Ptracer Documentation

Release 0.6.1

Pinterest Inc.

Jul 18, 2022

Contents

ptracer is a library providing on-demand, programmatic system call tracing in Python programs using ptrace.

ptracer works on Python 2.7 and Python 3.5 or later. Currently, only 64-bit Linux platforms are supported.

CHAPTER 1

Installation

ptracer has no external dependencies and the recommended way to install it is to use **pip**:

```
$ pip install ptracer
```

1.1 Building from source

If you want to build **ptracer** from a Git checkout you will need:

- A working C compiler.
- CPython header files. These can usually be obtained by installing the relevant Python development package: **python-dev/python3-dev** on Debian/Ubuntu, **python-devel/python3-devel** on RHEL/Fedora.

Once the above requirements are satisfied, use the usual `setup.py` commands or `pip install -e .` to install the newly built version in development mode.

1.2 Running tests

To execute the testsuite simply run:

```
$ python setup.py test
```

CHAPTER 2

ptracer Usage

The most common way of tracing a block of code is to surround it with the *context()* context manager:

```
import traceback
import ptracer

def callback(syscall):
    print('{}({}) -> {}'.format(
        syscall.name,
        ', '.join(repr(arg.value) for arg in syscall.args),
        syscall.result.text))
    print('Traceback: ')
    print(''.join(traceback.format_list(syscall.traceback)))

with ptracer.context(callback):
    open('/dev/null', 'wb')
```

ptracer also provides the explicit *enable()* and *disable()* functions to begin and terminate tracing.

2.1 Filtering

Ptracer allows elaborate syscall filtering via the *filter* argument:

```
flt = [
    ptracer.SysCallPattern(
        name='open',
        args=[
            re.compile(b'/tmp/.*'),
            lambda arg: arg.value & os.O_WRONLY
        ],
        result=lambda res: res.value > 0
    )
]
```

(continues on next page)

(continued from previous page)

```
with ptracer.context(callback, filter=flt):
    # traced code
    ...
```

In the above example, ptracer will invoke the callback only for successful attempts to open files in the "/tmp" directory for writing.

See the documentation for the *SysCallPattern* class for more information on setting up filters.

CHAPTER 3

Module Reference

context(*callback*, *filter=None*)
Set up and return a tracing context object that should be used as a context manager. Tracing will begin once the context manager block is entered, and will terminate on block exit.

The *callback* parameter specifies a callable that should accept a *SysCall* instance as a single argument. The *callback* is invoked asynchronously in a thread separate from the traced program.

If *filter* is not `None`, it is expected to contain a *SysCallPattern* instance or an iterable of `SysCallPattern` instances. The *callback* will be called if the syscall matches any of the provided patterns. If *filter* is `None`, no filtering is done, and *callback* will be invoked for every syscall.

enable(*callback*, *filter=None*)
Start tracing of the current program immediately. The *callback* and *filter* arguments have the same meaning as in *context()*. To stop tracing call *disable()*.

disable()
Stop tracing of the current program.

class SysCall
A description of a system call performed by a program. `SysCall` instances are passed to the callback passed to *context()* or *enable()*.

name
The name of the system call. If the name could not be identified, the property will contain `'<syscallnumber>'`, where `syscallnumber` is a platform-specific integer representing the system call.

pid
The system identifier of the OS thread in which the system call was performed.

args
A list of *SysCallArg* instances representing the system call arguments. The values of the arguments are taken *after* the system call exit.

result
An instance of *SysCallResult* representing the system call return value.

traceback
A list of stack trace entries similar to the one returned by `traceback.extract_stack`.

The trace corresponds to the call stack which triggered the system call.

class SysCallArg
A description of a system call argument. Instances of *SysCall* contain a list of `SysCallArg` objects in the `args` attribute.

name
The name of the syscall parameter. If the name could not be identified, this property will contain `paramN` for the N-th argument.

type
The type of the syscall parameter represented by a *CType* instance. If the real type could not be identified, the type will be reported as `unsigned long`.

raw_value
An integer representing the raw value of the syscall argument.

value
An object representing the unpacked value of the syscall argument according to its type. For pointer values this will be the dereferenced value. Known types will be converted into corresponding Python values.

class SysCallResult
A description of a system call return value. Instances of *SysCall* contain an `SysCallResult` object in the `result` attribute.

type
The type of the syscall return value represented by a *CType* instance. If the real type could not be identified, the type will be reported as `unsigned long`.

raw_value
An integer representing the raw value of the syscall return value.

value
An object representing the unpacked value of the syscall return value according to its type. For pointer values this will be the dereferenced value. Known types will be converted into corresponding Python values.

class CType
A description of a system call value type.

names
A list of tokens in the C declaration of the type. For example, `'unsigned long'` will be represented as `['unsigned', 'long']`.

ctype
A ctypes data type.

ptr_indirection
The number of pointer indirections. For example, a `'const char **'` type will have `ptr_indirection` of 2, and the `ctype` attribute set to `c_char`.

class SysCallPattern(*name=None*, *args=None*, *result=None*)
An object used to match system calls. *name*, *args*, and *result* specify the *patterns* for the corresponding attributes of the *SysCall* object. If specified, *args*, should be a list of patterns matching the order of syscall arguments, and not all arguments have to be listed. Each pattern value can be:

- A callable that receives a *SysCallArg* or a *SysCallResult* instance and returns `True` when the value matches, and `False` otherwise.

- An object with a `match()` method that received an unpacked value of a syscall attribute and returns `True` when the value matches, and `False` otherwise. A regular expression object can be used. For example: `SysCallPattern(name=re.compile('open.*'))`.
- Any python object, which is compared with the unpacked value directly.

match (*syscall*)
Return `True` if *syscall* matches the pattern and `False` otherwise.

Python Module Index

p

Index