用GDB调试分析Python解释器

安全研究部 陈庆

有些 Python 程序无法被 Ctrl-C 打断，也无法处理 SIGTERM 信号。本文利用 GDB 对 Python 解释器进行调试分析，找出前述现象的深层次原因。

一、令人困惑的现象

一段最简演示代码如下：

```
import sys, os, signal, time, threading
def test_0 () :
    while True :
        time.sleep( 1 )
def test_1 () :
    cond    = threading.Condition()
    cond.acquire()
    cond.wait()
def on_SIGTERM ( signum, frame ) :
    print 'signum = %u' % signum
def main ( prog, args ) :
    print os.getpid()
    signal.signal( signal.SIGTERM, on_SIGTERM )
    if 0 == len( args ) or 0 == int( args[0], 0 ) :
        print 'call test_0()'
        test_0()
    else :
        print 'call test_1()'
        test_1()
    print 'Reachless'
if '__main__' == __name__ :
    try :
        main( os.path.basename( sys.argv[0] ), sys.argv[1:] )
    except KeyboardInterrupt :
        pass
```

执行 "DebugPythonWithGDB.py 1", 发现 Ctrl-C 无法将其中止，kill 时 SIGTERM 信号句柄未被执行 (无 "signum = 15" 输出)。

这个问题最初是西安研发中心张龙提出的。我研究了一下，发现这个现象背后的原因比我想像的复杂得多。

二、准备调试环境

现在新版 GDB、新的 Linux 发行版对调试 Python 已经有了很好的支持，不需要四处打 Patch。本文在 x86/Debian 8.0 上演示。

安装调试版本的 Python 解释器：

```
# aptitude install python2.7-dbg
```

下载 Python 源码：

```
[root@ /usr/src/python]> apt-get source python2.7-dbg
```

编辑 GDB 初始化文件：

```
set debug-file-directory /usr/lib/debug
set directories /usr/src/python/python2.7-2.7.9/Modules
```

查看 "/usr/lib/debug/usr/bin/python2.7-gdb.py"，在其中搜索 "py-"，寻找可以在 (gdb) 提示符下使用的 Python 调试命令：

```
py-bt
    调用栈回溯
py-up
py-down
    切换栈帧，up 向大数方向移动，down 向小数方向移动
py-list
py-list start
py-list start, end
    显示当前栈帧的源代码
py-locals
    显示当前栈帧所有的局部变量
py-print ...
    显示当前栈帧指定的局部变量
```

如果在原生 bt 的显示中看到 "<value optimized out>"，不要使用 "py-*" 系列命令，否则有可能触发 SIGSEGV。

三、调试分析 Python 解释器

```
$ /usr/bin/python2.7-dbg DebugPythonWithGDB.py 1
28082
call test_1()
```

利用 SystemTap 查看 Python 解释器此时安装的 C 级信号句柄：

```
# stap -DMAXACTION=10000 -g psig.stp -x $(pgrep -f DebugPythonWithGDB.py) | grep TERM
TERM    caught   0x8216f2d  0
```

这个信息表示针对 SIGTERM 安装有信号句柄，其入口在 0x8216f2d，结尾的 0 对应 sa_flags，表明没有指定 SA_RESTART。

用 GDB 调试目标进程，针对 SIGTERM 的信号句柄设置断点，调整 GDB 对 SIGTERM 的处理方式：

```
# gdb -q -ex "b *0x8216f2d" -ex "handle SIGTERM nostop print pass" -ex c -p $(pgrep -f DebugPythonWithGDB.py)
```

从另一个终端向 Python 解释器发送 SIGTERM 信号：

```
# kill -TERM $(pgrep -f DebugPythonWithGDB.py)
```

GDB 中断点命中：

```
Program received signal SIGTERM, Terminated.
Breakpoint 1, signal_handler (sig_num=15) at ../Modules/signalmodule.c:185
```

