WinAppDbg Documentation

Release 1.4

Mario Vilas

August 22, 2010

CONTENTS

The *WinAppDbg* python module allows developers to quickly code instrumentation scripts in **Python** under a **Windows** environment.

It uses **ctypes** to wrap many Win32 API calls related to debugging, and provides a powerful abstraction layer to manipulate threads, libraries and processes, attach your script as a debugger, trace execution, hook API calls, handle events in your debugee and set breakpoints of different kinds (code, hardware and memory). Additionally it has no native code at all, making it easier to maintain or modify than other debuggers on Windows.

The intended audience are QA engineers and software security auditors wishing to test or fuzz Windows applications with quickly coded Python scripts. Several *ready to use utilities* are shipped and can be used for this purposes.

Current features also include disassembling x86 native code (using the diStorm disassembler), debugging multiple processes simultaneously and produce a detailed log of application crashes, useful for fuzzing and automated testing.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting started

This is what you need to know to download, install and begin to use *WinAppDbg*:

1.1.1 Download

The current version is **1.4**. You can choose **any** of the following files (if in doubt, pick the **first**):

Windows (32 bits)

- winappdbg-1.4.win32.exe
- winappdbg-1.4.win32.msi

Windows (64 bits)

- winappdbg-1.4.win-amd64.exe
- winappdbg-1.4.win-amd64.msi

Source code

- winappdbg-1.4.zip
- winappdbg-1.4.tar.bz2

The Sourceforge project's download page contains all versions. You can also get the bleeding-edge version as a source code tarball from the subversion repository.

1.1.2 Install

Simply run the Windows installer package and follow the wizard.

Alternatively, if you prefer using EasyInstall (setuptools), type the following at the command prompt

```
easy_install winappdbg
```

And *WinAppDbg* will be automatically downloaded and installed from the PyPI repository.

If you prefer to install directly from the sources package, extract it to any temporary folder and run the following command

```
setup.py install
```

1.1.3 Dependencies

Naturally you need the Python interpreter. There are two basic flavors, just pick your favorite:

- The **official** Python interpreter (free, open source). This is the preferred choice.
- ActiveState ActivePython (free, closed source). It should work but in 64 bit Windows the *ctypes* module is missing and you'll have to install it manually.

If you're still using Python 2.4 you'll need to install some additional modules:

- The ctypes module is needed to interface with the Win32 API.
- The SQLite python bindings can be used with the crash logger tool to store the crash information in an SQLite database file.

The *diStorm <http://code.google.com/p/distorm/>* disassembler is also required. You can download the official Python wrappers (32 bits only, manual install) or our own installers. Bear in mind that the official build is more likely to stay up to date.

Note: If you don't install diStorm, all classes and methods of the debugger not related to dissassembling will still work correctly.

Optional packages

The following packages provide extra features and performance improvements, but they're not required to use *WinAppDbg*:

- The PyODBC module gives the crach logger tool the ability to connect to MSSQL databases.
- The Python specializing compiler, Psyco. *WinAppDbg* will experience a performance gain just by installing it, no additional steps are needed. You can download it from here.
- PyReadline is useful when using the console tools shipped with *WinAppDbg*, but they'll work without it. Basically what it does is provide autocomplete and history for console applications.
- The py2exe package. You can use it to generate standalone binaries for any tools made with *WinAppDbg*. See the instructions on how to use the *Makefile*.

1.1.4 Support

This package has been tested under **Windows XP** and above (both 32 and 64 bits) using **Python 2.6**. It was loosely tested under *Windows 2000*, *Wine* and *ReactOS*, and some bugs are to be expected in these platforms (mainly due to missing APIs).

If you find a bug or have a feature suggestion, don't hesitate to send an email to the [https://lists.sourceforge.net/lists/listinfo/winappdbg-users winappdbg-users] mailing list. Both comments and complaints are welcome! :)

The following tables show which Python interpreters, operating systems and processor architectures are currently supported. **Full** means all features are fully functional. **Partial** means some features may be broken and/or untested.

Experimental means there is a subversion branch with at least partial support, but hasn't been merged to trunk yet. **Untested** means that though no testing was performed it should probably work.

- Python interpreters

Python 2.4	**full**	
Python 2.5	**full**	
Python 2.6	**full**	
Python 2.7	**full**	
Python 3.x	*experimental*	(see this branch)

- Operating systems

Windows XP	**full**	
Windows Vista	**full**	
Windows 7	**full**	
Windows Server 2003	**full**	
Windows Server 2003 R2	**full**	
Windows Server 2008	**full**	
Windows Server 2008 R2	**full**	
Windows 2000 and older	*partial*	(some Win32 APIs didn't exist yet)
ReactOS	*untested*	(probably similar to Windows 2000)
Linux (using Wine)	*untested*	(reported to work on Ubuntu)

- Architectures

Intel x86 (32 bits) and compatible	**full**	
Intel x86_x64 (64 bits) and compatible	*partial*	(function hooks are not implemented)
Intel IA64 (Itanium)	*experimental*	(no actual Itanium system to test it on, help is needed!)

1.1.5 License

This package is released under the BSD license, so as a user you are entitled to create derivative work and *redistribute* it if you wish. A makefile is provided to automatically generate the source distribution package and the Windows installer, and can also generate the documentation for all the modules using Epydoc. The sources to this documentation are also provided and can be compiled with Sphinx.

1.2 Tools

The *WinAppDbg* package comes with a collection of tools useful for common tasks when debugging or fuzzing a program. The most important tool, the *Crash logger*, attaches to any number of target processes and collects crash dump information in a SQLite database. It can also apply *heuristics* to discard multiple occurrences of the same crash.

The source code of these tools can also be read for more examples on programming using *WinAppDbg*.

The following tools are shipped with the *WinAppDbg* package:

1.2.1 Crash logger

- `crash_logger.py`:


crash_logger.py -vc ..\misc\crasher.exe 1
[14:29:56.0950] Crash logger started, Wed Apr 22 14:29:56 2009
[14:29:57.0081] pid 4068 tid 2284: Process C:\Documents and Settings\Mario Vilas
\Desktop\winappdbg\misc\crasher.exe started, entry point at 0x00401220
[14:29:57.0091] pid 4068 tid 2284: Loaded ntdll.dll at 0x7c900000
[14:29:57.0111] pid 4068 tid 2284: Loaded C:\WINDOWS\system32\kernel32.dll at 0x
7c800000
[14:29:57.0111] pid 4068 tid 2284: Loaded C:\WINDOWS\system32\msvcrt.dll at 0x77
c10000
[14:29:57.0121] pid 4068 tid 2284: System breakpoint hit
[14:29:57.0151] pid 4068 tid 2284: Access violation (first chance) at crasher.ex
e!0x1326
Registers:
eax=00000000 ebx=00004000 ecx=ffffffff edx=00000031 esi=00000000 edi=00000000
eip=00401326 esp=0022fef0 ebp=0022ff78 iopl=0 no up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00010246
Code disassembly:
0x00401316 ¦ 46 ¦ inc esi
0x00401317 ¦ e9 eb000000 ¦ jmp 0x401407
0x0040131c ¦ c745 f4 00000000 ¦ mov dword [ebp-0xc], 0x0
0x00401323 ¦ 8b45 f4 ¦ mov eax, [ebp-0xc]
* 0x00401326 ¦ c600 61 ¦ mov byte [eax], 0x61
0x00401329 ¦ e9 d9000000 ¦ jmp 0x401407
0x0040132e ¦ c70424 67304000 ¦ mov dword [esp], 0x403067
0x00401335 ¦ e8 ¦ db 0xe8
Stack pointers:
[esp+0x00] -> 31 00 ab ab ab ab ab ab ab ee fe ee fe 00 00 1.....
[esp+0x04] -> e0 ff 22 00 94 5c c3 77 70 20 c1 77 ff ff ff ff ..".....\.wp..w....
[esp+0x08] -> d0 2c 3e 00 20 2d 3e 00 78 2d 3e 00 c8 2d 3e 00 .,>..->.x->..->.
[esp+0x0c] -> e8 30 02 00 00 83 7d 08 01 75 11 e8 b5 ff ff ff .0....}..u.....
[esp+0x14] -> c2 0c 00 ff ff ff ff 45 d5 93 7c 4e d5 93 7c ff .....E..¦N..¦.
[esp+0x18] -> e8 78 b1 00 00 c3 cc cc cc cc 6a 10 68 88 20 .x.....j.h..
[esp+0x1c] -> c8 00 00 00 3a 01 00 00 ff ee ff ee 62 10 00 50 ....:.....b..P
[esp+0x24] -> c3 cc cc cc cc cc 6a 10 68 88 20 c1 77 e8 2b b1 ......j.h...w.+.
[esp+0x30] -> 2e 2e 5c 6d 69 73 63 5c 63 72 61 73 68 65 72 2e ..\misc\crasher.
[esp+0x32] -> 75 00 6d 00 65 00 6e 00 74 00 73 00 20 00 61 00 u.m.e.n.t.s...a.
[esp+0x38] -> ff ff ff ff 00 00 00 00 39 c3 c2 77 ff ff ff ff ....9..w....
Stack dump:
0022fef0: e2 3d 3e 00 3c ff 22 00 38 2c 3e 00 cb 12 40 00 .=>.<."8,>...@.
0022ff00: ff ff ff ff 5d 00 91 7c de c2 c2 77 00 00 3e 00 ....]..¦...w..>.
0022ff10: 00 00 00 00 e3 c2 c2 77 00 00 00 00 10 00 00 00 ....w........
0022ff20: e0 1e 24 00 01 00 00 00 88 20 c1 77 ff ff ff ff ..$.........w....
0022ff30: ce c3 c2 77 18 ff 22 00 0c 00 00 00 e0 ff 22 00 ...w.."........".
0022ff40: 94 5c c3 77 70 20 c1 77 ff ff ff ff e3 c2 c2 77 .\.wp..w.......w
0022ff50: 14 b8 c3 77 20 3e 3e 00 c0 3d 3e 00 0c 00 00 00 ...w.>>..=>.....
0022ff60: ad ae c3 77 00 00 00 00 00 00 00 00 00 00 00 00 ...w............

Attaches as a debugger or starts a new process for debugging. Whenever an interesting debug event occurs (i.e. a bug is found) it can save the info to a database and/or log it through standard output.

Some simple *heuristics* can be used to try to determine whether two crashes were caused by the same bug, in order to discard duplicates. It can also try to guess how exploitable would the found crashes be, using similar heuristics to those of !exploitable.

Additional features allow setting breakpoints at the target process(es), attaching to spawned child processes, restarting crashed processes, and running a custom command when a crash is found.

- `crash_report.py`:

  Shows the contents of the crashes database file to standard output.

