0))
```

Still in development, **coming shortly** to main branch

Setting a Breakpoint (part 1)

```
bp = d.breakpoint(
    position: <int | str>,
    hardware: <True | False> = False,
    condition: <str | None> = None,
    length: <int> = 1,
    callback: <None | Callable> = None
)

bp = d.breakpoint(0x374)
bp = d.breakpoint("main")
bp = d.breakpoint("main+2a")
bp = d.breakpoint(0x738, hardware=True)
```


Memory Access

```
d.memory[d.rsp]                                  1 byte
d.memory[0x200 : 0x400]                          0x200 bytes
d.memory["main+a1"]                              1 byte
d.memory["main" : "main+8"]                      8 bytes
d.memory["main", 29]                             29 bytes


d.memory["main_arena"] = b"0123456789"           10 bytes
```


Process Communication

```
r = d.run()
d.cont()

r.recvuntil(b"password: ")
r.sendline(b"ionoscarso")
r.sendafter(b"flag: ", b"replyctf{the_earth_is_flat}")

print(r.recvline())
```

run() returns a pipe to the process which works just like pwntools.

Validating your script with GDB

If you think the script is behaving incorrectly you can always double-check it from GDB.

```
d.migrate_to_gdb()
```

or

```
from libdebug import libcontext

libcontext.terminal = ["tmux", "splitw", "-h"]

d.migrate_to_gdb()
```

When you quit GDB the script will go back to running.

Demo Time

Setting a Breakpoint (part 2: callbacks)

```
bp = d.breakpoint(
    position: <int | str>,
    hardware: <True | False> = False,
    [...],
    callback: <None | Callable> = None
)

def callback(t, bp):
    print(hex(t.rip))
    print(t.memory[d.rdi, 8].hex())


bp = d.breakpoint(0xdeadbeef, hardware=True, callback=callback)
```


Setting a Breakpoint (part 3: watchpoints)

```
bp = d.breakpoint(
    [...],
    condition: <str | None> = None,
    length: <int> = 1,
    callback: <None | Callable> = None
)
```

```
bp = d.breakpoint(0xdeadbeef, hardware=True, condition="rw")
```

```
bp = d.breakpoint(0xdeadbeef, hardware=True, condition="rw",
length=8, callback=callback)
```

```
bp = d.watchpoint(0xdeadbeef, condition="rw")
```


Setting a Breakpoint (part 4: what's a Breakpoint?)

What can you do with the Breakpoint class?

```
bp.enable()
bp.disable()


print(bp.hit_count)


if bp.hit_on(d): # particularly useful for watchpoints (not yet implemented for watchpoints)
    print(hex(d.rdi))
```


Control Flow Control

Is there anything other than d.cont()?

- `d.step()`
  Steps by a single instruction. This is **hardware assisted** and should always work.
- `d.step_until(position: <int | str>, max_steps: int = -1)`
  Steps until the desired position is reached or for max_steps, whichever comes first.
- `d.finish() - WIP`
  Continues the execution until the current function returns to its caller.

Syscall Hooking

Breakpoints are cool and all of that, but what about syscalls?

```
def enter_write(t, syscall_num):
    print(hex(t.syscall_arg0))
    print(t.memory[t.syscall_arg1, t.syscall_arg2].bytes())

def exit_write(t, syscall_num):
    print(hex(t.syscall_return))

hook = d.hook_syscall("write", on_enter=enter_write, on_exit=exit_write)
```


Demo Time

Multithreading Support (part 1)

What about threads?

```
def callback(t: ThreadContext, bp):
    print(hex(t.rip))
    print(t.memory[d.rdi, 8].hex())
```

`ThreadContext` is not the debugger, it is the thread in the debugged process that hit the specific breakpoint.

Each thread has its id, state, and registers.

The debugger exposes as its own the properties of the first thread.

Multithreading Support (part 2: accessing threads)

```
d = debugger("./threaded_test")
d.run()
d.cont()

for _ in range(15):
    for thread in d.threads:
        print(thread.thread_id, hex(thread.rip))

    d.cont()

d.kill()
```


Multithreading Support (part 3: thread control flow)

During debugging, the threads are either all running or all stopped.

When a thread stops, all other threads are instantly stopped, as in GDB.

`d.cont()` continues all threads.

`d.step(thread: ThreadContext)` can step a single thread.

The one part I forgot: Debugger (part 1)

```
def debugger(
    argv: str | list[str] = [],
    enable_aslr: bool = False,
    env: dict[str, str] | None = None,
    continue_to_binary_entrypoint: bool = True,
    auto_interrupt_on_command: bool = False,
) -> _InternalDebugger:
```

ASLR is disabled by default.

If env is none we inherit from the debugging script.

The one part I forgot: Debugger (part 2: interaction)

```
def debugger(
    [...],
    auto_interrupt_on_command: bool = False,
) -> _InternalDebugger:
```

By default every command waits for the debugged process to stop by itself, and you can force a stop with `d.interrupt()`.
By changing the value of this option to `True`, every command automatically enforces a stop, and you can force a wait with `d.wait()`.

What else is coming?

- Support for other architectures is WIP
  `i386` and `aarch64` are available but experimental
- Support for checkpoints and state save is WIP (ask @lo_no)
- Proper support for signals is WIP (ask @lo_no)
- Support for cross-arch debugging is WIP (~~ask @lo_no~~ idk who to ask)
- Support for syscall emulation is WIP
- Other things I guess?

Thanks for your attention

For any questions, ask @lo_no. He’s got nothing to do.