WAD:
A Module for Converting Fatal Extension Errors into Python Exceptions

David M. Beazley
Department of Computer Science
University of Chicago
beazley@cs.uchicago.edu

March 6, 2001
Python Extension Building

A popular use of Python

- Hand-written extensions.
- FPIG
- pyfort
- SIP
- BPL
- CXX
- Extension Classes
- GRAD
- SWIG
- (Apologies to anyone I missed)

Extension building is fun

- Python as control language for C, C++, or Fortran.
- Rapid development and prototyping
- Nice user interfaces

But, debugging of extensions is problematic

- At the very least, it’s annoying.
A Python Error

% python spam.py
Traceback (most recent call last):
  File "spam.py", line 15, in ?
    blah()
  File "spam.py", line 12, in blah
    bar()
  File "spam.py", line 9, in bar
    foo()
  File "spam.py", line 6, in foo
    spam(3)
  File "spam.py", line 3, in spam
    doh(n)
NameError: There is no variable named 'doh'
An Extension Error

% python spam.py
Segmentation Fault (core dumped)
%

or

% python spam.py
Bus Error (core dumped)

or

% python spam.py
Assertion failed: n > 0, file debug.c, line 54
Abort (core dumped)
%

Well, obviously something “bad” happened
Common Failure Modes

Uninitialized Data
  • Improper initialization of libraries.
  • Forgetting to call an initialization function?
  • Calling functions in the wrong order?

Improper argument checking
  • Passing of NULL pointers.
  • Improper conversion of Python objects to C.

Failed assertions
  • Library may make extensive use of assert().
  • This is good, but it causes execution to abort.

Weird stuff
  • Illegal instructions.
  • Bus error. Memory alignment problems.

Math errors
  • Floating point exception (SIGFPE).
  • Of course, this only happens after 50 hours of computation.
GDB Traceback

```
(gdb) where
  #0 0xff1d9bf0 in __sigprocmask () from /usr/lib/libthread.so.1
  #1 0xff1ce628 in __resetsig () from /usr/lib/libthread.so.1
  #2 0xff1dcd18 in __sigon () from /usr/lib/libthread.so.1
  #3 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #4 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #5 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #6 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #7 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #8 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #9 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #10 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #11 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #12 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #13 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #14 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #15 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #16 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #17 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #18 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #19 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #20 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #21 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #22 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #23 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #24 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #25 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #26 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #27 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #28 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #29 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
  #30 0x2810e8c in __thr_kill () from /usr/lib/libthread.so.1
```
Debugging Problems

General problem
- Traditional debugger doesn’t know anything about Python scripts.
- Mostly provides information about the implementation of Python.
- Can’t fully answer question of “how did I get here?”
- A problem if you have a lot of Python code.

Sometimes it is hard to reproduce a problem
- Run-time environment may be complex.
- Problems may be due to timing or precise event sequences.
- Problem may only occur after a long period of time.

Other issues
- Requires users to run a separate application (very unpython).
- Requires users to have a C development environment installed.
- Assumes users know how to use the C debugger.

Claim: I think you can do better
WAD

Wrapped Application Debugger

- Idea: Maybe you could turn fatal extension errors into Python Exceptions
- Seg faults, bus errors, illegal instructions, failed assertions, and math errors.

Demo
% python
>>> import debug
>>> debug.seg_crash()
Segmentation fault (core dumped)
%
% python
>>> import debug
>>> import libwadpy
WAD Enabled
>>> debug.seg_crash()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  SegFault: [ C stack trace ]

#2   0x000281c0 in call_builtin(func=0x1cbaf0, arg=0x18f114, kw=0x0) in 'ceval.c', line 2650
#1   0xfeee26b8 in _wrap_seg_crash(self=0x0, args=0x18f114) in 'pydebug.c', line 510
#0   0xfeee1258 in seg_crash(0x1, 0xfeef2d48, 0x19a9f8, 0x0, 0x7365675f, 0x5f5f6469) in 'debug.c', line 18

/u0/beazley/Projects/WAD/WAD/Test/debug.c, line 18

    int seg_crash() {
        int *a = 0;
        => *a = 3;
        return 1;
    }
Big Picture

WAD

- WAD is a dynamically loadable Python extension module.
- Converts catastrophic errors to Python exceptions.

Key features

- No modifications to Python
- No modifications to extensions.
- No recompilation.
- No relinking.
- No separate debugger required (gdb, dbx, etc.)
- No C, C++, Fortran development environment needed.
- No added performance penalty.