- `crash_report_mssql.py`:

  Shows the contents of the crashes MS SQL database to standard output.

1.2.2 Process tools

These tools were inspired by the **ptools** suite by Nicolás Economou.

- `pinject.py`:

  Forces a process to load a DLL library of your choice.

- `plist.py`:

  Shows a list of all currently running processes.

- `pmap.py`:

  Shows a map of a process memory space.

- `pfind.py`:

  Finds the given text, binary data, binary pattern or regular expression in a process memory space.

- `pread.py`:

  Reads the memory contents of a process to standard output or any file of your choice.

- `pwrite.py`:

  Writes to the memory of a process from the command line or any file of your choice.

- `pkill.py`:

  Terminates a process or a batch of processes.

- `ptrace.py`:

  Traces execution of a process. It supports three methods: single stepping, single stepping on branches, and native syscall hooking.

- `pdebug.py`:

  Extremely simple command line debugger. It's main feature is being written entirely in Python, so it's easy to modify or write plugins for it.

1.2.3 Miscellaneous

- `SelectMyParent.py`:

  Allows you to create a new process specifying any other process as it's parent, and inherit it's handles. See the blog post by Didier Stevens for the original C version.

- `hexdump.py`:

  Shows an hexadecimal dump of the contents of a file.

1.3 Programming guide

This guide will show you through the most commonly used classes and methods of the *WinAppDbg* module, and provide some examples of use for each one. The goal is to give you a bird's eye perspective on what the library can do and how, without having to go through the reference material.

1.3.1 Instrumentation

You can implement process instrumentation in your Python scripts by using the provided set of classes: *System*, *Process*, *Thread* and *Module*. Each one acts as a snapshot of the processes, threads and DLL modules in the system.

A *System* object is a snapshot of all running processes. It contains *Process* objects, which in turn are snapshots of processes. A *Process* object contains *Thread* and *Module* objects.

Note: You don't need to be attached as a debugger for these classes to work.

The System class

The *System* class basically behaves like a snapshot of the running processes. It can enumerate processes and perform operations on a batch of processes.

Example #1: enumerating running processes

Download

```
from winappdbg import System

# Request debugging privileges for the current process
# This is needed to get some information from services
# (Try commenting out this line to see what happens!)
System.request_debug_privileges()

# Create a system snaphot
system = System()

# Now we can enumerate the running processes
for process in system:
    print "%d:\t%s" % ( process.get_pid(), process.get_filename() )
```

Example #2: starting a new process

Download

```
from winappdbg import System

import sys

# Instance a System object
system = System()

# Get the target application
command_line = system.argv_to_cmdline( sys.argv[ 1 : ] )

# Start a new process
system.start_process( command_line )    # see the docs for more options
```

The Process class

The *Process* class lets you manipulate any process in the system. You can get a *Process* instance by enumerating a *System* snapshot, or instancing one directly by providing the process ID.

A *Process* object allows you to manipulate the process memory (read, write, allocate and free operations), create new threads in the process, and more. It also acts as a snapshot of it's threads and DLL modules.

Example #3: enumerating threads and DLL modules in a process

Download

```
from winappdbg import Process, HexDump

def print_threads_and_modules( pid ):

    # Instance a Process object
    process = Process( pid )
    print "Process %d" % process.get_pid()

    # Now we can enumerate the threads in the process...
    print "Threads:"
    for thread in process.iter_threads():
        print "\t%d" % thread.get_tid()

    # ...and the modules in the process
    print "Modules:"
    for module in process.iter_modules():
        print "\t%s\t%s" % ( HexDump.address( module.get_base() ), module.get_filename() )
```

Example #4: killing a process

Download

```
from winappdbg import Process

def process_kill( pid ):
```

```
    # Instance a Process object
    process = Process( pid )

    # Kill the process
    process.kill()
```

Example #5: reading the process memory

Download

```
from winappdbg import Process

def process_read( pid, address, length ):

    # Instance a Process object
    process = Process( pid )

    # Read the process memory
    data = process.read( address, length )

    # Return a Python string with the memory contents
    return data
```

Example #6: loading a DLL into the process

Download

```
from winappdbg import Process

def load_dll( pid, filename ):

    # Instance a Process object
    process = Process( pid )

    # Load the DLL library in the process
    process.inject_dll( filename )
```

Example #7: getting the process memory map

Download

```
from winappdbg import win32, Process, HexDump

def print_memory_map( pid ):

    # Instance a Process object
    process = Process( pid )

    # Get the process memory map
    memoryMap = process.get_memory_map()

    # Now you could do this...
    #
```

```
    #   from winappdbg import CrashDump
    #   print CrashDump.dump_memory_map( memoryMap ),
    #
    # ...but let's do it the hard way:

    # For each memory block in the map...
    print "Address   \tSize      \tState     \tAccess    \tType"
    for mbi in memoryMap:

        # Address and size of memory block
        BaseAddress = HexDump.address(mbi.BaseAddress)
        RegionSize  = HexDump.address(mbi.RegionSize)

        # State (free or allocated)
        if   mbi.State == win32.MEM_RESERVE:
            State   = "Reserved  "
        elif mbi.State == win32.MEM_COMMIT:
            State   = "Commited  "
        elif mbi.State == win32.MEM_FREE:
            State   = "Free      "
        else:
            State   = "Unknown   "

        # Page protection bits (R/W/X/G)
        if mbi.State != win32.MEM_COMMIT:
            Protect = "          "
        else:
##            Protect = "0x%.08x" % mbi.Protect
            if   mbi.Protect & win32.PAGE_NOACCESS:
                Protect = "--- "
            elif mbi.Protect & win32.PAGE_READONLY:
                Protect = "R-- "
            elif mbi.Protect & win32.PAGE_READWRITE:
                Protect = "RW- "
            elif mbi.Protect & win32.PAGE_WRITECOPY:
                Protect = "RC- "
            elif mbi.Protect & win32.PAGE_EXECUTE:
                Protect = "--X "
            elif mbi.Protect & win32.PAGE_EXECUTE_READ:
                Protect = "R-X "
            elif mbi.Protect & win32.PAGE_EXECUTE_READWRITE:
                Protect = "RWX "
            elif mbi.Protect & win32.PAGE_EXECUTE_WRITECOPY:
                Protect = "RCX "
            else:
                Protect = "??? "
            if   mbi.Protect & win32.PAGE_GUARD:
                Protect += "G"
            else:
                Protect += "-"
            if   mbi.Protect & win32.PAGE_NOCACHE:
                Protect += "N"
            else:
                Protect += "-"
            if   mbi.Protect & win32.PAGE_WRITECOMBINE:
                Protect += "W"
            else:
                Protect += "-"
```

```
            Protect += "    "

        # Type (file mapping, executable image, or private memory)
        if   mbi.Type == win32.MEM_IMAGE:
            Type    = "Image     "
        elif mbi.Type == win32.MEM_MAPPED:
            Type    = "Mapped    "
        elif mbi.Type == win32.MEM_PRIVATE:
            Type    = "Private   "
        elif mbi.Type == 0:
            Type    = "Free      "
        else:
            Type    = "Unknown   "

        # Print the memory block information
        fmt = "%s\t%s\t%s\t%s\t%s"
        print fmt % ( BaseAddress, RegionSize, State, Protect, Type )
```

The Thread class

A *Thread* object lets you manipulate any thread in any process in the system. You can get a *Thread* instance by enumerating a *Process* snapshot, or instancing one manually by providing the thread ID.

You can manipulate the thread context (read and write to it's registers), perform typical debugger operations (getting stack traces, etc), suspend and resume execution, and more.

Example #8: freeze all threads in a process

Download

```
from winappdbg import Process, System

def freeze_threads( pid ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Process object
    process = Process( pid )

    # This would also do the trick...
    #
    #   process.suspend()
    #
    # ...but let's do it the hard way:

    # Lookup the threads in the process
    process.scan_threads()

    # For each thread in the process...
    for thread in process:

        # Suspend the thread execution
        thread.suspend()

def unfreeze_threads( pid ):
```

```
    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Process object
    process = Process( pid )

    # This would also do the trick...
    #
    #   process.resume()
    #
    # ...but let's do it the hard way:

    # Lookup the threads in the process
    process.scan_threads()

    # For each thread in the process...
    for thread in process:

        # Resume the thread execution
        thread.resume()
```

Example #9: print a thread's context

Download

```
from winappdbg import Thread, CrashDump, System

def print_thread_context( tid ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Thread object
    thread = Thread( tid )

    # Suspend the thread execution
    thread.suspend()

    # Get the thread context
    try:
        context = thread.get_context()

    # Resume the thread execution
    finally:
        thread.resume()

    # Display the thread context
    print
    print CrashDump.dump_registers( context ),
```

Example #10: print a thread's code disassembly

Download

```
from winappdbg import Thread, CrashDump, System

def print_thread_disassembly( tid ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Thread object
    thread = Thread( tid )

    # Suspend the thread execution
    thread.suspend()

    # Get the thread's currently running code
    try:
        eip  = thread.get_pc()
        code = thread.disassemble_around( eip )

        # You can also do this:
        # code = thread.disassemble_around_pc()

        # Or even this:
        # process = thread.get_process()
        # code    = process.disassemble_around(eip)

    # Resume the thread execution
    finally:
        thread.resume()

    # Display the thread context
    print
    print CrashDump.dump_code( code, eip ),
```

The Module class

A *Module* object lets you manipulate any thread in any process in the system. You can get a *Module* instance by enumerating a *Process* snapshot. *Module* objects can be used to resolve the addresses of exported functions in the process address space.

Example #11: resolve an API function in a process

Download

```
from winappdbg import Process, System

def print_api_address( pid, modName, procName ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Process object
    process = Process( pid )

    # Lookup it's modules
    process.scan_modules()
```

```
    # Get the module
    module = process.get_module_by_name( modName )
    if not module:
        print "Module not found: %s" % modName
        return

    # Resolve the requested API function address
    address = module.resolve( procName )

    # Print the address
    if address:
        print "%s!%s == 0x%.08x" % ( modName, procName, address )
    else:
        print "Could not resolve %s in module %s" % (procName, modName)
```

1.3.2 Debugging

Debugging operations are performed by the *Debug* class. You can receive notification of debugging events by passing a custom event handler to the *Debug* object when creating it - each event is represented by an *Event* object. Custom event handlers can also be subclasses of the *EventHandler* class.

Debug objects can also set *breakpoints, watches and hooks* and support the use of *labels*.

The Debug class

A *Debug* object provides methods to launch new processes, attach to and detach from existing processes, and manage breakpoints. It also contains a *System* snapshot to instrument debugged processes - this snapshot is updated automatically for processes being debugged.

