WinAppDbg Documentation

Release 1.5

Mario Vilas

December 20, 2013

Contents

CHAPTER 1

Introduction

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 tools* are shipped and can be used for this purposes.

Current features also include disassembling x86/x64 native code, debugging multiple processes simultaneously and produce a detailed log of application crashes, useful for fuzzing and automated testing.

Here is a list of software projects that use *WinAppDbg* in alphabetical order:

- Heappie! is a heap analysis tool geared towards exploit writing. It allows you to visualize the heap layout during the heap spray or heap massaging stage in your exploits. The original version uses vtrace but here's a patch to use WinAppDbg instead. The patch also adds 64 bit support.
- PyPeElf is an open source GUI executable file analyzer for Windows and Linux released under the BSD license. You can download it here and there's also a blog.
- python-haystack is a heap analysis framework, focused on classic C structure matching. The basic functionality is to search in a process' memory maps for a specific C Structures. The extended reverse engineering functionality aims at reversing structures from memory/heap analysis.
- SRS is a tool to spy on registry API calls made by the program of your choice.
- Tracer.py is a "small and cute" execution tracer, in the words of it's author :) to aid in differential debugging.
- unpack.py is a script using WinAppDbg to automatically unpack malware, written by Karl Denton.

And this is a list of some alternatives to *WinAppDbg* in case it doesn't suit your needs, also in alphabetical order:

- ImmLib is a Python library to integrate your custom scripts into *Immunity Debugger*. It can only function inside the debugger, but it's the best solution if you aim at writing plugins for that debugger instead of standalone tools.
- Kenshoto's vtrace debugger is a full fledged multiplatform debugger written in Python, and a personal favorite of mine. I took a few ideas from it when designing *WinAppDbg* and, while I feel mine is more complete when it comes to Windows-specific features, this is what I'd definitely recommend for multi-OS projects. See also the community branch.

- OllyPython is an *OllyDbg* plugin that integrates a Python debugger. Naturally it only works within OllyDbg and is not suitable for standalone projects.
- PyDbg is another debugging library for Python that is part of the *Paimei* framework, but may work separately as well. It works on Windows and OSX. It predates *WinAppDbg* by quite some time but it's also been unmaintained for long, and it only works in Python versions 2.4 and 2.5. A newer branch called PyDbg64 implements 64 bit support for both platforms.
- PyDbgEng is a similar project to *WinAppDbg*, but it uses the Microsoft Debug Engine as a back end while *WinAppDbg* uses only bare Win32 API calls. The advantage of this approach is the ability to support kernel debugging, which is not allowed by the Win32 API alone. The disadvantage is having to install the Windows SDK/WDK to the machine where you run your scripts (or at least the components needed for debugging). See also the Buggery project which is based on *PyDbgEng*.
- PyDbgExt is the reverse of *PyDbgEng*: instead of instancing the *Microsoft Debug Engine* from a Python interpreter, it embeds a Python interpreter inside the Microsoft debugger *WinDbg*.
- pygdb is a simple wrapper on the GNU debugger that provides a GTK interface to it. Works in Linux and OSX.
- PyKd is like *PyDbgEng* and *PyDbgExt* combined into one - it can be both used from within the debugger and a standalone Python interpreter. Being a younger project it's still in alpha state, but looks very promising!
- PyMem is a memory instrumentation library written in Python for Windows. It provides a subset of the functionality found in *WinAppDbg*, but if you're developing a tool that only needs to manipulate a process memory you may find it convenient to support both backends and leave the choice to the user.
- python-ptrace is another debugger library for Python with the same goals as *WinAppDbg*. Here the approach used was to call the ptrace syscall, so naturally it works only on POSIX systems (BSD, Linux, maybe OSX). If Kenshoto's vtrace is not an option you could try combining this with *WinAppDbg* to implement a multiplatform tool.
- PythonGdb is an embedded Python interpreter for the GNU debugger. It's already included in GDB 7.
- Radare is a console based multiplatform disassembler, debugger and reverse engineering framework. Python is among the languages supported for plugins and scripting.
- Universal Hooker (uhooker) is a Python library to implement function hooks in other processes. While its functionality overlaps with some of *WinAppDbg*, the hooks implementation of *uhooker* is superior. Unfortunately the last update was in 2007. :(

See also the wonderful Python Arsenal for RE for an up to date reference of security related Python tools, available online and in PDF format.

CHAPTER 2

Programming Guide

2.1 Downloading and installing

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

2.1.1 Latest version

The latest version is **1.5** (20 Dec 2013). There are different installers depending on your Python version (32 and 64 bits) and the source code can be installer via the setup.py script. All of them work in all supported Windows versions - by 32-bit or 64-bit it means the Python interpreter, not the OS itself.