查看调用栈回溯：

```
(gdb) bt
#0  signal_handler (sig_num=15) at ../Modules
```

```
/signalmodule.c:185
#1  <signal handler called>
#2  0xb772dd3c in __kernel_vsyscall ()
#3  0xb76f8df5 in sem_wait@@GLIBC_2.1 ()
#4  0x0818716e in PyThread_acquire_lock
#5  0x0822e43c in lock_PyThread_acquire_lock
#6  0x080bc300 in PyCFunction_Call
#7  0x08149d0b in call_function
#8  0x081454ec in PyEval_EvalFrameEx at ../
Python/ceval.c:2679
```

此时断在 C 级信号句柄中，让流程回到 sem_wait() 所在：

```
(gdb) select-frame 3
(gdb) finish
...
0x0818716e in PyThread_acquire_lock at ../Python/thread_
pthread.h:324
(gdb) list
319             (void) error; /* silence unused-but-set-variable
warning */
320             dprintf(("PyThread_acquire_lock(%p, %d)
called\n", lock, waitflag));
321
322         do {
323             if (waitflag)
324                 status = fix_status(sem_wait(thelock));
325             else
326                 status = fix_status(sem_trywait(thelock));
327         } while (status == EINTR); /* Retry if interrupted by
a signal */
328
(gdb) output waitflag
1
```

327 行有一个可疑的循环，看看能否跳出循环：

```
(gdb) b 327
Breakpoint 2 at 0x818719f: file ../Python/thread_pthread.h,
line 327.
(gdb) disable 1
(gdb) c
Continuing.
Breakpoint 2, PyThread_acquire_lock (lock=0x983f8b8,
waitflag=1) at ../Python/thread_pthread.h:327
327         } while (status == EINTR); /* Retry if interrupted by
a signal */
(gdb) output status
4
```

4 就是 EINTR，表示 sem_wait() 被 SIGTERM 打断，返回

EINTR。然后从 327 行继续循环调用 sem_wait()。

此时的 C 级调用栈回溯：

```
(gdb) bt
#0  PyThread_acquire_lock at ../Python/thread_pthread.
h:327
#1  0x0822e43c in lock_PyThread_acquire_lock
#2  0x080bc300 in PyCFunction_Call
#3  0x08149d0b in call_function
#4  0x081454ec in PyEval_EvalFrameEx at ../
Python/ceval.c:2679
```

此时的 Python 级调用栈回溯：

```
(gdb) py-bt
#4 Frame 0xb747fcf4
   waiter.acquire()
#8 Frame 0xb73b146c, for file DebugPythonWithGDB.
py, line 13, in test_1
   cond.wait()
#11 Frame 0xb735f1ac, for file DebugPythonWithGDB.
py, line 26, in main
   test_1()
#14 Frame 0xb740f49c, for file DebugPythonWithGDB.
py, line 31
main( os.path.basename( sys.argv[0] ), sys.argv[1:] )
```

流程位于 PyEval_EvalFrameEx() 中，查看 "Python/ceval.c:2679"，在解释执行 PVM 指令的主 switch 中，具体是 "case CALL_FUNCTION:"。

现在我们知道了这里存在死循环或者说无限循环，但这与 Python 级信号句柄有什么关系？C 级信号句柄与 Python 级信号句柄是如何发生关联的？

四、Python 级信号句柄与延迟调用

在 Python 代码中调用 signal.signal() 安装信号句柄时，实际调用 signal_signal()：

```
static struct
{
    /*
     * 是否收到信号。signal_handler() 得到执行时将该成员设为 1。
     */
    int        tripped;
    /*
     * Python 级的信号句柄。
     */
    PyObject   *func;
} Handlers[NSIG];
static PyObject * signal_signal ( PyObject *self,
PyObject *args )
```

```
{
    ...
    else
    {
        func    = signal_handler;
    }
    /*
     * PyOS_setsig() 最终调了 C 函数 sigaction() 或 signal()
安装信号句柄。这个封
     * 装很简单，sa_flags 保持为 0，没有指定 SA_
RESTART。调了
     * siginterrupt(sig,1)，尽可能打断系统调用，
使之返回 EINTR。
     */
    if ( PyOS_setsig( sig_num, func ) == SIG_ERR )
        ...
    old_handler             = Handlers[sig_num].func;
    Handlers[sig_num].tripped   = 0;
    Py_INCREF( obj );
    /*
     * 设置 Python 级信号句柄
     */
    Handlers[sig_num].func      = obj;
```