Example #1: starting a new process and waiting for it to finish

Download

```
from winappdbg import Debug

import sys

# Instance a Debug object
debug = Debug()
try:

    # Start a new process for debugging
    debug.execv( sys.argv[ 1 : ] )

    # Wait for the debugee to finish
    debug.loop()

# Stop the debugger
finally:
    debug.stop()
```

Example #2: attaching to a process and waiting for it to finish

Download

```
from winappdbg import Debug

import sys

# Get the process ID from the command line
pid = int( sys.argv[1] )

# Instance a Debug object
debug = Debug()
try:

    # Attach to a running process
    debug.attach( pid )

    # Wait for the debugee to finish
    debug.loop()

# Stop the debugger
finally:
    debug.stop()
```

Example #3: attaching to a process by filename

Download

```
from winappdbg import Debug

import sys

# Get the process filename from the command line
filename = sys.argv[1]

# Instance a Debug object
debug = Debug()
try:

    # Lookup the currently running processes
    debug.system.scan_processes()

    # For all processes that match the requested filename...
    for ( process, name ) in debug.system.find_processes_by_filename( filename ):
        print process.get_pid(), name

        # Attach to the process
        debug.attach( process.get_pid() )

    # Wait for all the debugees to finish
    debug.loop()

# Stop the debugger
finally:
    debug.stop()
```

Example #4: killing a process by attaching to it

Download

```
from winappdbg import Debug

import sys
import thread

# Get the process ID from the command line
pid = int( sys.argv[1] )

# Instance a Debug object, set the kill on exit property to True
debug = Debug( bKillOnExit = True )

# Attach to a running process
debug.attach( pid )

# Exit the current thread, killing the attached process
thread.exit()
```

The Event class

So far we have seen how to attach to or start processes. But a debugger also needs to react to events that happen in the debugee, and this is done by passing a callback function as the **eventHandler** parameter when instancing the *Debug* object. This callback, when called, will receive as parameter an **Event** object which describes the event and contains a reference to the *Debug* object itself.

Example #5: handling debug events

Download

```
from winappdbg import Debug, HexDump

def my_event_handler( event ):

    # Get the event name
    name = event.get_event_name()

    # Get the event code
    code = event.get_event_code()

    # Get the process ID where the event occured
    pid = event.get_pid()

    # Get the thread ID where the event occured
    tid = event.get_tid()

    # Get the value of EIP at the thread
    pc = event.get_thread().get_pc()

    # Show something to the user
    format_string = "%s (%s) at address %s, process %d, thread %d"
    message = format_string % ( name, HexDump.integer(code), HexDump.address(pc), pid, tid )
    print message
```

```
def simple_debugger( argv ):

    # Instance a Debug object, passing it the event handler callback
    debug = Debug( my_event_handler )
    try:

        # Start a new process for debugging
        debug.execv( argv )

        # Wait for the debugee to finish
        debug.loop()

    # Stop the debugger
    finally:
        debug.stop()
```

The EventHandler class

Using a callback function is not very flexible when your code is too large. For that reason, the **EventHandler** class is provided.

Instead of a function, you can define a subclass of *EventHandler* where each method of your class should match an event - for example, to receive notification on new DLL libraries being loaded, define the *load_dll* method in your class. If you don't want to receive notifications on a specific event, simply don't define the method in your class.

These are the most important event notification methods:

Notification name	*What does it mean?*	*When is it received*
create_process	The debugger has attached to a new process.	When attaching to a process, when starting a new process for debugging, or when the debugee starts a new process and the *bFollow* flag was set to *True*.
exit_process	A debugee process has finished executing.	When a process terminates by itself or when the *Process.kill* method is called.
create_thread	A debugee process has started a new thread.	When the process creates a new thread or when the _Process.start_thread_ method is called.
exit_thread	A thread in a debugee process has finished executing.	When a thread terminates by itself or when the *Thread.kill* method is called.
load_dll	A module in a debugee process was loaded.	When a process loads a DLL module by itself or when the *Process.inject_dll* method is called.
unload_dll	A module in a debugee process was unloaded.	When a process unloads a DLL module by itself.
exception	An exception was raised by the debugee.	When a hardware fault is triggered or when the process calls RaiseException().

The event handler can also receive notifications for specific exceptions as a different event. When you define the method for that exception, it takes precedence over the more generic *exception* method.

These are the most important exception notification methods:

Notification name	*What does it mean?*	*When is it received*
breakpoint	A breakpoint exception was raised by the debugee.	When a hardware fault is triggered by the int3 opcode, when the process calls DebugBreak(), or when a code breakpoint set by your program is triggered.
single_step	A single step exception was raised by the debugee.	When a hardware fault is triggered by the trap flag or the icebp opcode, or when a hardware breakpoint set by your program is triggered.
guard_page	A guard page exception was raised by the debugee.	When a guard page is hit or when a page breakpoint set by your program is triggered.

In addition to all this, the *EventHandler* class provides a simple method for API hooking: the **apiHooks** class property. This property is a dictionary of tuples, specifying which API calls to hook on what DLL libraries, and how many parameter does each call take. That's it! The *EventHandler* class will automatically hooks this APIs for you when the corresponding library is loaded, and a method of your subclass will be called when entering and leaving the API function.

Example #6: tracing execution

Download

```
from winappdbg import Debug, EventHandler, HexDump, CrashDump, win32


class MyEventHandler( EventHandler ):


    # Create process events go here
    def create_process( self, event ):

        # Start tracing the main thread
        event.debug.start_tracing( event.get_tid() )


    # Create thread events go here
    def create_thread( self, event ):

        # Start tracing the new thread
        event.debug.start_tracing( event.get_tid() )


    # Single step events go here
    def single_step( self, event ):

        # Show the user where we're running
        thread = event.get_thread()
        pc     = thread.get_pc()
        code   = thread.disassemble( pc, 0x10 ) [0]
        print "%s: %s" % ( HexDump.address(code[0]), code[2].lower() )


def simple_debugger( argv ):

    # Instance a Debug object, passing it the MyEventHandler instance
```

```
    debug = Debug( MyEventHandler() )
    try:

        # Start a new process for debugging
        debug.execv( argv )

        # Wait for the debugee to finish
        debug.loop()

    # Stop the debugger
    finally:
        debug.stop()
```

Example #7: intercepting API calls

Download

```
class MyEventHandler( EventHandler ):


    # Here we set which API calls we want to intercept
    apiHooks = {

        # Hooks for the kernel32 library
        'kernel32.dll' : [
                            #  Function            Parameters
                            ( 'CreateFileA'     ,    7  ),
                            ( 'CreateFileW'     ,    7  ),
                          ],

        # Hooks for the advapi32 library
        'advapi32.dll' : [
                            #  Function            Parameters
                            ( 'RegCreateKeyExA' ,    9  ),
                            ( 'RegCreateKeyExW' ,    9  ),
                          ],
    }


    # Now we can simply define a method for each hooked API.
    # Methods beginning with "pre_" are called when entering the API,
    # and methods beginning with "post_" when returning from the API.


    def pre_CreateFileA( self, event, ra, lpFileName, dwDesiredAccess,
              dwShareMode, lpSecurityAttributes, dwCreationDisposition,
                                   dwFlagsAndAttributes, hTemplateFile ):

        self.__print_opening_ansi( event, "file", lpFileName )

    def pre_CreateFileW( self, event, ra, lpFileName, dwDesiredAccess,
              dwShareMode, lpSecurityAttributes, dwCreationDisposition,
                                   dwFlagsAndAttributes, hTemplateFile ):

        self.__print_opening_unicode( event, "file", lpFileName )

    def pre_RegCreateKeyExA( self, event, ra, hKey, lpSubKey, Reserved,
```

```
                                    lpClass, dwOptions, samDesired,
                                    lpSecurityAttributes, phkResult,
                                                lpdwDisposition ):

        self.__print_opening_ansi( event, "key", lpSubKey )

    def pre_RegCreateKeyExW( self, event, ra, hKey, lpSubKey, Reserved,
                                    lpClass, dwOptions, samDesired,
                                    lpSecurityAttributes, phkResult,
                                                lpdwDisposition ):

        self.__print_opening_unicode( event, "key", lpSubKey )


    def post_CreateFileA( self, event, retval ):
        self.__print_success( event, retval )

    def post_CreateFileW( self, event, retval ):
        self.__print_success( event, retval )

    def post_RegCreateKeyExA( self, event, retval ):
        self.__print_success( event, retval )

    def post_RegCreateKeyExW( self, event, retval ):
        self.__print_success( event, retval )


    # Some helper private methods...

    def __print_opening_ansi( self, event, tag, pointer ):
        string = event.get_process().peek_string( pointer )
        tid    = event.get_tid()
        print  "%d: Opening %s: %s" % (tid, tag, string)

    def __print_opening_unicode( self, event, tag, pointer ):
        string = event.get_process().peek_string( pointer, fUnicode = True )
        tid    = event.get_tid()
        print  "%d: Opening %s: %s" % (tid, tag, string)

    def __print_success( self, event, retval ):
        tid = event.get_tid()
        if retval:
            print "%d: Success: %x" % (tid, retval)
        else:
            print "%d: Failed!" % tid
```

Breakpoints, watches and hooks

A *Debug* object provides a small set of methods to set breakpoints, watches and hooks. These methods in turn use an underlying, more sophisticated interface that is described at the wiki page HowBreakpointsWork.

The **break_at** method sets a code breakpoint at the given address. Every time the code is run by any thread, a callback function is called. This is useful to know when certain parts of the debugee's code are being run (for example, set it at the beginning of a function to see how many times it's called).

The **hook_function** method sets a code breakpoint at the beginning of a function and allows you to set two callbacks - one when entering the function and another when returning from it. It works pretty much like the *apiHooks* property of

the *EventHandler* class, only it doesn't need the function to be exported by a DLL library. It's useful for intercepting calls to internal functions of the debugee, if you know where they are.

The **watch_variable** method sets a hardware breakpoint at the given address. Every time a read or write access is made to that address, a callback function is called. It's useful for tracking accesses to a variable (for example, a member of a C++ object in the heap). It works only on specific threads, to monitor the variable on the entire process you must set a watch for each thread.

Finally, the **watch_buffer** method sets a page breakpoint at the given address range. Every time a read or write access is made to that part of the memory a callback function is called. It's similar to *watch_variable* but it works for the entire process, not just a single thread, and it allows any range to be specified (*watch_variable* only works for small address ranges, from 1 to 8 bytes).

Debug objects also allow *stalking*. Stalking basically means to set one-shot breakpoints - that is, breakpoints that are automatically disabled after they're hit for the first time. The term was originally coined by **Pedram Amini** for his Process Stalker tool, and this technique is key to differential debugging.