The rest of this talk

- Using WAD
- Gory implementation details
- Limitations
- Future directions.
Using WAD

1. Explicit import
   
   import libwadpy

2. Implicit linking
   
   ld -shared $(OBJS) -o foomodule.so -lwadpy

   - Automatically loads WAD when the extension is loaded.

What WAD provides

- 4 new Python exceptions (SegFault, BusError, AbortError, IllegalInstruction)
- Exceptions are added to __builtin__ module.
- A new Python type (WadObject). Returned as an exception value.
- Otherwise, no public functions, constants, or variables (libwadpy is empty).
- Also: WAD is completely self contained
Exception Handling with WAD

Just like ordinary Python exception handling

• Except that you get a much more interesting exception value

```python
try:
    naughtly bits
except SegFault, s:
    t = s.args[0]  # Get trace object
    print t        # Prints stack trace
    len(t)         # Number of stack frames
    f = s[3]       # Returns a stack frame
    f.__FILE__     # Source file
    f.__LINE__     # Source line
    f.__EXE__      # Object file
    f.__PC__       # Program counter
    f.__STACK__    # Raw stack frame
    ...
    f.name         # Value of parameter or local name
```
Implementation Overview

Unix signal handling

- SIGSEGV
- SIGBUS
- SIGABRT
- SIGFPE
- SIGILL

Process introspection

- Discovering program context.
- Reading of object files
- Collection of debugging data

Abort and return to Python

- How do you actually get back to the interpreter?
Control Flow

```python
>>> foo.spam()
```

Python Internals

```
call_builtin()
```

Extension Code
Control Flow (cont)

```python
>>> foo.spam()
```

Python Internals

```python
call_builtin()
```

Extension Code

WAD

1. Examine call stack
2. Collect debugging information.
3. Look for safe place to return.
4. Raise Python exception
5. Abort execution.
Control Flow (cont)

```python
>>> foo.spam()
Traceback (most recent call last):
  File "spam.py", line 22, in foo.spam()
    foo.spam()
SegFault: [ C stack trace ]
```

Extension Code

```
Python Internals
  call_builtin()
```

```c
WAD
1. Examine call stack
2. Collect debugging information.
3. Look for safe place to return.
4. Raise Python exception
5. Abort execution.
```

WAD : 9th International Python Conference, Long Beach, California, March 6, 2001
Signal Handling

Traditional Signal Handling

```c
void seg_handler(int signo) {
    ...
    printf("Aiiee!!!!");
    ...
    return;
}

void foo() {
    ...
    signal(SIGSEGV, seg_handler);
    ...
    naughty bits
    ...
}
```

- Signal handler executes on error.
- Unfortunately, execution resumes at point of error (and repeats).
- Note: Python signal module can't handle SEGSEGV and related signals.
Signal Handling

Advanced Signal Handling

```c
void seg_handler(signo, siginfo, context) {
    printf("Aiiee!!!!");
    ...
    modify context
    return;
}

void foo() {
    ...
    sigaction(SIGSEGV, ...);
    ...
    naughty bits
    ...
}

bar() {
    ...
    nice bits
    ...
}
```

- Rarely used form of sigaction() allows signal handler to modify context
- Includes all CPU registers, program counter (PC), stack pointer (SP)
- Changes take effect on return from signal handler.
- Normally used to implement user-level thread libraries.
WAD: In a Nutshell

Signal handling + context rewriting

- Signal handler collects process information.
- Raise Python exception.
- Rewrite process context so that Python interpreter regains control.
- Return from signal handler.

Issues

- How do you perform process introspection?
- How do you figure out where to return in Python?
- How do you abort execution without breaking the universe?
Finding Program Context

1. Generate raw stack trace
   
   - A very simple while loop.
   - Get sequence of PC, SP values and stack frames.
Finding Program Context

stack trace (PC)                                        process memory map
0x0001f7c4                                               python
0x0001f904
0x0001fecd
0x0003a7c8
0x0003ac1c
0x0003b77c
0x0003b7a8
0x0003b7f8
0x000237e8
0x0002675c
0x0002808c
0x000281c0
0xfeee241c
0xfeee1178
0x000bee2c
0xfeee350e4
0xfeee49b08
0xff1d0e84
0xff1cdd10
0xff1d9bf0

signal -> 0xff1d9bf0

2. Read process memory map from /proc
   - Get base/bounds for Python executable, all shared libraries, heap, stack, etc.
3. Map stack trace to memory map

- Determines the module associated with each stack frame.
- Note: memory map also used to validate the stack trace.
4. Map to symbolic names

- Read ELF symbol table from object files in memory map
- Symbols defined by a simple (name, base, size) triple.
Gathering Debugging Information

Items of interest

- Source filename
- Source line number
- Function parameters (names, values)
- Local variables (names, values).