signal_handler() 是 Python 解释器自带的通用型 C 级信号句柄，是那种最原始的单形参信号句柄，不是高大上的三形参信号句柄。所有 Python 级信号句柄只对应这一个 C 级信号句柄，二者不是同步调用关系，只是通过 Handlers[] 全局数组存在异步关系。

```
static void signal_handler ( int sig_num )
{
...
{
trip_signal( sig_num );
}
static void trip_signal ( int sig_num )
{
/*
* 表示 Python 解释器有空时需要调用这个 Python 级信号句柄
*/
Handlers[sig_num].tripped   = 1;
...
Py_AddPendingCall( checksignals_witharg, NULL );
```

signal_handler() 真正干的活就是调用 trip_signal()，后者处理 Handlers[] 全局数组，通知 Python 解释器有 Python 级未决信号待处理。

Py_AddPendingCall() 安排 " 延迟调用 "，Python 解释器会择机调用 checksignals_witharg()，进行 Python 级异步信号处理。

Python 解释器并不会在收到信号时立即调用 Python 级信号句柄。

checksignals_witharg() 最终会去调用 PyErr_CheckSignals(), 后者负责调用 Python 级信号句柄 :

```
int PyErr_CheckSignals ( void )
{
...
for ( i = 1; i < NSIG; i++ )
{
if ( Handlers[i].tripped )
{
...
Handlers[i].tripped = 0;
if ( arglist )
{
/*
* 调用 Python 级信号句柄
*/
result  = PyEval_CallObject( Handlers[i].func, arglist );
```

那什么时候由谁来调用 checksignals_witharg() 呢? 这就必须了解 Python 字节码解释执行流程 :

```
PyObject * PyEval_EvalFrameEx ( PyFrameObject *f,
int throwflag )
{
...
 /*
* 解释执行 PVM 指令的主循环
*/
for ( ; ; )
{
...
if ( --_Py_Ticker < 0 )
{
...
_Py_Ticker  = _Py_CheckInterval;
...
if ( pendingcalls_to_do )
{
/*
* 在此处理 " 延迟调用 "。
 */
if ( Py_MakePendingCalls() < 0 )
{
...
}
/*
* 臭名昭著的全局解释器锁。
```

```
            */
            if ( interpreter_lock )
            {
                ...
                PyThread_release_lock( interpreter_lock );
                PyThread_acquire_lock( interpreter_lock, 1 );
                ...
            }
        }
        ...
        /*
         * 解释执行 PVM 指令的主 switch
         */
        READ_TIMESTAMP( inst0 );
        switch ( opcode )
        {
        ...
        }

        ...
    } /* main loop */
    ...
```

PyEval_EvalFrameEx() 负责解释执行 PVM(Python 虚拟机) 指令，它会在适当时候调用 Py_MakePendingCalls() 处理 " 延迟调用 "，如果存在 Python 级未决信号，就由 Py_MakePendingCalls() 最终调用 checksignals_witharg()。

什么是适当时候? 解释执行 PVM 指令的主循环缺省情况下每执行 100 条 PVM 指令就会来处理一下 " 延迟调用 "。每条 PVM 指令可以认为是原子操作。一条 PVM 指令有可能引发 C 函数调用，后者隶属该条 PVM 指令，属于同一个最小执行单位。100 这个阈值可以通过 sys.setcheckinterval() 调整。