The stalking methods and their equivalents are the following:

Stalking method	*Equivalent to*
stalk_at	break_at
stalk_function	hook_function
stalk_variable	watch_variable
stalk_buffer	watch_buffer

Example #8: setting a breakpoint

Download

```
# This function will be called when our breakpoint is hit
def action_callback( event ):

    # Get the address of the top of the stack
    stack   = event.get_thread().get_sp()

    # Get the return address of the call
    address = event.get_process().read_pointer( stack )

    # Get the process and thread IDs
    pid     = event.get_pid()
    tid     = event.get_tid()

    # Show a message to the user
    message = "kernel32!CreateFileW called from %s by thread %d at process %d"
    print message % ( HexDump.address(address), tid, pid )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object
        module = event.get_module()

        # If it's kernel32.dll...
        if module.match_name("kernel32.dll"):

            # Get the process ID
```

```
            pid = event.get_pid()

            # Get the address of CreateFile
            address = module.resolve( "CreateFileW" )

            # Set a breakpoint at CreateFile
            event.debug.break_at( pid, address, action_callback )

            # If you use stalk_at instead of break_at,
            # the message will only be shown once
            #
            # event.debug.stalk_at( pid, address, action_callback )
```

Example #9: hooking a function

Download

```
# This function will be called when the hooked function is entered
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string
    lpFmt = event.get_process().peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    parameters = event.get_thread().read_stack_dwords( count, offset = 3 )

    # Show a message to the user
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID
            pid = event.get_pid()

            # Get the address of wsprintf
            address = module.resolve( "wsprintfW" )

            # Hook the wsprintf function
            event.debug.hook_function( pid, address, wsprintf, paramCount = 2 )

            # Use stalk_function instead of hook_function
            # to be notified only the first time the function is called
            #
            # event.debug.stalk_function( pid, address, wsprintf, paramCount = 2 )
```

Example #10: watching a variable

Download

```
# This function will be called when the breakpoint is hit
def entering( event ):

    # Get the thread object
    thread = event.get_thread()

    # Get the thread ID
    tid = thread.get_tid()

    # Get the return address location (the top of the stack)
    stack_top = thread.get_sp()

    # Get the return address and the parameters from the stack
    return_address, hModule, lpProcName = thread.read_stack_dwords( 3 )

    # Get the string from the process memory
    procedure_name = event.get_process().peek_string( lpProcName )

    # Show a message to the user
    message = "%.08x: GetProcAddress(0x%.08x, %r);"
    print message % ( return_address, hModule, procedure_name )

    # Watch the DWORD at the top of the stack
    try:
        event.debug.stalk_variable( tid, stack_top, 4, returning )
        #event.debug.watch_variable( tid, stack_top, 4, returning )

    # If no more slots are available, set a code breakpoint at the return address
    except RuntimeError:
        event.debug.stalk_at( event.get_pid(), return_address, returning_2 )


# This function will be called when the variable is accessed
def returning( event ):

    # Get the address of the watched variable
    variable_address = event.breakpoint.get_address()

    # Stop watching the variable
    event.debug.dont_stalk_variable( event.get_tid(), variable_address )
    #event.debug.dont_watch_variable( event.get_tid(), variable_address )

    # Get the return address (in the stack)
    return_address = event.get_process().read_uint( variable_address )

    # Get the return value (in EAX)
    return_value = event.get_thread().get_context() [ 'Eax' ]

    # Show a message to the user
    message = "%.08x: GetProcAddress() returned 0x%.08x"
    print message % ( return_address, return_value )


# This function will be called if we ran out of hardware breakpoints,
```

```
# and we ended up setting a code breakpoint at the return address
def returning_2( event ):

    # Get the return address from the breakpoint
    return_address = event.breakpoint.get_address()

    # Remove the code breakpoint
    event.debug.dont_stalk_at( event.get_pid(), return_address )

    # Get the return value (in EAX)
    return_value = event.get_thread().get_context() [ 'Eax' ]

    # Show a message to the user
    message = "%.08x: GetProcAddress() returned 0x%.08x"
    print message % ( return_address, return_value )


# This event handler sets a breakpoint at kernel32!GetProcAddress
class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object
        module = event.get_module()

        # If it's kernel32...
        if module.match_name("kernel32.dll"):

            # Get the process ID
            pid = event.get_pid()

            # Get the address of GetProcAddress
            address = module.resolve( "GetProcAddress" )

            # Set a breakpoint at the entry of the GetProcAddress function
            event.debug.break_at( pid, address, entering )
```

Example #11: watching a buffer

Download

```
class MyHook (object):

    # Keep record of the buffers we watch
    def __init__(self):
        self.__watched = dict()


    # This function will be called when entering the hooked function
    def entering( self, event, ra, hModule, lpProcName ):

        # Ignore calls using ordinals intead of names
        if lpProcName & 0xFFFF0000 == 0:
            return

        # Get the procedure name
        procName = event.get_process().peek_string( lpProcName )
```

```
        # Ignore calls using an empty string
        if not procName:
            return

        # Show a message to the user
        print "GetProcAddress( %r );" % procName

        # Watch the procedure name buffer for access
        pid     = event.get_pid()
        address = lpProcName
        size    = len(procName) + 1
        action  = self.accessed
        event.debug.watch_buffer( pid, address, size, action )

        # Use stalk_buffer instead of watch_buffer to be notified
        # only of the first access to the buffer.
        #
        # event.debug.stalk_buffer( pid, address, size, action )

        # Remember the location of the buffer
        self.__watched[ event.get_tid() ] = ( address, size )


    # This function will be called when leaving the hooked function
    def leaving( self, event, return_value ):

        # Get the thread ID
        tid = thread.get_tid()

        # Get the buffer location
        ( address, size ) = self.__watched[ tid ]

        # Stop watching the buffer
        event.debug.dont_watch_buffer( event.get_pid(), address, size )
        #event.debug.dont_stalk_buffer( event.get_pid(), address, size )

        # Forget the buffer location
        del self.__watched[ tid ]


    # This function will be called every time the procedure name buffer is accessed
    def accessed( self, event ):

        # Show the user where we're running
        thread = event.get_thread()
        pc     = thread.get_pc()
        code   = thread.disassemble( pc, 0x10 ) [0]
        print "0x%.08x: %s" % ( code[0], code[2].lower() )


class MyEventHandler( EventHandler ):

    # Called guard page exceptions NOT raised by our breakpoints
    def guard_page( self, event ):
        print event.get_exception_name()

    # Called on DLL load events
    def load_dll( self, event ):
```

```
        # Get the new module object
        module = event.get_module()

        # If it's kernel32...
        if module.match_name("kernel32.dll"):

            # Get the process ID
            pid = event.get_pid()

            # Get the address of wsprintf
            address = module.resolve( "GetProcAddress" )

            # Hook the wsprintf function
            event.debug.hook_function( pid, address, MyHook().entering, paramCount = 2 )
```

Labels

Labels are used to represent memory locations in a more user-friendly way than simply using their addresses. This is useful to provide a better user interface, both for input and output. Also, labels can be useful when DLL libraries in a debugee are relocated on each run - memory addresses change every time, but labels don't.

For example, the label *"kernel32 CreateFileA"* always points to the *CreateFileA* function of the *kernel32.dll* library. The actual memory address, on the other hand, may change across Windows versions.

In addition to exported functions, debugging symbols are used whenever possible.

A complete explanation on how labels work can be found at the wiki page HowLabelsWork.

Example #12: getting the label for a given memory address

Download

```
from winappdbg import System, Process

def print_label( pid, address ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Process object
    process = Process( pid )

    # Lookup it's modules
    process.scan_modules()

    # Resolve the requested label address
    label = process.get_label_at_address( address )

    # Print the label
    print "%s == 0x%.08x" % ( label, address )
```

Example #13: resolving a label back into a memory address

Download

```
from winappdbg import System, Process

def print_label_address( pid, label ):

    # Request debug privileges
    System.request_debug_privileges()

    # Instance a Process object
    process = Process( pid )

    # Lookup it's modules
    process.scan_modules()

    # Resolve the requested label address
    address = process.resolve_label( label )

    # Print the address
    print "%s == 0x%.08x" % ( label, address )
```

1.3.3 The Win32 API wrappers

The `win32` submodule provides a collection of useful API wrappers for most operations needed by a debugger. This will allow you to perform any task that the abstraction layer for some reason can't deal with, or won't deal with in the way you need. In most cases you won't need to resort to this, but it's important to know it's there.

Except in some rare cases, the rationale to port the API calls to Python was:

- Take Python basic types as input, return Python basic types as output.
- Functions that in C take an output pointer and a size as input, in Python take neither and return the output data directly (the wrapper takes care of allocating the memory buffers).
- Functions that in C have to be called twice (first to get the buffer size, then to get the data) in Python only have to be called once (returns the data directly).
- Functions in C with more than one output pointer return tuples of data in Python.
- Functions in C that return an error condition, raise a Python exception (*WindowsError*) on error and return the data on success.
- Default parameter values were added when possible. The default for all optional pointers is *NULL*. The default flags are usually the ones that provide all possible access (for example, the default flags value for *GetThreadContext* is *CONTEXT_ALL*)
- For APIs with ANSI and Widechar versions, both versions are wrapped. If at least one parameter is a Unicode string en Widechar version is called (and all string parameters are converted to Unicode), otherwise the ANSI version is called. Either ANSI or Widechar versions can be used explicitly (for example, *CreateFile* can be called as *CreateFileA* or *CreateFileW*).