Debugging information is stored in object files

- If code compiled with -g
- However, debugging data is not loaded into memory during execution.

Collection strategy

- Load all object files found in process memory map.
- Search for debugging data for each symbol in the stack trace.
Gathering Debugging Information

STABS

- Language neutral specification of source information.
- Includes locations, types, functions, parameters, locals, line numbers, etc.
- Decoding is a major head explosion (and that’s all I will say about it).
Final Result

Get a C data structure representing program state

- `name = "spam"`
  - `source = "/u0/beazley/WAD/Test/spam.c"`
  - `line = 42`
  - `args = [ ("n", -1) ]`
  - `stack = < raw stack data >`
  - `next`

- `name = "wrap_spam"
  - `source = "/u0/beazley/WAD/Test/spam_wrap.c"
  - `line = 1782`
  - `args = [ ("self", 0x0), ("args", 0x1782308) ]`
  - `stack = < raw stack data >`
  - `next`

- `name = "call_builtin"
  - `source = "/public/software/Python-2.0/Python/ceval.c"
  - `line = 2650`
  - `args = [ ("func", 0x1cc2d8), ("self", 0x0), ("args, ...) ]`
  - `stack = < raw stack data >`
  - `next`
Returning to Python

Step 1: Examine stack trace for a suitable return point

Call Stack

_start
main
Py_Main
PyRun<AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
_wrap_spam
spam
__eprintf
abort
raise
__thrp_kill
__signon
__sigprocmask

Return Table

<table>
<thead>
<tr>
<th>Function name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>call_builtin</td>
<td>0</td>
</tr>
<tr>
<td>PyObject_GetattrString</td>
<td>0</td>
</tr>
<tr>
<td>PyObject_SetattrString</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Return value = 0 (NULL)

- Table contains Python functions that call ext. code.
- Search looks for first function found on stack.
- Return value used when raising exceptions (e.g., return NULL on error.)
Returning to Python

Step 2: Raise an exception

- If no valid Python return function, print stack trace and exit.
- Otherwise, raise Python exception.
- SegFault, AbortError, BusError, IllegalInstruction, FloatingPointError

Exception value

- Is a special Python type wadObject
- Contains entire stack trace and all data collected.
- Is really just a wrapper around the C data structure described earlier.
- str() and repr() methods simply dump the stack trace as a string.
- Other methods provide access to raw data.

```
try:
    # some naughty extension code
except SegFault, s:
    print "Whoa!"
    print s           # Dump a stack trace
```
Returning to Python

Step 3: Modify process context and return

- Chop off the call stack and return with an error/exception

Call Stack

```
_start
main
Py_Main
PyRun<AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
_wrap_spam
spam
__eprintf
abort
raise
__thrpu_kill
__sigprocmask
```

```
_start
main
Py_Main
PyRun<AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
```

return from signal

```
NULL, AbortError
```
A "Slight" Complication

Return mechanism is similar to:

- setjmp/longjmp in C
- C++ exception handling.

However...

- Python is not instrumented or modified in any way.
- There is no corresponding setjmp() call.
- There is no matching try { ... } clause in C++.

This means...

- We are returning to some "arbitrary" location in the Python executable.
- Never designed with such a non-local procedure return in mind.

This is a bit of a problem:

- Corrupted CPU registers.
The Register Save Problem

```c
foo() {
    ...
    call bar
    ...
}
bar() {
    ...
    do stuff
    ...
}
```

CPU Registers

Call Stack

- foo locals

Each procedure uses CPU registers.
- Temporaries, local variables, memory addressing, etc.
Register Save (cont)

```c
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    ...
    do stuff
    ...
}
```

**Caller-save**

- Must save certain registers before calling a new procedure.
- Restore after procedure returns.
Register Save (cont)

```c
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    save_callee_regs
    ...
    do stuff
    ...
    restore_callee_regs
}
```

**Callee-save**

- Procedures save registers they plan to overwrite.
- Restore values prior to returning
Register Save (cont)

```plaintext
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    save_callee_regs
    ...
    do stuff
    ...
    restore_callee_regs
}
```

