python-ptrace Documentation

Release 0.9.3

Victor Stinner

February 13, 2017

Contents

python-ptrace is a debugger using ptrace (Linux, BSD and Darwin system call to trace processes) written in Python.

- python-ptrace documentation
- python-ptrace at GitHub
- python-ptrace at the Python Cheeseshop (PyPI)

python-ptrace is an opensource project written in Python under GNU GPLv2 license.

CHAPTER 1

Features

- High level Python object API : PtraceDebugger and PtraceProcess
- Able to control multiple processes: catch fork events on Linux
- Read/write bytes to arbitrary address: take care of memory alignment and split bytes to cpu word
- Execution step by step using ptrace_singlestep() or hardware interruption 3
- Can use distorm disassembler
- Dump registers, memory mappings, stack, etc.
- *Syscall tracer and parser* (strace.py command)

Status:

- Supported operating systems: Linux, FreeBSD, OpenBSD
- Supported architectures: x86, x86_64 (Linux), PPC (Linux), ARM (Linux EAPI)

Missing features:

- Symbols: it's not possible to break on a function or read a variable value
- No C language support: debugger shows assembler code, not your C (C++ or other language) code!
- No thread support

CHAPTER 2

Table of Contents

2.1 Install python-ptrace

2.1.1 Linux packages

- Debian: python-ptrace Debian package.
- Mandriva: python-ptrace Mandriva package
- OpenEmbedded: python-ptrace recipe
- Arch Linux: python-ptrace Arch Linux package
- Gentoo: dev-python/python-ptrace

See also python-ptrace on Python Package Index (PyPI)

2.1.2 Install from source

Download tarball

Get the latest tarball at the Python Package Index (PyPI).

Download development version

Download the development version using Git:

```
git clone https://github.com/haypo/python-ptrace.git
```

Browse python-ptrace source code.

python-ptrace dependencies

- Python 2.6+/3.3+: http://python.org/
- distorm disassembler (optional) http://www.ragestorm.net/distorm/

Installation

Type as root:

```
python setup.py install
```

Or using sudo program:

```
sudo python setup.py install
```

2.1.3 cptrace

For faster debug and to avoid ctypes, you can also install cptrace: Python binding of the ptrace() function written in C:

```
python setup_cptrace.py install
```

2.1.4 Run tests

Run tests with tox

The tox project can be used to build a virtual environment run tests against different Python versions (Python 2 and Python 3).

To run all tests on Python 2 and Python 3, just type:

```
tox
```

To only run tests on Python 2.7, type:

```
tox -e py2
```

Available environments:

- `py2`: Python 2
- `py3`: Python 3

Run tests manually

Type:

```
python runtests.py
python test_doc.py
```

It's also possible to run a specific test:

```
PYTHONPATH=$PWD python tests/test_strace.py
```

2.2 python-ptrace usage

2.2.1 Hello World

Short example attaching a running process. It gets the instruction pointer, executes a single step, and gets the new instruction pointer:

```
import ptrace.debugger
import signal
import subprocess
import sys

def debugger_example(pid):
    debugger = ptrace.debugger.PtraceDebugger()

    print("Attach the running process %s" % pid)
    process = debugger.addProcess(pid, False)
    # process is a PtraceProcess instance
    print("IP before: %#x" % process.getInstrPointer())

    print("Execute a single step")
    process.singleStep()
    # singleStep() gives back control to the process. We have to wait
    # until the process is trapped again to retrieve the control on the
    # process.
    process.waitSignals(signal.SIGTRAP)
    print("IP after: %#x" % process.getInstrPointer())

    process.detach()
    debugger.quit()

def main():
    args = [sys.executable, '-c', 'import time; time.sleep(60)']
    child_popen = subprocess.Popen(args)
    debugger_example(child_popen.pid)
    child_popen.kill()
    child_popen.wait()

if __name__ == "__main__":
    main()
```

2.2.2 API

PtraceProcess

The PtraceProcess class is an helper to manipulate a traced process.