Example #1: finding a DLL in the search path

Download

```
import sys

from winappdbg import win32

fullpath, basename = win32.SearchPath( None, sys.argv[1], '.dll' )
```

```
print "Full path: %s" % fullpath
print "Base name: %s" % basename
```

Example #2: killing a process by attaching to it

Download

```
import sys
import thread

from winappdbg import win32

def processKiller(dwProcessId):

    # Attach to the process
    win32.DebugActiveProcess( dwProcessId )

    # Quit the current thread
    thread.exit()
```

Example #3: enumerating heap blocks using the Toolhelp library

Download

```
from winappdbg.win32 import *

def print_heap_blocks( pid ):

    # Determine if we have 32 bit or 64 bit pointers
    if sizeof(SIZE_T) == sizeof(DWORD):
        fmt = "%.8x\t%.8x\t%.8x"
        hdr = "%-8s\t%-8s\t%-8s"
    else:
        fmt = "%.16x\t%.16x\t%.16x"
        hdr = "%-16s\t%-16s\t%-16s"

    # Print a banner
    print "Heaps for process %d:" % pid
    print hdr % ("Heap ID", "Address", "Size")

    # Create a snapshot of the process, only take the heap list
    hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPHEAPLIST, pid )

    # Enumerate the heaps
    heap = Heap32ListFirst( hSnapshot )
    while heap is not None:

        # For each heap, enumerate the entries
        entry = Heap32First( heap.th32ProcessID, heap.th32HeapID )
        while entry is not None:

            # Print the heap id and the entry address and size
            print fmt % (entry.th32HeapID, entry.dwAddress, entry.dwBlockSize)

            # Next entry in the heap
```

```
            entry = Heap32Next( entry )

        # Next heap in the list
        heap = Heap32ListNext( hSnapshot )

    # No need to call CloseHandle, the handle is closed automatically when it goes out of scope
    return
```

Example #4: enumerating modules using the Toolhelp library

Download

```
from winappdbg.win32 import *

def print_modules( pid ):

    # Determine if we have 32 bit or 64 bit pointers
    if sizeof(SIZE_T) == sizeof(DWORD):
        fmt = "%.8x    %.8x    %s"
        hdr = "%-8s    %-8s    %s"
    else:
        fmt = "%.16x    %.16x    %s"
        hdr = "%-16s    %-16s    %s"

    # Print a banner
    print "Modules for process %d:" % pid
    print
    print hdr % ("Address", "Size", "Path")

    # Create a snapshot of the process, only take the heap list
    hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, pid )

    # Enumerate the modules
    module = Module32First( hSnapshot )
    while module is not None:

        # Print the module address, size and pathname
        print fmt % ( module.modBaseAddr,
                      module.modBaseSize,
                      module.szExePath )

        # Next module in the process
        module = Module32Next( hSnapshot )

    # No need to call CloseHandle, the handle is closed automatically when it goes out of scope
    return
```

Example #5: enumerating device drivers

Download

```
from winappdbg.win32 import *

def print_drivers( fFullPath = False ):

    # Determine if we have 32 bit or 64 bit pointers
```

```
    if sizeof(SIZE_T) == sizeof(DWORD):
        fmt = "%.08x\t%s"
        hdr = "%-8s\t%s"
    else:
        fmt = "%.016x\t%s"
        hdr = "%-16s\t%s"

    # Get the list of loaded device drivers
    ImageBaseList = EnumDeviceDrivers()
    print "Device drivers found: %d" % len(ImageBaseList)
    print
    print hdr % ("Image base", "File name")

    # For each device driver...
    for ImageBase in ImageBaseList:

        # Get the device driver filename
        if fFullPath:
            DriverName = GetDeviceDriverFileName(ImageBase)
        else:
            DriverName = GetDeviceDriverBaseName(ImageBase)

        # Print the device driver image base and filename
        print fmt % (ImageBase, DriverName)
```

1.3.4 More examples

Set a debugging timeout

Sometimes you'll want to set a maximum time to debug your target, especially when fuzzing. This is an example on how to code a custom debugging loop with a timeout. It launches the Windows Calculator and stops when the target process is closed or after a 5 seconds timeout.

Download

```
from winappdbg import *
from time import time

dbg = Debug(bKillOnExit = True)
try:
    dbg.execl('calc.exe')
    maxTime = time() + 5    # 5 seconds timeout
    while dbg.get_debugee_count() > 0 and time() < maxTime:
        try:
            print time()
            event = dbg.wait(1000)
        except WindowsError, e:
            if win32.winerror(e) in (win32.ERROR_SEM_TIMEOUT, win32.WAIT_TIMEOUT):
                continue
            raise
        try:
            dbg.dispatch(event)
        finally:
            dbg.cont(event)
finally:
    dbg.stop()
```

Dump the memory of a process

This is an example on how to dump the memory map and contents of a process into an SQLite database. A table is created where each row is a memory region, and the columns are the properties of that region (address, size, mapped filename, etc.) and it's data. The data is compressed using zlib to reduce the database size, but simply commenting out line 160 stores the data in uncompressed form.

Download

```
import os
import sys
import zlib
import winappdbg
from winappdbg import win32

try:
    import sqlite3 as sqlite
except ImportError:
    from pysqlite2 import dbapi2 as sqlite

# Create a snaphot of running processes
system = winappdbg.System()
system.request_debug_privileges()
system.scan_processes()

# Get all processes that match the requested filenames
for filename in sys.argv[1:]:
    for process, pathname in system.find_processes_by_filename(filename):
        pid = process.get_pid()
        print "Dumping memory for process ID %d" % pid

        # Parse the database filename
        dbfile   = '%d.db' % pid
        if os.path.exists(dbfile):
            counter = 1
            while 1:
                dbfile = '%d_%.3d.db' % (pid, counter)
                if not os.path.exists(dbfile):
                    break
                counter += 1
            del counter
        print "Creating database %s" % dbfile

        # Connect to the database and get a cursor
        database = sqlite.connect(dbfile)
        cursor   = database.cursor()

        # Create the table for the memory map
        cursor.execute("""
            CREATE TABLE MemoryMap (
                Address INTEGER PRIMARY KEY,
                Size    INTEGER,
                State   STRING,
                Access  STRING,
                Type    STRING,
                File    STRING,
                Data    BINARY
            )
        """)
```

```
        # Get a memory map of the process
        memoryMap       = process.get_memory_map()
        mappedFilenames = process.get_mapped_filenames(memoryMap)

        # For each memory block in the map...
        for mbi in memoryMap:

            # Address and size of memory block
            BaseAddress = mbi.BaseAddress
            RegionSize  = mbi.RegionSize

            # State (free or allocated)
            if   mbi.State == win32.MEM_RESERVE:
                State   = "Reserved"
            elif mbi.State == win32.MEM_COMMIT:
                State   = "Commited"
            elif mbi.State == win32.MEM_FREE:
                State   = "Free"
            else:
                State   = "Unknown"

            # Page protection bits (R/W/X/G)
            if mbi.State != win32.MEM_COMMIT:
                Protect = ""
            else:
                if   mbi.Protect & win32.PAGE_NOACCESS:
                    Protect = "--- "
                elif mbi.Protect & win32.PAGE_READONLY:
                    Protect = "R-- "
                elif mbi.Protect & win32.PAGE_READWRITE:
                    Protect = "RW- "
                elif mbi.Protect & win32.PAGE_WRITECOPY:
                    Protect = "RC- "
                elif mbi.Protect & win32.PAGE_EXECUTE:
                    Protect = "--X "
                elif mbi.Protect & win32.PAGE_EXECUTE_READ:
                    Protect = "R-X "
                elif mbi.Protect & win32.PAGE_EXECUTE_READWRITE:
                    Protect = "RWX "
                elif mbi.Protect & win32.PAGE_EXECUTE_WRITECOPY:
                    Protect = "RCX "
                else:
                    Protect = "??? "
                if   mbi.Protect & win32.PAGE_GUARD:
                    Protect += "G"
                else:
                    Protect += "-"
                if   mbi.Protect & win32.PAGE_NOCACHE:
                    Protect += "N"
                else:
                    Protect += "-"
                if   mbi.Protect & win32.PAGE_WRITECOMBINE:
                    Protect += "W"
                else:
                    Protect += "-"

            # Type (file mapping, executable image, or private memory)
            if   mbi.Type == win32.MEM_IMAGE:
```

```
            Type    = "Image"
        elif mbi.Type == win32.MEM_MAPPED:
            Type    = "Mapped"
        elif mbi.Type == win32.MEM_PRIVATE:
            Type    = "Private"
        elif mbi.Type == 0:
            Type    = ""
        else:
            Type    = "Unknown"

        # Mapped file name, if any
        FileName = mappedFilenames.get(BaseAddress, None)

        # Read the data contained in the memory block, if any
        Data = None
        if mbi.has_content():
            print 'Reading %s-%s' % (
                winappdbg.HexDump.address(BaseAddress),
                winappdbg.HexDump.address(BaseAddress + RegionSize)
            )
            Data = process.read(BaseAddress, RegionSize)
            Data = zlib.compress(Data, zlib.Z_BEST_COMPRESSION)
            Data = sqlite.Binary(Data)

        # Output a row in the table
        cursor.execute(
            'INSERT INTO MemoryMap VALUES (?, ?, ?, ?, ?, ?, ?)',
            (BaseAddress, RegionSize, State, Protect, Type, FileName, Data)
        )

    # Commit the changes, close the cursor and the database
    database.commit()
    cursor.close()
    database.close()
    print "Ok."
print "Done."
```

Find alphanumeric addresses to jump to

This example will find all memory addresses in a target process that are executable and whose address consists of alphanumeric characters only. This is useful when exploiting a stack buffer overflow and the input string is limited to alphanumeric characters only.

Download

```
from struct import pack
from winappdbg import System, Process, HexDump

# Iterator of alphanumeric executable addresses
def iterate_alnum_jump_addresses(memory_snapshot):

    # Determine the size of a pointer in the current architecture
    if System.bits == 32:
        fmt = 'L'
    elif System.bits == 64:
        fmt = 'Q'
    else:
```

```
        raise NotImplementedError

    # Iterate the memory regions of the target process
    for mbi in memory_snapshot:

        # Discard non executable memory
        if not mbi.is_executable():
            continue

        # Yield each alphanumeric address in this memory region.
        address     = mbi.BaseAddress
        max_address = address + mbi.RegionSize
        while address < max_address:
            packed = pack(fmt, address)
            if packed.isalnum():
                yield address, packed
            address = address + 1

# Iterate and print alphanumeric executable addresses.
def print_alnum_jump_addresses(pid):

    # Request debug privileges so we can inspect the memory of services too.
    System.request_debug_privileges()

    # Suspend the process so there are no malloc's and free's while iterating.
    process = Process(pid)
    process.suspend()
    try:

        # Get an iterator for the target process memory.
        iterator = process.generate_memory_snapshot()

        # Print each executable alphanumeric address.
        for address, packed in iterate_alnum_jump_addresses(iterator):
            print HexDump.address(address), repr(packed)

    # Resume the process when we're done.
    # This is inside a "finally" block, so if the program is interrupted
    # for any reason we don't leave the process suspended.
    finally:
        process.resume()
```

Trace all calls to text drawing in GDI

This example hooks all text drawing functions in GDI and prints the text. It can be useful to extract text messages and logs from GUI programs.