五、Python 级信号句柄失效的原因

下面的伪代码解释了原始问题中 Python 级信号句柄失效的本质原因 :

```
C 级信号句柄 ( 只能在主线程中安装 )
安排 " 延迟调用 "，暗含全局计数器清零
Python 字节码解释器
无限循环
{
if ( 全局计数器为 0，表示已经解释执行了 100 条 PVM 指令 )
{
计数器恢复最大值 ( 缺省 100)
if ( 存在 " 延迟调用 " )
{
主线程处理 " 延迟调用 "
{
```

```
                    /*
                     * 流程需要到这里才有机会执行 on_SIGTERM()。虽然现在计
                     * 数据器已经为 0，但流程永远到不了这里。
                     */
                    主线程调用 Python 级信号句柄
                }
                主动释放 GIL，给其他线程执行的机会
                再次申请 GIL
            }
        }
        else
        {
            /*
             * 流程在此，sem_wait() 在此陷入死锁。SIGTERM 打断系统调用，
             * sem_wait() 返回 EINTR，但外层的 PyThread_acquire_lock()
             * 封装流程又陷入死循环，再次调用 sem_wait()，再次陷入死锁。
             */
            解释执行 1 条 PVM 指令
            全局计数器递减
        }
    }
}
```

Python 级信号句柄就是个摆设，基本无用，还是 C 级信号句柄管事。假设单条 PVM 指令的解释执行流程陷入等待、阻塞、死循环或死锁，而某信号到达无法使之摆脱此状态并结束本条 PVM 指令的解释执行时，Python 级信号句柄永无机会执行，比如 Ctrl-C 失效。由于臭名昭著的 GIL 的存在，Python 级多线程并不能避免这种情况发生。假设非主线程解释执行某条 PVM 指令时流程陷入等待、阻塞、死循环或死锁，而某信号到达无法使之摆脱此状态并结束该条 PVM 指令的解释执行，该线程将无法主动释放 GIL，主线程将永远挂起。而只有主线程才能处理 " 延迟调用 "、执行 Python 级信号句柄。

Python 给 SIGINT 也安装了 Python 级信号句柄，从而取消了 OS 默认的 Term 行为。本来 SIGINT 的 Python 级信号句柄会抛出 KeyboardInterrupt 异常，现在没这机会，所以 Ctrl-C 无法将其中止。SIGINT 的 Python 级信号句柄是 signal.default_int_handler，这是一个 built-in 函数。

如果 Python 源码中针对线程对象调用 join()，在被等待线程存活期间 (终止前) 很容易出现 Python 级信号句柄得不到执行的现象，比如 Ctrl-C 失效。原理同上，join() 在等待，此时那条 PVM 指令一直没结束。

t.join()、cond.wait() 最终都在调 PyThread_acquire_lock()，内里封装 sem_wait()。如果只是 sem_wait() 还没事，因为有

EINTR。但外层的 PyThread_acquire_lock() 封装流程特别处理了 EINTR，在用户态 (相比 SA_RESTART) 人工重启 sem_wait()，局面恶化。

如果用 signal.signal() 安装 Python 级信号句柄，后者只有在 Python 字节码解释执行函数 PyEval_EvalFrameEx() 重获控制权时才有可能被调用，收到信号时 Python 级信号句柄从来都不是被立即执行，假设当前 EIP 位于 C 代码中，比如 built-in 函数或 C 扩展模块，Python 级信号句柄就会被安排 " 延迟调用 "，当前述 C 代码将控制权交还给 PyEval_EvalFrameEx() 时，Python 级信号句柄得到执行机会。这个过程可能很长，完全不可预测。如果用 readline() 进入交互模式，除非你输入点什么并回车，否则你的 Python 级信号句柄绝对不会得到执行，因为 readline() 使流程进入 built-in 函数，直到输入一行内容，控制权才会交还给 PyEval_EvalFrameEx()。

因为 C 级信号句柄 signal_handler() 总是返回，而不是 siglongjmp() 之类的，因此针对同步信号 SIGFPE、SIGSEGV 安装 Python 级信号句柄没有任何意义，该崩还得崩。

关于 Python 的信号处理机制，可以小结成几个问答：

Q: Python 是如何处理信号的，我安装的信号句柄何时得到执行，与 C 编程时的情形一样吗?