Example:

```
tracer = PtraceProcess(pid)              # attach the process
tracer.singleStep()                      # execute one instruction
tracer.cont()                            # continue execution
tracer.syscall()                         # break at next syscall
tracer.detach()                          # detach process

# Get status
tracer.getreg('al')                      # get AL register value
regs = tracer.getregs()                  # read all registers
bytes = tracer.readBytes(regs.ax, 10)    # read 10 bytes
tracer.dumpCode()                        # dump code (as assembler or hexa is the disassembler is mi
tracer.dumpStack()                       # dump stack (memory words around ESP)

# Modify the process
shellcode = '...'
```

```
ip = tracer.getInstrPointer()             # get EIP/RIP register
bytes = tracer.writeBytes(ip, shellcode) # write some bytes
tracer.setreg('ebx', 0)                   # set EBX register value to zero
```

Read `ptrace/debugger/process.py` source code to see more methods.

2.3 Trace system calls (syscalls)

python-ptrace can trace system calls using `PTRACE_SYSCALL`.

2.3.1 PtraceSyscall

ptrace.syscall module contains PtraceSyscall class: it's a parser of Linux syscalls similar to strace program.

Example:

```
connect(5, <sockaddr_in sin_family=AF_INET, sin_port=53, sin_addr=212.27.54.252>, 28) = 0
open('/usr/lib/i686/cmov/libcrypto.so.0.9.8', 0, 0 <read only>) = 4
mmap2(0xb7e87000, 81920, 3, 2066, 4, 297) = 0xb7e87000
rt_sigaction(SIGWINCH, 0xbfb7d4a8, 0xbfb7d41c, 8) = 0
```

You can get more information: result type, value address, argument types, and argument names.

Examples:

```
long open(const char* filename='/usr/lib/i686/cmov/libcrypto.so.0.9.8' at 0xb7efc027, int flags=0, in
long fstat64(unsigned long fd=4, struct stat* buf=0xbfa46e2c) = 0
long set_robust_list(struct robust_list_head* head=0xb7be5710, size_t len_ptr=12) = 0
```

2.3.2 strace.py

Program strace.py is very close to strace program: display syscalls of a program. Example:

Features

- Nice output of signal: see [[signal|python-ptrace signal handling]]
- Supports multiple processes
- Can trace running process
- Can display arguments name, type and address
- Option `--filename` to show only syscall using file names
- Option `--socketcall` to show only syscall related to network (socket usage)
- Option `--syscalls` to list all known syscalls

Example

```
$ ./strace.py /bin/ls
execve(/bin/ls, [['/bin/ls'],|[/* 40 vars */]]) = 756
brk(0)                                  = 0x0805c000
access('/etc/ld.so.nohwcap', 0)         = -2 (No such file or directory)
mmap2(NULL, 8192, 3, 34, -1, 0)         = 0xb7f56000
access('/etc/ld.so.preload', 4)         = -2 (No such file or directory)
(...)
close(1)                                = 0
munmap(0xb7c5c000, 4096)                = 0
exit_group(0)
---done---
```

Options

The program has many options. Example with `--socketcall` (display only network functions):

```
$ ./strace.py --socketcall nc localhost 8080
execve(/bin/nc, [['/bin/nc',|'localhost', '8080']], [[/*|40 vars */]]) = 12948
socket(AF_FILE, SOCK_STREAM, 0)         = 3
connect(3, <sockaddr_un sun_family=AF_FILE, sun_path=/var/run/nscd/socket>, 110) = -2 (No such file o
socket(AF_FILE, SOCK_STREAM, 0)         = 3
connect(3, <sockaddr_un sun_family=AF_FILE, sun_path=/var/run/nscd/socket>, 110) = -2 (No such file o
socket(AF_INET, SOCK_STREAM, 6)         = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, 3217455272L, 4) = 0
connect(3, <sockaddr_in sin_family=AF_INET, sin_port=8080, sin_addr=127.0.0.1>, 16) = -111 (Connectio
(...)
```

2.4 gdb.py

`gdb.py` is a command line debugger *similar to gdb*, but with fewer features: no symbol support, no C language support, no thread support, etc.