Download

```
from winappdbg import Debug, EventHandler, DebugLog
from ctypes import *

#------------------------------------------------------------------------------

# BOOL TextOut(
#   __in  HDC hdc,
#   __in  int nXStart,
```

```
#   __in  int nYStart,
#   __in  LPCTSTR lpString,
#   __in  int cbString
# );

def TextOutA(event, ra, hdc, nXStart, nYStart, lpString, cbString):
    log_ansi(event, "TextOutA", lpString, cbString)

def TextOutW(event, ra, hdc, nXStart, nYStart, lpString, cbString):
    log_wide(event, "TextOutW", lpString, cbString)

# BOOL ExtTextOut(
#   __in  HDC hdc,
#   __in  int X,
#   __in  int Y,
#   __in  UINT fuOptions,
#   __in  const RECT *lprc,
#   __in  LPCTSTR lpString,
#   __in  UINT cbCount,
#   __in  const INT *lpDx
# );
def ExtTextOutA(event, ra, hdc, X, Y, fuOptions, lprc, lpString, cbCount, lpDx):
    log_ansi(event, "ExtTextOutA", lpString, cbCount)

def ExtTextOutW(event, ra, hdc, X, Y, fuOptions, lprc, lpString, cbCount, lpDx):
    log_wide(event, "ExtTextOutW", lpString, cbCount)

# typedef struct _POLYTEXT {
#   int     x;
#   int     y;
#   UINT    n;
#   LPCTSTR lpstr;
#   UINT    uiFlags;
#   RECT    rcl;
#   int     *pdx;
# } POLYTEXT, *PPOLYTEXT;
class POLYTEXT(Structure):
    _fields_ = [
        ('x',       c_int),
        ('y',       c_int),
        ('n',       c_uint),
        ('lpstr',   c_void_p),
        ('uiFlags', c_uint),
        ('rcl',     c_uint * 4),
        ('pdx',     POINTER(c_int)),
    ]

# BOOL PolyTextOut(
#   __in  HDC hdc,
#   __in  const POLYTEXT *pptxt,
#   __in  int cStrings
# );

def PolyTextOutA(event, ra, hdc, pptxt, cStrings):
    process = event.get_process()
    sizeof_polytext = sizeof(POLYTEXT)
    while cStrings:
        txt = process.read_structure(pptxt, POLYTEXT)
```

```
        log_ansi(event, "PolyTextOutA", txt.lpstr, txt.n)
        pptxt = pptxt + sizeof_polytext
        cStrings = cStrings - 1

def PolyTextOutW(event, ra, hdc, pptxt, cStrings):
    process = event.get_process()
    sizeof_polytext = sizeof(POLYTEXT)
    while cStrings:
        txt = process.read_structure(pptxt, POLYTEXT)
        log_wide(event, "PolyTextOutW", txt.lpstr, txt.n)
        pptxt = pptxt + sizeof_polytext
        cStrings = cStrings - 1

#------------------------------------------------------------------------------

def log_ansi(event, fn, lpString, nCount):
    if lpString and nCount:
        if c_int(nCount).value == -1:
            lpString = event.get_process().peek_string(lpString, fUnicode = False)
        else:
            lpString = event.get_process().peek(lpString, nCount)
        print DebugLog.log_text("%s( %r );" % (fn, lpString))

def log_wide(event, fn, lpString, nCount):
    if lpString and nCount:
        if c_int(nCount).value == -1:
            lpString = event.get_process().peek_string(lpString, fUnicode = True)
        else:
            lpString = event.get_process().peek(lpString, nCount * 2)
            lpString = unicode(lpString, 'U16', 'replace')
        print DebugLog.log_text("%s( %r );" % (fn, lpString))

class MyEventHandler( EventHandler ):
    def load_dll(self, event):
        pid = event.get_pid()
        module = event.get_module()
        if module.match_name("gdi32.dll"):
            event.debug.hook_function(pid, module.resolve("TextOutA"),       TextOutA,       paramCou
            event.debug.hook_function(pid, module.resolve("TextOutW"),       TextOutW,       paramCou
            event.debug.hook_function(pid, module.resolve("ExtTextOutA"),    ExtTextOutA,    paramCou
            event.debug.hook_function(pid, module.resolve("ExtTextOutW"),    ExtTextOutW,    paramCou
            event.debug.hook_function(pid, module.resolve("PolyTextOutA"),   PolyTextOutA,   paramCou
            event.debug.hook_function(pid, module.resolve("PolyTextOutW"),   PolyTextOutW,   paramCou

def simple_debugger(argv):
    print DebugLog.log_text("Trace started on %s" % argv[0])
    debug = Debug( MyEventHandler() )
    try:
        debug.execv(argv)
        debug.loop()
    finally:
        debug.stop()
    print DebugLog.log_text("Trace stopped on %s" % argv[0])
```

Enumerate all named global atoms

Global atoms are WORD numeric values that can be associated to arbitrary strings. They are used primarily for IPC purposes on Windows. This example shows how to retrieve the string from any atom value.

Download

```
from winappdbg.win32 import GlobalGetAtomName, MAXINTATOM

# print all valid named global atoms to standard output
def print_atoms():
    for x in xrange(0, MAXINTATOM):
        try:
            n = GlobalGetAtomName(x)
            if n == "#%d" % x:      # comment out to print
                continue            # valid numeric atoms
            print "Atom %4x: %r" % (x, n)
        except WindowsError:
            pass
```

1.3.5 Advanced topics

This section contains some more detailed explanations on the internal workings of *WinAppDbg* and how to perform more complex tasks with it.

About the unique Crash keys

The key is a tuple of the elements I thought can uniquely identify a crash, at least to some practical extent, so crashes generated by the same bug will not be included more than once. It's supposed to be opaque to the user of the class, so it can easily be changed to reflect different heuristics without breaking existing code.

A more flexible implementation would be to have a set of classes of key objects to choose from, each with a different heuristic, coded in the comparison operator. For now I'll leave that for a future version, if the need ever arises.

So far this simple implementation using a tuple has worked well for me. But if you need something different, just derive from the *Crash* class and reimplement the *key()* method.

To disable detection completely, subclass *Crash* and return *self* at the *key()* method.

This is what I chose to include in the key and why:

- **Event code** and **exception code**:

  Wouldn't make sense not to include them. :)

- **Program counter** (EIP/RIP):

  The same fault in different places of the code are most likely different bugs. However, different faults in the same place are not necessarily the same bug, so we can't rely on this alone.

  To avoid problems with DLL relocations, a *label* is used whenever possible.

- **Stack trace** (EIP/RIP values only):

  This heuristic is actually meant to detect different ways of triggering the same bug, rather than different bugs. But it's also useful to detect heap overflows, since all of them will be triggered at the same set of EIPs (where the heap routines are located) but coming from different parent functions.

  To avoid problems with DLL relocations, *labels* are used whenever possible.

- **Debug string**:

  Different debug strings mean most likely different bugs. There's a catch: if the debug string is generated from something else (like the value of some variable we don't care about), this heuristic may fail and give us more crashes than we really wanted. This is the case for strings generated by heaps in debug mode, as they often include the heap chunk addresses. If this becomes a problem you can filter out the unwanted debug string events before passing them to the container.

This is what I chose **NOT** to include in the key and why:

- **Exception address**:

  Most exceptions caught are page faults, and in that case we're more interested in the program counter, since a page fault is generally triggered by corrupting a pointer, and the corrupted value itself isn't really useful to uniquely identifying the crash it produces.

  Then again I still want to review this heuristic for each specific type of exception for the next version, to make sure it's not getting too many false negatives. I didn't give much thought for scenarios other than page faults when I thought about this one. :(

- **First chance** or **second chance**:

  Generally second chance exceptions are exactly the same as first chance exceptions, they simply mean the application didn't handle them. Depending on the application you're debugging you could be interested in logging either first chance or second chance exceptions only, but rarely both.

- **Process** and **thread IDs**:

  One might say, two processes could crash at the same address because of different bugs. But the problem is, the process and thread IDs are dependent on a particular execution of the target application, and we want to be able to compare crashes from multiple executions.

- **Stack contents** and **register values**:

  Both are most likely to contain garbage we're not interested in, plus many values are dependent on a particular execution of the application.

  By ignoring this we might be missing different ways to trigger the same bug, though.

A closer look at how labels work

Labels are an approximated way of referencing memory locations across different executions of the same process, or different processes with common modules. They are not meant to be perfectly unique, and some errors may occur when multiple modules with the same name are loaded, or when module filenames can't be retrieved.

The following examples assume there is a running process called *"calc.exe"* and the current user has enough privileges to debug it. The resolved addresses may vary in your system.

Labels syntax

This is the syntax of labels:

Where all components are optional and blank spaces are ignored.

- The **module** is a module name as returned by *Module.get_name()*.
- The **function** is a string with an exported function name.
- The **ordinal** is an integer with an exported function ordinal.

```
module + offset !
module ! function + offset
module ! # ordinal + offset
```

- The **offset** is an integer number. It may be an offset from the module base address, or the function address. If not specified, the default is *0*.

If debugging symbols are available, they are used automatically in addition to exported functions.

Integer numbers in labels may be expressed in any format supported by HexInput.integer(), but by default they are in hexadecimal format (for example *0x1234*).

If only the **module** or the **function** are specified, but not both, the exclamation mark (**!**) may be omitted in fuzzy mode (explained later in this document). However, resolving the label may be a little slower, as all module names have to be checked to resolve the ambiguity.

Generating labels

To create a new label, use the **parse_label** static method of the **Process** class:

```
>>> import winappdbg
>>> winappdbg.Process.parse_label()                                    # no arguments
'0x0'
>>> winappdbg.Process.parse_label(None, None, None)                    # empty label
'0x0'
>>> winappdbg.Process.parse_label(None, None, 512)                     # offset or address
'0x200'
>>> winappdbg.Process.parse_label("kernel32")                          # module base
'kernel32!'
>>> winappdbg.Process.parse_label("kernel32", "CreateFileA")           # exported function...
'kernel32!CreateFileA'
>>> winappdbg.Process.parse_label("kernel32", 16)                      # ...by ordinal
'kernel32!#0x10'
>>> winappdbg.Process.parse_label("kernel32", None, 512)               # module base + offset
'kernel32!0x200'
>>> winappdbg.Process.parse_label(None, "CreateFileA")                 # function in any module...
'!CreateFileA'
>>> winappdbg.Process.parse_label(None, 16)                            # ...by ordinal
'!#0x10'
>>> winappdbg.Process.parse_label(None, "CreateFileA", 512)            # ...plus an offset...
'!CreateFileA+0x200'
>>> winappdbg.Process.parse_label(None, 16, 512)                       # ...by ordinal
'!#0x10+0x200'
>>> winappdbg.Process.parse_label("kernel32", "CreateFileA", 512)      # full label...
'kernel32!CreateFileA+0x200'
>>> winappdbg.Process.parse_label("kernel32", 16, 512)                 # ...by ordinal
'kernel32!#0x10+0x200'
```

The **get_label_at_address** method automatically guesses a good label for any given address in the process.

```
>>> import winappdbg
>>> aSystem = winappdbg.System()
>>> aSystem.request_debug_privileges()
True
>>> aSystem.scan()
```

```
>>> aProcess = aSystem.find_processes_by_filename("calc.exe")[0][0]
>>> aProcess.get_label_at_address(0x7c801a28)                    # address within kernel32.dl
'kernel32+0x1a28!'
```

Splitting labels

To split labels back to their original *module*, *function* and *offset* components there are two modes. The **strict** mode allows only labels that have been generated with *parse_label*. The **fuzzy** mode has a more flexible syntax, and supports some notation abuses that can only be resolved by a live *Process* instance.