Procedure return

- Callee restores registers
- Caller restores registers
Register Save (cont)

```python
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    save_callee_regs
    ...
    naughty bits
    ...
    restore_callee_regs
}
```

Aborted return

- Callee-saved register values are lost (never restored)
- Corrupts CPU state in caller on return (this is usually bad)
Register Restoration

Solution: SPARC

- Each procedure gets a fresh set of CPU registers (i.e., a “window”)
- To restore state: simply roll back the register windows

Solution: i386

- Manually inspect machine code of function prologues
- Figure out where callee-save registers are saved on call-stack
- Restore values while walking up the call stack.

```
blah:
55  pushl  %ebp
89 e5  movl  %esp,%ebp
83 ec 2c  subl  $0x2c,%esp
57  pushl  %edi
56  pushl  %esi
53  pushl  %ebx
```

- Only a heuristic. Might get it wrong, but the return to Python may still work.
- Not as bad as it sounds---implementation is fairly simple.
The Auto-Initialization Hack

One final bit...

- How do you get WAD to initialize itself when linked to extensions?

```cpp
class WadInit {
public:
    WadInit() {
        wad_initialize();
    }
};
static WadInit winit;
```

- Dynamic link/loader automatically invokes C++ static constructors on import.
- Constructors are invoked before any extension code executes.
Implementation Details

Implementation

- Mostly ANSI C, some assembly, some C++
- ~1500 semicolons
- Most code related to introspection (debugging, symbol tables, etc...)
- Core is Python independent (only 166 semicolons related to Python).
- Execution is isolated (own stack and memory management).
- Does not rely upon third party libraries (e.g., libbfd).

Compatibility

- Sun Sparc Solaris
- i386 Linux (recent kernels).
- Python 1.5 and newer (class based exceptions)
- Miscellaneous compatibility issues on Linux.
- Also supports Tcl.
Limitations

Non-local return, aborted execution

- May leak memory
- No destruction of objects in C++.
- May interact poorly with C++ exceptions.
- May result in unreleased system resources (files, sockets, etc.).
- May result in deadlock (if holding locks when error occurs).

Unrecoverable errors

- Extensions that destroy or corrupt Python interpreter data.
- Stack overflow (results in corrupted call-stack).

Compiler optimization

- False reporting of debugging data, source files, and lines.
- Incorrect register recovery (-fomit-frame-pointer)

Compatibility

- Mixing threads and signals is extremely problematic.
- WAD requires fully functional signal implementation.
- Some versions of Linux, Linux+Threads do not work.
More Limitations

Debugging information

- Only simple datatypes are currently understood.
- No special C++ support (classes, name demangling, etc.)
- No understanding of structures.

Things that just don’t work

- Breakpoints
- Single-step execution.
- Restart
Related Work

Surprisingly little literature on this topic

- PyDebug.
- Programming environments for Common Lisp (FFI).
- Asynchronous exception handling (ML, Haskell)
- Rn (A mixed interpreter-compiled system for Fortran)
- Modifications to gdb for debugging Common Lisp (WCL).
- Java mixed-mode debugging (Java + JNI).  ???
- Perl (sigtrap module can print perl stack trace on fatal error).
Future Directions

Better error recovery and data reporting
  • Make the WAD core as generic as possible.
  • Better heuristics for certain errors (corrupted call stack, corrupted heap).
  • Improved collection of debugging information.

Support for more platforms
  • Obviously. Maybe. Not.

Integration with Python debugger, IDEs?
  • Demo.

Other languages
  • Tcl, Ruby, Perl, etc. (Tcl works now).

Bizarre execution modes?
  • Restarts?
  • Breakpoints?
  • Code patching?
Conclusions

Extension programming

- A lot of people are building extensions.
- Debugging has always been a little annoying.

Conventional wisdom

- Modify an existing debugger to understand Python.
- Why reinvent the wheel (especially debuggers)?

Why not reevaluate the situation?

- Traditional debugging model is awkward for extension programming.
- Exception handling approach is cool and fits in nicely with Python scripts.
- Simply knowing where code crashed is enough to fix a lot of bugs.
- The exception approach is also nice when distributing extensions.

Bottom line: WAD is mostly a proof of concept

- Common extension errors can be handled within Python.
- Can extend Python exception handling to compiled extensions.
More Information

http://systems.cs.uchicago.edu/wad

• This is work in progress.
• Not ready for prime time yet.
• Many related problems to work on.
• Volunteers welcome.
• I’m also looking for students.