2.4.1 Some commands

- `cont`: continue execution
- `stepi`: execute one instruction
- `step`: execute one instruction, but don't enter into calls

Type `help` to list all available commands.

2.4.2 Features

- print command displays value as decimal and hexadecimal, but also the related memory mapping (if any):

```
(gdb) print $eip
Decimal: 3086383120
Hexadecimal: 0xb7f67810
Address is part of mapping: 0xb7f67000-0xb7f81000 => /lib/ld-2.6.1.so (r-xp)
```

- Nice output of signal: see [[signal|python-ptrace signal handling]]

- Syscall tracer with command "sys": see *python-ptrace system call tracer <syscall>*. Short example:

```
(gdb) sys
long access(char* filename='/etc/ld.so.nohwcap' at 0xb7f7f35b, int mode=F_OK) = -2 (No such file
```

- Supports multiple processes:

```
(gdb) proclist
<PtraceProcess #24187> (active)
<PtraceProcess #24188>
(gdb) proc
Process ID: 24187 (parent: 24182)
Process state: T (traced)
Process command line: [['tests/fork_execve']
(...)
(gdb)|switch; proc
Switch to <PtraceProcess #24188>
Process ID: 24188 (parent: 24187)
Process state: T (traced)
Process command line: ['/bin/ls']]
(...)
```

- Allow multiple commands on the same line using ";" separator:

```
(gdb) print $eax; set $ax=0xdead; print $eax
Decimal: 0
Hexadecimal: 0x00000000
Set $ax to 57005
Decimal: 57005
Hexadecimal: 0x0000dead
```

- Only written in pure Python code, so it's easy to extend
- Expression parser supports all arithmetic operator (`a+b`, `a/b`, `a<<b`, `a&b`, `...`), parenthesis, use of registers, etc. and pointer dereference (ex: `print *($ebx+0xc)`).

2.4.3 Screenshot

```
$ ./gdb.py ls
execve(/bin/ls, [['/bin/ls'],|[/* 40 vars */]]) = 16182
(gdb) where
ASM 0xb7f47810: MOV EAX, ESP <==
ASM 0xb7f47812: CALL 0xb7f47a60
ASM 0xb7f47817: MOV EDI, EAX
ASM 0xb7f47819: CALL 0xb7f47800
(gdb) regs
     EBX = 0xb7f4781e
     ECX = 0x0001d2f4
     EDX = 0xb7f61ff4
     ESI = 0x00000000
     (...)
(gdb) proc
Process ID: 16182
Process command line: [['/bin/ls']
Process|environment: ['TERM=xterm', 'SHELL=/bin/bash', (...)]]
Process working directory: /home/haypo/prog/fusil/ptrace/trunk
(gdb) stack
STACK: 0xbfc58000..0xbfc6e000
```

```
STACK -8: 0x00000000
STACK -4: 0xb7f4781e
STACK +0: 0x00000001
STACK +4: 0xbfc6c6bb
STACK +8: 0x00000000
(gdb) maps
MAPS: 08048000-0805b000 r-xp 00000000 08:03 2588939    /bin/ls
MAPS: 0805b000-0805c000 rw-p 00012000 08:03 2588939    /bin/ls
(...)
MAPS: b7f61000-b7f63000 rw-p 00019000 08:03 1540553    /lib/ld-2.6.1.so
MAPS: bfc58000-bfc6e000 rw-p bfc58000 00:00 0          [[stack]
MAPS:|ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]]
(gdb) quit
Quit.
Terminate <PtraceProcess pid=16182>
Quit gdb.
```

2.5 python-ptrace process events

2.5.1 Process events

All process events are based on ProcessEvent class.

- ProcessExit: process exited with an exitcode, killed by a signal or exited abnormally
- ProcessSignal: process received a signal
- NewProcessEvent: new process created, e.g. after a fork() syscall

Attributes:

- All events have a “process” attribute
- ProcessExit has “exitcode” and “signum” attributes (both can be None)
- ProcessSignal has “signum” and “name” attributes

For NewProcessEvent, use process.parent attribute to get the parent process.