The **split_label** method will automatically use the *strict* mode when called as a static method, and the *fuzzy* mode when called as an instance method:

```
winappdbg.Process.split_method( "kernel32!CreateFileA" )          # static method, using the strict
aProcessInstance.split_method( "CreateFileA" )                    # instance method, using the fuzz
```

The **sanitize_label** method takes a fuzzy syntax label and converts it to strict syntax. This is useful when reading labels from user input and storing them for later use, when the process is no longer being debugged.

Strict syntax mode To explicitly use the *strict* syntax mode, call the **split_label_strict** method:

```
>>> import winappdbg
>>> winappdbg.Process.split_label_strict(None)                          # empty label
(None, None, None)
>>> winappdbg.Process.split_label_strict('')                            # empty label
(None, None, None)
>>> winappdbg.Process.split_label_strict('0x0')                         # NULL pointer
(None, None, None)
>>> winappdbg.Process.split_label_strict('0x200')                       # any memory address
(None, None, 512)
>>> winappdbg.Process.split_label_strict('0x200 ! ')                    # meaningless ! is ignored
(None, None, 512)
>>> winappdbg.Process.split_label_strict(' ! 0x200')                    # meaningless ! is ignored
(None, None, 512)
>>> winappdbg.Process.split_label_strict('kernel32 ! ')                 # module base
('kernel32', None, None)
>>> winappdbg.Process.split_label_strict('kernel32 ! CreateFileA')      # exported function...
('kernel32', 'CreateFileA', None)
>>> winappdbg.Process.split_label_strict('kernel32 ! # 0x10')           # ...by ordinal
('kernel32', 16, None)
>>> winappdbg.Process.split_label_strict('kernel32 ! 0x200')            # base address + offset...
('kernel32', None, 512)
>>> winappdbg.Process.split_label_strict('kernel32 + 0x200 ! ')         # ...alternative syntax
('kernel32', None, 512)
>>> winappdbg.Process.split_label_strict(' ! CreateFileA')              # function in any module.
(None, 'CreateFileA', None)
>>> winappdbg.Process.split_label_strict(' ! # 0x10')                   # ...by ordinal
(None, 16, None)
>>> winappdbg.Process.split_label_strict(' ! CreateFileA + 0x200')      # ...plus an offset...
(None, 'CreateFileA', 512)
>>> winappdbg.Process.split_label_strict(' ! # 0x10 + 0x200')           # ...by ordinal
(None, 16, 512)
>>> winappdbg.Process.split_label_strict('kernel32 ! CreateFileA + 0x200') # full label...
('kernel32', 'CreateFileA', 512)
>>> winappdbg.Process.split_label_strict('kernel32 ! # 0x10 + 0x200')   # ...by ordinal
('kernel32', 16, 512)
```

Fuzzy syntax mode To explicitly use the *fuzzy* syntax mode, call the **split_label_fuzzy** method:

```
>>> import winappdbg
>>> aSystem = winappdbg.System()
>>> aSystem.request_debug_privileges()
True
>>> aSystem.scan()
>>> aProcess = aSystem.find_processes_by_filename("calc.exe")[0][0]
>>> aProcess.split_label_fuzzy( "kernel32" )                    # allows no ! sign
('kernel32', None, None)
>>> aProcess.split_label_fuzzy( "kernel32.dll" )                # strips the default extensio
('kernel32', None, None)
>>> aProcess.split_label_fuzzy( "CreateFileA" )                 # can tell a module from a fu
(None, 'CreateFileA', None)
>>> aProcess.split_label_strict( "0x7c800000" )                 # strict mode can't tell base
(None, None, 2088763392)
>>> aProcess.split_label_fuzzy( "0x7c800000" )                  # fuzzy mode can tell base ad
('kernel32', None, None)
>>> aProcess.split_label_fuzzy( "0x7c800000 + 6696" )           # base address + offset
('kernel32', None, 6696)
>>> aProcess.split_label_fuzzy("0x7c801a28")                    # any memory address
('kernel32', None, 6696)
>>> aProcess.split_label_fuzzy( "0x200" )                       # address outside of any loa
(None, None, 512)
```

Resolving labels

The **resolve_label** method allows you to get the actual memory address the label points at the given process. If the module is not loaded or the function is not exported, the method fails with an exception.

```
>>> import winappdbg
>>> aSystem = winappdbg.System()
>>> aSystem.request_debug_privileges()
True
>>> aSystem.scan()
>>> aProcess = aSystem.find_processes_by_filename("calc.exe")[0][0]
>>> aProcess.resolve_label( "kernel32" )                        # module base
2088763392
>>> aProcess.resolve_label( "KERNEL32" )                        # module names are case inse
2088763392
>>> aProcess.resolve_label( "kernel32.dll" )
2088763392
>>> aProcess.resolve_label( "kernel32 + 0x200" )                # module + offset
2088763904
>>> aProcess.resolve_label( "kernel32 ! CreateFileA" )
2088770088
>>> aProcess.resolve_label( "CreateFileA" )                     # all loaded modules are sea
2088770088
>>> aProcess.resolve_label( " # 16" )                           # function ordinal
2090010350
>>> aProcess.resolve_label( " # 0x10" )                         # function ordinal in hexa
2090010350
>>> aProcess.resolve_label( "kernel32 ! CreateFileA + 0x200" )
2088770600
>>> aProcess.resolve_label( "CreateFileA + 0x200" )
2088770600
>>> aProcess.resolve_label( "0x7c800000" )                      # module base address
```

```
2088763392
>>> aProcess.resolve_label( "0x7c800000 ! CreateFileA" )
2088770088
```

A closer look at how breakpoints work

This wiki page aims at giving a more detailed explanation on how breakpoints really work, behind the simplified *break_at*, *stalk_at*, *watch_variable* and *watch_buffer* interface provided by the *Debug* objects. With this you can fine-tune the use of breakpoints in your programs.

Breakpoint types

Debug objects support three kinds of breakpoints: *code* breakpoints, *page* breakpoints and *hardware* breakpoints. Each kind of breakpoint causes an exception to be raised in the debugee. These exceptions are caught and handled automatically by the debugger.

Breakpoints have to be defined first and enabled later. The rationale behind this is that you can define as many breakpoints as you want, and then switch them on and off as you need to without having to delete them. This leads to a more efficient use of resources, and is consistent with what one expects of debuggers.

Code breakpoints are defined by the **define_code_breakpoint** method, enabled by the **enable_code_breakpoint** method. You can guess what are the methods to disable and erase code breakpoints. :)

Similarly, page breakpoints are defined by **define_page_breakpoint**, hardware breakpoints are defined by **define_hardware_breakpoint**, and so on.

Code breakpoints *Code* breakpoints are implemented by inserting an int3 instruction (xCC) at the address specified. When a thread tries to execute this instruction, a breakpoint exception is generated. It's global to the process because it overwrites the code to break at.

When hit, code breakpoints trigger a **breakpoint** event at your *event handler*.

Let's look at the signature of *define_code_breakpoint*:

```
def define_code_breakpoint(self, dwProcessId, address,   condition = True,
                                                             action = None):
```

Where **dwProcessId** is the Id of the process where we want to set the breakpoint and **address** is the location of the breakpoint in the process memory. The other two parameters are optional and will be *explained later*.

Page breakpoints *Page* breakpoints are implemented by changing the access permissions of a given memory page. This causes a guard page exception to be generated when the given page is accessed anywhere in the code of the process.

When hit, page breakpoints trigger a **guard_page** event at your *event handler*.

Let's see the signature of *define_page_breakpoint*:

```
def define_page_breakpoint(self, dwProcessId, address,       pages = 1,
                                                          condition = True,
                                                             action = None):
```

Where **dwProcessId** is the same. But now **address** needs to be page-aligned and **pages** is the number of pages covered by the breakpoint. This is because VirtualProtectEx() works only with entire pages, you can't change the access permissions on individual bytes.

Hardware breakpoints *Hardware* breakpoints are implemented by writing to the debug registers (DR0-DR7) of a given thread, causing a single step exception to be generated when the given address is accessed anywhere in the code for that thread only. It's important to remember the debug registers have different values for each thread, so this can't be done global to the process (you can set the same breakpoint in all the threads, though).

When hit, hardware breakpoints trigger a **single_step** event at your *event handler*.

The signature of *define_hardware_breakpoint* is this:

```
def define_hardware_breakpoint(self, dwThreadId, address,
                                            triggerFlag = BP_BREAK_ON_ACCESS,
                                               sizeFlag = BP_WATCH_DWORD,
                                              condition = True,
                                                 action = None):
```

Seems a little more complicated than the others. :)

The first difference we see is the *dwProcessId* parameter has been replaced by **dwThreadId**. This is because hardware breakpoints are only applicable to single threads, not to the entire process.

The **address** is any address in the process memory, even if it's unmapped. This can be useful to set breakpoints on DLL libraries before they are loaded (as long as they don't get relocated).

The **triggerFlag** parameter is used to specify exactly what event will trigger this breakpoint. There are four constants available:

Constant	*Meaning*
Debug.**BP_BREAK_ON_EXECUTION**	Break when executing on *address*.
Debug.**BP_BREAK_ON_WRITE**	Break when writing to *address*.
Debug.**BP_BREAK_ON_ACCESS**	Break when reading or writing to *address*.
Debug.**BP_BREAK_ON_IO_ACCESS**	*(Not currently used by today's hardware.)*

The **sizeFlag** parameter says how large is the memory region to watch. There are again four constants:

Constant	*Meaning*
Debug.**BP_WATCH_BYTE**	Applies to 1 byte from *address*.
Debug.**BP_WATCH_WORD**	Applies to 2 bytes (a word) from *address*.
Debug.**BP_WATCH_DWORD**	Applies to 4 bytes (a double word) from *address*.
Debug.**BP_WATCH_QWORD**	Applies to 8 bytes (a quad word) from *address*.

Since x86 processors only have enough room for **four** hardware breakpoints in the debug registers, you can **only enable four of them at a time for a single thread**. You can define as many as you want, though, provided you only keep a maximum of four enabled breakpoints per thread at any time.

Conditional and automatic breakpoints

We have seen above that all the methods to define breakpoins have the optional parameters **condition** and **action**. But what do they mean?