A:

Python 解释器自带通用型 C 级信号句柄 signal_handler()，收到信号时该函数立即得到执行。它将 Handlers[sig_num].tripped 置位，调 Py_AddPendingCall() 安排 " 延迟调用 "。Python 字节码解释执行函数 PyEval_EvalFrameEx() 重获控制权时调 Py_MakePendingCalls() 处理 " 延迟调用 ", Python 级信号句柄被执行。C 级的信号句柄永远是 signal_handler()，对于 Python 代码来说，不可更改。

Q: 如果进程正运行在 Python 的 C 扩展模块中，信号到达时会立即转去执行信号句柄吗?

A:

如果问的是 Python 级信号句柄，答案是，不会。要等流程离开 C 扩展模块、回到 Python 字节码解释器中时，才有机会执行 Python 级信号句柄。

顺便说一下 Python 程序中 Ctrl-C 失效的最简解决方案：

```
signal.signal( signal.SIGINT, signal.SIG_DFL )
```

这将导致 OS 默认 Term 行为生效。

六、常用条件断点

执行 built-in 函数 time.sleep() 时断下：

```
b PyCFunction_Call if (0==strcmp(PyString_AsString(((PyCFunctionObject *)func)->m_module),"time") && 0==strcmp(((PyCFunctionObject *)func)->m_ml->ml_name,"sleep"))
```

执行 Python 函数 on_SIGTERM() 时断下：

```
b PyEval_EvalFrameEx if (0==strcmp(PyString_AsString(f->f_code->co_name),"on_SIGTERM"))
```

执行到第 7 行时断下：

```
b ceval.c:1100 if (7==PyFrame_GetLineNumber(f))
```

这种方式可以针对任意行设断，"ceval.c:1100" 对应中 PyEval_EvalFrameEx() 中这行代码：

```
switch ( opcode )
```

七、在 GDB 中获取 Python 字节码

```
$ /usr/bin/python2.7-dbg
```

假设我们没符号，拦截 PyCode_New() 并查看形参 code、name、firstlineno:

```
# gdb -q -ex "b *PyCode_New" -p $(pgrep -f python2.7-dbg)
commands 1
silent
set $code=*(unsigned int *)($esp+0x14)
hexdump $code+0x1c *($code+0x10)
set $name=*(unsigned int *)($esp+0x30)+0x1c
printf "name=[%s]\n",$name
set $firstlineno=*(unsigned int *)($esp+0x34)
printf "firstlineno=[%u]\n",$firstlineno
end
```

在 Python 解释器中定义函数：

```
def foo () :
 print( 'Hello World!' )
```

断点命中，相关命令被执行：

```
B7394E7C 64 01 00 47 48 64 00 00-53                d..GHd..S
name=[foo]
firstlineno=[1]
(gdb) c
Continuing.
B73967C4 64 00 00 84 00 00 5A 00-00 64 01 00 53    d.....Z..d..S
name=[<module>]
firstlineno=[1]
(gdb)
```

"64 01 00 47 48 64 00 00 53" 这种就是 Python 字节码。这里演示的办法可以对付 co_code 成员被藏起来的那些修改过的 Python 解释器。

八、其他

在 GDB 中还可以对 Python 源代码进行热 Patch，主要原理是 PyImport_ReloadModule()。

调用 PyImport_ReloadModule() 时机要选对，比如 py-bt 显示当前栈帧位于 xxx 中，就不能立即 reload xxx，必须 finish 并确保离开 xxx 之后再 reload xxx，否则必将触发 SIGABRT。所以 _main_ 是不能 reload 的。

限于篇幅，这里不再进行具体演示。

九、参考文献

https://wiki.python.org/moin/DebuggingWithGdb

https://docs.python.org/devguide/gdb.html

https://docs.python.org/2/library/signal.html

https://docs.python.org/2/c-api/