Note: ProcessSignal has a display() method to display its content. Use it just after receiving the message because it reads process memory to analyze the reasons why the signal was sent.

2.5.2 Wait for any process event

The most generic function is waitProcessEvent(): it waits for any process event (exit, signal or new process):

```
event = debugger.waitProcessEvent()
```

To wait one or more signals, use waitSignals() methods. With no argument, it waits for any signal. Events different than signal are raised as Python exception. Examples:

```
signal = debugger.waitSignals()
signal = debugger.waitSignals(SIGTRAP)
signal = debugger.waitSignals(SIGINT, SIGTERM)
```

Note: signal is a ProcessSignal object, use signal.signum to get the signal number.

2.5.3 Wait for a specific process events

To wait any event from a process, use waitEvent() method:

```
event = process.waitEvent()
```

To wait one or more signals, use waitSignals() method. With no argument, it waits for any signal. Other process events are raised as Python exception. Examples:

```
signal = process.waitSignals()
signal = process.waitSignals(SIGTRAP)
signal = process.waitSignals(SIGINT, SIGTERM)
```

Note: As debugger.waitSignals(), signal is a ProcessSignal object.

2.6 python-ptrace signal handling

2.6.1 Introduction

PtraceSignal tries to display useful information when a signal is received. Depending on the signal number, it shows different information.

It uses the current instruction decoded as assembler code to understand why the signal is raised.

Only Intel x86 (i386, maybe x86_64) is supported now.

When a process receives a signal, python-ptrace tries to explain why the signal was emitted.

General information (not shown for all signals, e.g. not for SIGABRT):

- CPU instruction causing the crash
- CPU registers related to the crash
- Memory mappings of the memory addresses

Categorize signals:

- SIGFPE
  - Division by zero
- SIGSEGV, SIGBUS
  - Invalid memory read
  - Invalid memory write
  - Stack overflow
  - Invalid memory access
- SIGABRT
  - Program abort
- SIGCHLD
  - Child process exit

2.6.2 Examples

Division by zero (SIGFPE)

```
Signal: SIGFPE
Division by zero
- instruction: IDIV DWORD [[EBP-0x8]
- register ebp=0xbfdc4a98
```

Invalid memory read/write (SIGSEGV)

```
Signal: SIGSEGV
Invalid read from 0x00000008
- instruction: MOV EAX, [EAX+0x8]]
- mapping: 0x00000008 is not mapped in memory
- register eax=0x00000000
```

```
PID: 23766
Signal: SIGSEGV
Invalid write to 0x00000008 (size=4 bytes)
- instruction: MOV DWORD [[EAX+0x8],|0x2a
- mapping: 0x00000008..0x0000000b is not mapped in memory
- register eax=0x00000000
```

Given information:

- Address of the segmentation fault
- (if possible) Size of the invalid memory read/write
- CPU instruction causing the crash
- CPU registers related to the crash
- Memory mappings of the related memory address

Stack overflow (SIGSEGV)

```
Signal: SIGSEGV
STACK OVERFLOW! Stack pointer is in 0xbf534000-0xbfd34000 => [stack]] (rw-p)
- instruction: MOV BYTE [[EBP-0x1004],|0x0
- mapping: 0xbf533430 is not mapped in memory
- register <stack ptr>=0xbf533430
- register ebp=0xbf534448
```

Child exit (SIGCHLD)

```
PID: 24008
Signal: SIGCHLD
Child process 24009 exited normally
Signal sent by user 1000
```

Information:

- Child process identifier
- Child process user identifier

2.6.3 Examples

Invalid read:

```
Signal: SIGSEGV
Invalid read from 0x00000008
- instruction: MOV EAX, [EAX+0x8]
- mapping: (no memory mapping)
- register eax=0x00000000
```

Invalid write (MOV):

```
Signal: SIGSEGV
Invalid write to 0x00000008 (size=4 bytes)
- instruction: MOV DWORD [EAX+0x8], 0x2a
- mapping: (no memory mapping)
- register eax=0x00000000
```

abort():

```
Signal: SIGABRT
Program received signal SIGABRT, Aborted.
```

2.6.4 Source code

See:

- `ptrace/debugger/ptrace_signal.py`
- `ptrace/debugger/signal_reason.py`

2.7 cptrace Python module

Python binding for ptrace written in C.