The *condition* parameter The **condition** parameter determines if the breakpoint is *conditional* or *unconditional*.

If it's set to *True* (the default value) the breakpoint is **unconditional**. Unconditional breakpoints always call the corresponding method of the event handler.

And if it's set to a **function** (or any other callable Python object), the breakpoint is **conditional**. Conditional breakpoints, when hit, call the *condition* callback. If this callback returns *True* the event handler method is also called, otherwise it isn't. This allows you to set breakpoints that will only trigger an event under specific conditions (for example, only stop the execution when *EAX* equals *0x100*, ignore it otherwise).

```
# condition callback
def eax_is_100(event):

  aThread = event.get_thread()
  Eax     = aThread.get_context()['Eax']

  if Eax == 0x100:

    # We are interested on this!
    return True

  # False alarm, ignore it...
  return False

# Will only break when eax is 100 in that process at that address
def break_when_eax_is_100(debug, pid, address):
  debug.define_code_breakpoint(pid, address, condition = eax_is_100)
  debug.enable_code_breakpoint(pid, address)
```

The *action* parameter The **action** parameter allows you to set another callback. When not used, the breakpoint is **interactive**, meaning when it's hit (and it's condition callback returns *True*) the event handler method is called. But when it's used, the breakpoint is **automatic**, and that means this callback is called **instead** of the event handler method.

Automatic breakpoints are useful for setting tasks to be done "behind the back" of the event handler, so they don't have to be treated as special cases by your event handler routines.

```
# action callback
def change_eax_value(event):

  # Get the thread that hit the breakpoint
  aThread = event.get_process()

  # Set a new value for the EAX register
  aThread.set_register('Eax', 0xBAADF00D)

# Will automatically change the return value of the function
def auto_change_return_value(debug, pid, address):
  # 'address' must be the location of the 'ret' instruction
  debug.define_code_breakpoint(pid, address, action = change_eax_value)
  debug.enable_code_breakpoint(pid, address)
```

Breakpoints can be both *conditional* and *automatic*. Here is another example reusing the code above:

```
# Will automatically change the return value of the function,
# but only when the original value was 0x100
def conditionally_change_return_value(debug, pid, address):
    # 'address' must be the location of the 'ret' instruction
    debug.define_code_breakpoint(pid, address, condition = eax_is_100,
                                               action = change_eax_value)
    debug.enable_code_breakpoint(pid, address)
```

One-shot breakpoints

Breakpoints of all types can also be **one-shot**. This means they're automatically disabled after being hit. This is useful for one time events, for example a debugger might want to set a one-shot breakpoint at the next instruction for tracing.

You could also set one-shot breakpoints to do code coverage, where multiple executions of the same code are not relevant.

Note that one-shot breakpoints are only **disabled**, not deleted, so you can enable them again. Any disabled breakpoint can be enabled again, as a normal breakpoint or as one-shot, independently of how it's been used before.

To set one-shot breakpoints, after defining them use one of the **enable_one_shot_code_breakpoint**, **enable_one_shot_page_breakpoint** or **enable_one_shot_hardware_breakpoint** methods to enable it.

```
# Will automatically change the return value of the function,
# but only when the original value was 0x100,
# and only the next time the function is called
def conditionally_change_return_value(debug, pid, address):
    # 'address' must be the location of the 'ret' instruction
    debug.define_code_breakpoint(pid, address, condition = eax_is_100,
                                               action = change_eax_value)
    debug.enable_one_shot_code_breakpoint(pid, address)
```

Batch operations on breakpoints

The following methods are provided for working on all breakpoints at once:

Method	*Description*
enable_all_breakpoints	Enables all disabled breakpoints in all processes.
enable_one_shot_all_breakpoints	Enables for one shot all disabled breakpoints in all processes.
disable_all_breakpoints	Disables all breakpoints in all processes.
erase_all_breakpoints	Erases all breakpoints in all processes.

These methods work with all breakpoints of a single process:

Method	*Description*
enable_process_breakpoints	Enables all disabled breakpoints for the given process.
enable_one_shot_process_breakpoints	Enables for one shot all disabled breakpoints for the given process.
disable_process_breakpoints	Disables all breakpoints for the given process.
erase_process_breakpoints	Erases all breakpoints for the given process.

Accessing the breakpoint objects

For even more fine-tuning you might also want to access the *Breakpoint* objects directly. The **get_code_breakpoint** method retrieves a code breakpoint in a process, **get_page_breakpoint** works for page breakpoints in a process, and **get_hardware_breakpoint** gets the hardware breakpoint in a thread.

While it's always safe to request information from a *Breakpoint* object, it may not be so when modifying it, so be careful what methods you call. The following methods are safe to call:

Method	*Description*
is_disabled	If *True*, breakpoint is disabled.
is_running	If *True*, breakpoint was recently hit.
is_here	Returns *True* if the breakpoint is within the given address range.
get_address	Returns the breakpoint location.
get_size	Returns the breakpoint size in bytes.
is_conditional	If True, the breakpoint is conditional.
get_condition	Returns the breakpoint *condition* parameter.
set_condition	Changes the breakpoint *condition* parameter.
is_automatic	If True, the breakpoint is automatic.
get_action	Returns the breakpoint *action* parameter.
set_action	Changes the breakpoint *action* parameter.
get_slot	*(For hardware breakpoints only)* Returns the debug register number used by this breakpoint, or *None* if the breakpoint is disabled or running.
get_trigger	*(For hardware breakpoints only)* Returns the *trigger* parameter.
get_watch	*(For hardware breakpoints only)* Returns the *watch* parameter.
get_size_in_pages	*(For page breakpoints only)* Get the number of pages covered by the breakpoint.
align_address_to_page_start	*(For page breakpoints only)* Align the given address to the start of the page it occupies.
align_address_to_page_end	*(For page breakpoints only)* Align the given address to the end of the page it occupies.
get_buffer_size_in_pages	*(Static, for page breakpoints only)* Get the number of pages in use by the given buffer.

Listing the breakpoints

Debug objects also allow you to retrieve lists of defined breakpoints, filtered by different criteria. This listing methods return lists of tuples, and inside this tuples are the *Breakpoint* objects described earlier.

The following table describes the listing methods and what they return, where **pid** is a process ID, **tid** is a thread ID and **bp** is a *Breakpoint* object.

Method	*Description*
get_all_code_breakpoints	Returns all code breakpoints as a list of tuples (pid, bp).
get_all_page_breakpoints	Returns all page breakpoints as a list of tuples (pid, bp).
get_all_hardware_breakpoints	Returns all hardware breakpoints as a list of tuples (tid, bp).
get_process_code_breakpoints	Returns all code breakpoints for the given process.
get_process_page_breakpoints	Returns all page breakpoints for the given process.
get_thread_hardware_breakpoints	Returns all hardware breakpoints for the given thread.
get_process_hardware_breakpoints	Returns all hardware breakpoints for each thread in the given process as a list of tuples (tid, bp).

1.4 Building your own distribution packages

WinAppDbg is released under the `BSD license`, so as a user you are entitled to create derivative work and redistribute it if you wish. A makefile is provided to automatically generate the source distribution package and the Windows installer, and can also generate the documentation for all the modules using Epydoc.

1.4.1 Prerequisites

A Make utility is required to use the makefile. Without it you're going to have to run each command manually to generate the documentation and packages. We're using GNU Make for Windows from the GNU Win32 project.

Tar and BZip2 utilities are required to compress .tar.bz2 files. We're also using the packages from the GNU Win32 project.

The Epydoc package is required to autogenerate the documentation. [http://www.graphviz.org/ GraphViz] is used by Epydoc to generate UML graphs for the documentation.

This documentation was generated using Sphinx. The reStructuredText sources are provided with the source code downloads only.

A Latex compiler is used to generate the documentation in PDF format. We're currently using MikTex 2.7 on Windows.

The HTML help can be compiled to a .CHM file using Microsoft HTML Help Workshop.

The py2exe package is used to generate standalone binaries for the tools. This step is optional. You can (also optionally) compress the executables with UPX.

All of these tools should be present in the **PATH** environment variable.

Download Make for Windows
Download Tar for Windows
Download BZip2 for Windows
Download Epydoc
Download Sphinx
Download GraphViz
Download MikTex 2.7
Download HTML Help Workshop
Download py2exe
Download UPX

1.4.2 Installer script

Both the source code and Windows installer packages are generated with the Distutils standard package, which is already shipped with your Python distribution. The `setup.py` file is the installer script that contains the package metadata and the list of files to include.

You can find more information on Distutils installer scripts here.

1.4.3 Makefile usage

The `Makefile` is run using the "make" command. These are the commands supported by our makefile:

Building the project

- **make all**

  Generates the all documentation and builds all the packages.

- **make clean**

  Removes all files and directories created by the other make commands.

Building each component

- **make doc**

  Generates only the documentation, in all supported formats but CHM (the HTML Help Workshop returns an error condition because of warnings, which would stop the make process).

- **make html**

  Generates only the documentation in HTML format.

- **make pdf**

  Generates only the documentation in PDF format.

- **make chm**

  Generates only the documentation in CHM format. Depends on the HTML documentation.

- **make dist**

  Builds only the distribution packages, in all supported formats, for the current platform and architecture.

- **make sdist**

  Builds only the source distribution package (that is, the *zip* file). This package contains the documentation and can later be used to install the module by uncompressing it and running *setup.py*.

- **make bdist**

  Builds only the Windows installer package (that is, the *exe* file) for the current platform and architecture. This package does not contain the documentation and cannot be manually extracted to search for individual files.

- **make py2exe**

  Builds standalone binaries for the tools, using py2exe. This step is optional.

- **make upx**

  Compresses the executables generated wth **py2exe**, using UPX. This step is optional.

1.4.4 Directory structure

This is the directory structure expected for the makefile and the install script to work.

Input directories

- **/examples**

  This folder contains the example scripts shipped with python-winappdbg. They're the same examples found in the project wiki pages. It's included **only** in the source distribution package.

- **/tools**

  This folder contains the utility scripts shipped with python-winappdbg. It's included in both the source distribution package and the Windows installer.

- **/winappdbg**

  This folder contains the winappdbg module files. It's included in both the source distribution package and the Windows installer.

Output directories

- **/build**

  Temporary folder created when building the source distribution and Windows installer. You can safely delete this.

- **/dist**

  This is where the source distribution and the Windows installer files are stored.

- **/dist/py2exe**

  This is where the standalone binary files are stored. It's only created when the **make py2exe** command is run.

- **/html**

  This is where the autogenerated documentation files are stored, in HTML format. If you compile this documentation into a .CHM file it'll also be stored here.

- **/pdf**

  This is where the autogenerated documentation files are stored, in PDF and PostScript format.