2.7.1 Example

Dummy example:

```
>>> import cptrace
>>> cptrace.ptrace(1, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ptrace(request=1, pid=1, 0x(nil), 0x(nil)) error #1: Operation not permitted
```

2.8 Authors

2.8.1 Contributors

- Anthony Gelibert <anthony.gelibert AT me.com> - fix cptrace on Apple
- Dimitris Glynos <dimitris AT census-labs.com> - follow gdb.py commands

- Jakub Wilk <jwilk AT debian.org> - fix for GNU/kFreeBSD
- Mark Seaborn <mrs AT mythic-beasts.com> - Create -i option for strace
- Mickaël Guérin aka KAeL <mickael.guerin AT crocobox.org> - OpenBSD port
- teythoon - convert all classes to new-style classes
- 20. Bursztyka aka Tuna <tuna AT lyua.org> - x86_64 port
- Victor Stinner <victor.stinner AT gmail.com> - python-ptrace author and maintainer

2.8.2 Thanks

- procps authors (top and uptime programs)

2.9 Changelog

2.9.1 python-ptrace 0.9.2 (2017-02-12)

- Issue #35: Fix strace.py when tracing multiple processes: use the correct process. Fix suggested by Daniel Trnka.

2.9.2 python-ptrace 0.9.1 (2016-10-12)

- Added tracing of processes created with the clone syscall (commonly known as threads).
- gdb.py: add `gcore` command, dump the process memory.
- Allow command names without absolute path.
- Fix ptrace binding: clear errno before calling ptrace().
- Fix PtraceSyscall.exit() for unknown error code
- Project moved to GitHub: https://github.com/haypo/python-ptrace
- Remove the `ptrace.ctypes_errno` module: use directly the `ctypes.get_errno()` function
- Remove the `ptrace.ctypes_errno` module: use directly `ctypes.c_int8`, `ctypes.c_uint32`, ... types

2.9.3 python-ptrace 0.9 (2016-04-23)

- Add all Linux syscall prototypes
- Add error symbols (e.g. ENOENT), in addition to error text, for strace
- Fix open mode so O_RDONLY shows if it's the only file access mode
- Python 3: fix formatting of string syscall arguments (ex: filenames), decode bytes from the locale encoding
- Issue #17: syscall parser now supports O_CLOEXEC and SOCK_CLOEXEC, fix unit tests on Python 3.4 and newer

2.9.4 python-ptrace 0.8.1 (2014-10-30)

- Update MANIFEST.in to include all files
- Fix to support Python 3.5

2.9.5 python-ptrace 0.8 (2014-10-05)

- Issue #9: Rewrite waitProcessEvent() and waitSignals() methods of PtraceProcess to not call waitpid() with pid=-1. There is a race condition with waitpid(-1) and fork, the status of the child process may be returned before the debugger is noticed of the creation of the new child process.
- Issue #10: Fix PtraceProcess.readBytes() for processes not created by the debugger (ex: fork) if the kernel blocks access to private mappings of /proc/pid/mem. Fallback to PTRACE_PEEKTEXT which is slower but a debugger tracing the process is always allowed to use it.

2.9.6 python-ptrace 0.7 (2013-03-05)

- Experimental support of Python 3.3 in the same code base
- Drop support of Python 2.5
- Remove the ptrace.compatibility module
- Fix Process.readStruct() and Process.readArray() on x86_64
- Experimental support of ARM architecture (Linux EAPI), strace.py has been tested on Raspberry Pi (armv6l)

2.9.7 python-ptrace 0.6.6 (2013-12-16)

- Fix os_tools.RUNNING_LINUX for Python 2.x compiled on Linux kernel 3.x
- Support FreeBSD on x86_64
- Add missing prototype of the unlinkat() system call. Patch written by Matthew Fernandez.

2.9.8 python-ptrace 0.6.5 (2013-06-06)

- syscall: fix parsing socketcall on Linux x86
- syscall: fix prototype of socket()

2.9.9 python-ptrace 0.6.4 (2012-02-26)

- Convert all classes to new-style classes, patch written by teythoon
- Fix compilation on Apple, patch written by Anthony Gelibert
- Support GNU/kFreeBSD, patch written by Jakub Wilk
- Support sockaddr_in6 (IPv6 address)

2.9.10 python-ptrace 0.6.3 (2011-02-16)

- Support distrom3
- Support Python 3
- Rename strace.py option –socketcall to –socket, and fix this option for FreeBSD and Linux/64 bits
- Add MANIFEST.in: include all files in source distribution (tests, cptrace module, ...)

2.9.11 python-ptrace 0.6.2 (2009-11-09)

- Fix 64 bits sub registers (set mask for eax, ebx, ecx, edx)

2.9.12 python-ptrace 0.6.1 (2009-11-07)

- Create follow, showfollow, resetfollow, xray commands in gdb.py. Patch written by Dimitris Glynos
- Project website moved to http://bitbucket.org/haypo/python-ptrace/
- Replace types (u)intXX_t by c_(u)intXX
- Create MemoryMapping.search() method and MemoryMapping now keeps a weak reference to the process

2.9.13 python-ptrace 0.6 (2009-02-13)

User visible changes:

- python-ptrace now depends on Python 2.5
- Invalid memory access: add fault address in the name
- Update Python 3.0 conversion patch
- Create -i (–show-ip) option to strace.py: show instruction pointer
- Add a new example (itrace.py) written by Mark Seaborn and based on strace.py

API changes:

- PtraceSyscall: store the instruction pointer at syscall enter (if the option instr_pointer=True, disabled by default)
- Remove PROC_DIRNAME and procFilename() from ptrace.linux_proc

Bugfixes:

- Fix locateProgram() for relative path
- Fix interpretation of memory fault on MOSVW instruction (source is ESI and destination is EDI, and not the inverse!)

2.9.14 python-ptrace 0.5 (2008-09-13)

Visible changes:

- Write an example (the most simple debugger) and begin to document the code
- gdb.py: create “dbginfo” command
- Parse socket syscalls on FreeBSD

- On invalid memory access (SIGSEGV), eval the dereference expression to get the fault address on OS without siginfo (e.g. FreeBSD)
- Fixes to get minimal Windows support: fix imports, fix locateProgram()

Other changes:

- Break the API: - Rename PtraceDebugger.traceSysgood() to PtraceDebugger.enableSysgood() - Rename PtraceDebugger.trace_sysgood to PtraceDebugger.use_sysgood - Remove PtraceProcess.readCode()
- Create createChild() function which close all files except stdin, stdout and stderr
- On FreeBSD, on process exit recalls waitpid(pid) to avoid zombi process

2.9.15 python-ptrace 0.4.2 (2008-08-28)

- BUGFIX: Fix typo in gdb.py (commands => command_str), it wasn't possible to write more than one command...
- BUGFIX: Fix typo in SignalInfo class (remove "self."). When a process received a signal SIGCHLD (because of a fork), the debugger exited because of this bug.
- BUGFIX: Debugger._wait() return abnormal process exit as a normal event, the event is not raised as an exception
- PtraceSignal: don't clear preformatted arguments (e.g. arguments of execve)

2.9.16 python-ptrace 0.4.1 (2008-08-23)

- The project has a new dedicated website: http://python-ptrace.hachoir.org/
- Create cptrace: optional Python binding of ptrace written in C (faster than ptrace, the Python binding written in Python with ctypes)
- Add name attribute to SignalInfo classes
- Fixes to help Python 3.0 compatibility: don't use sys.exc_clear() (was useless) in writeBacktrace()
- ProcessState: create utime, stime, starttime attributes

2.9.17 python-ptrace 0.4.0 (2008-08-19)

Visible changes:

- Rename the project to "python-ptrace" (old name was "Ptrace)
- strace.py: create –ignore-regex option
- PtraceSignal: support SIGBUS, display the related registers and the instruction
- Support execve() syscall tracing

Developer changes:

- New API is incompatible with 0.3.2
- PtraceProcess.waitProcessEvent() accepts optional blocking=False argument
- PtraceProcess.getreg()/setreg() are able to read/write i386 and x86-64 "sub-registers" like al or bx
- Remove iterProc() function, replaced by openProc() with explicit call to .close() to make sure that files are closed

- Create searchProcessesByName()
- Replace CPU_PPC constant by CPU_POWERPC and create CPU_PPC32 and CPU_PPC64
- Create MemoryMapping object, used by readMappings() and findStack() methods of PtraceProcess
- Always define all PtraceProcess methods but raise an error if the function is not implemented

2.10 TODO

2.10.1 Main tasks

- Fix strace.py –socketcall: SyscallState.enter() calls ignore_callback before socketcall are proceed
- Support other backends:
  - GDB MI: http://code.google.com/p/pygdb/
  - ktrace: (FreeBSD and Darwin): would help Darwin support
  - utrace (new Linux debugger): http://sourceware.org/systemtap/wiki/utrace
  - vtrace?
  - PyDBG: works on Windows
- Backtrace symbols:
  - GNU BFD?
  - elfsh?
  - addr2line program?
  - See dl_iterate_phdr() function of libdl
- Support other disassemblers (than distorm), and so both Intel syntax (Intel and AT&T)
  - BFD
  - http://www.ragestorm.net/distorm/
  - libasm (ERESI)
  - libdisasm (bastard)
  - http://www.woodmann.com/collaborative/tools/index.php/Category:X86_Disassembler_Libraries
- Support threads: other backends (than python-ptrace) already support threads

2.10.2 Minor tasks

- Fix gdb.py “step” command on a jump. Example where step will never stop:

```
(gdb) where
ASM 0xb7e3b917: JMP 0xb7e3b8c4 (eb ab)
ASM 0xb7e3b919: LEA ESI, [ESI+0x0] (8db426 00000000)
```

- Remove gdb.py “except PtraceError, err: if err.errno == ESRCH” hack, process death detection should be done by PtraceProcess or PtraceDebugger
- Use Intel hardware breakpoints: set vtrace source code

- Support Darwin:
  - ktrace? need to recompile Darwin kernel with KTRACE option
  - get registers: http://unixjunkie.blogspot.com/2006/01/darwin-ptrace-and-registers.html
  - PT_DENY_ATTACH: http://steike.com/code/debugging-itunes-with-gdb/
  - PT_DENY_ATTACH: http://landonf.bikemonkey.org/code/macosx/ptrace_deny_attach.20041010201303.11809.mojo.html

CHAPTER 3

Links

3.1 Project using python-ptrace

- Fusil the fuzzer

3.2 python-ptrace announces

- fuzzing mailing list
- reverse-engineering.net

3.3 ptrace usage

- Sandboxing: Plash

3.4 Similar projects

- vtrace: Python library (Windows and Linux) supporting threads
- subterfuge by Mike Coleman: Python library (Linux): contains Python binding of ptrace written in C for Python 2.1/2.2. It doesn't work with Python 2.5 (old project, not maintained since 2002)
- strace program (Linux, BSD)
- ltrace program (Linux)
- truss program (Solaris and BSD)
- pytstop by Philippe Biondi: debugger similar to gdb but in very alpha stage (e.g. no disassembler), using ptrace Python binding written in C (from subterfuge)
- strace.py by Philippe Biondi
- Fenris: suite of tools suitable for code analysis, debugging, protocol analysis, reverse engineering, forensics, diagnostics, security audits, vulnerability research
- PyDBG: Windows debugger written in pure Python

3.5 Interesting articles

- (fr) Surveiller les connexions avec auditd (2007)
- Playing with ptrace() for fun and prot (2006)
- PTRACE_SETOPTIONS tests (2005)
- Process Tracing Using Ptrace (2002)