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

Installer packages

- winappdbg-1.5.win32.msi - All supported 32-bit Python versions
- winappdbg-1.5.win-amd64.msi - All supported 64-bit Python versions

Source code

- winappdbg-1.5.zip - Manual install (setup.py)

The programming manuals can be consulted online, but they're also available for download:

Windows Help Files

- winappdbg-tutorial-1.5.chm - Introduction and tutorials
- winappdbg-reference-1.5.chm - Complete reference material

HTML format

- winappdbg-tutorial-1.5.tar.bz2 - Introduction and tutorials
- winappdbg-reference-1.5.tar.bz2 - Complete reference material

PDF format (suitable for printing)

- winappdbg-tutorial-1.5.pdf - Introduction and tutorials
- winappdbg-reference-1.5.pdf - Complete reference material

2.1.2 Older versions

Older versions are still available for download as well:

Version **1.4** *(10 Dec 2010)*	• Downloads • Online help • Tutorial
Version **1.3** *(12 Feb 2010)*	• Downloads • Online help
Version **1.2** *(16 Jun 2009)*	• Downloads • Online help
Version **1.1** *(18 May 2009)*	• Downloads • Online help
Version **1.0** *(22 Apr 2009)*	• Downloads • Online help

2.1.3 Dependencies

Naturally you need the Python interpreter. It's recommended to use Python 2.7. You'll have to install the 32 bit VM to debug 32 bit targets and the 64 bit VM for 64 bit targets. Both VMs can be installed on the same machine.

If you're still using Python 2.5 64-bit, you'll need to install ctypes as well. This is needed to interface with the Win32 API, and *WinAppDbg* won't work without it. Newer versions of Python already have this module.

The following packages provide extra features and performance improvements, they are very recommended but not mandatory.

Disassembler

WinAppDbg itself doesn't come with a disassembler, but all of the following are compatible. *WinAppDbg* will pick the most suitable one automatically when needed, but you can also decide which one to use.

- The diStorm disassembler by Gil Dabah:
  - Distorm 3.3, 32 bits (GPL v3)
  - Distorm 3.3, 64 bits (GPL v3)
  - Distorm 1.7.30, 32 bits (old version, BSD license)
  - Distorm 1.7.30, 64 bits (old version, BSD license)
- The BeaEngine disassembler by BeatriX:
  - BeaEngine 3.1.0, 32 bits
  - BeaEngine 3.1.0, 64 bits
- The Capstone disassembler by Nguyen Anh Quynh:
  - Capstone 1.0 bindings for Python 2.6 (Windows, 32 bits)

  - Capstone 1.0 bindings for Python 2.6 (Windows, 64 bits)
  - Capstone 1.0 bindings for Python 2.7 (Windows, 32 bits)
  - Capstone 1.0 bindings for Python 2.7 (Windows, 64 bits)
- The PyDasm Python bindings to libdasm by Ero Carrera:
  - PyDasm 1.5, 32 and 64 bits
- The Libdisassemble module from Immunity:
  - Libdisassemble 2.0, 32 bits
  - Libdisassemble 2.0, 64 bits
  - Libdisassemble 2.0, source install

Database storage

The SQL Alchemy ORM module gives *WinAppDbg* the ability to use a SQL database to store and find crash dumps. Most major database engines are supported.

Other goodies

With the Python specializing compiler, Psyco, *WinAppDbg* will experience a performance gain just by installing it, no additional steps are needed. You can download the sources and some old precompiled binaries from the official site and newer but unofficial builds from Voidspace.

Also 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.

2.1.4 Install

Simply run the **Windows installer** package and follow the wizard.

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

```
install.bat
```

You can also install WinAppDbg (stable versions only) from the Cheese Shop using any of the compatible **package managers**:

- PIP Installs Python

  ```
  pip install winappdbg
  ```

- PyPM (only when using ActivePython)
- Easy Install (formerly from Setuptools, now from Distribute)

  ```
  easy_install winappdbg
  ```

- Python Package Manager (it's a GUI installer)

2.1.5 Support

Minimim requirements:

- **Windows XP**
- **Python 2.5**

Recommended platform:

- **Windows 7**
- **Python 2.7**

It might work, but was not tested, under *Windows 2000*, *Wine* and *ReactOS*, and some bugs and missing features are to be expected in these platforms (mainly due to missing APIs).

Python 3 support was experimental up to *WinAppDbg 1.4*, and was dropped with *WinAppDbg 1.5*. There are currently no plans to support Python 3 in the near future - backwards compatibility would be broken and plenty of code would need to be refactored just to port it.

While there are still some issues that need ironing out, it may be worth trying out faster Python interpreters such as PyPy and IronPython.

If you find a bug or have a feature suggestion, don't hesitate to send an email to the 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. **Untested** means that though no testing was performed it should probably work. **Experimental** means it's not expected to work and although it might, you can encounter many bugs.

Python interpreters

Version	Status	Notes
CPython 2.4 and earlier	*not supported*	Use an *older version* of WinAppDbg in this case.
CPython 2.5 through 2.7	**full**	
CPython 3.0 and newer	*not supported*	Planned for WinAppDbg 2.0.
PyPy 1.4 and earlier	*not supported*	It doesn't seem to be available for download any more...
PyPy 1.5 and 1.6	*experimental*	The sqlite3 dll is missing, after you fix that it should be the same as newer versions.
PyPy 1.7 and newer	*experimental*	Some compatibility issues need fixing.
IronPython 2.0 and newer	*experimental*	Some compatibility issues need fixing.
Jython 2.5 and earlier	*not supported*	Support for ctypes is incomplete in this platform.

Operating systems

Version	Status	Notes
Windows 2000 and older	*not supported*	Some required Win32 API functions didn't exist yet.
Windows XP	**full**	
Windows Server 2003	**full**	
Windows Server 2003 R2	**full**	
Windows Vista	**full**	
Windows 7	**full**	
Windows Server 2008	**full**	
Windows Server 2008 R2	**full**	
Windows 8	*untested*	Probably similar to Windows 7.
Windows Server 2012	*untested*	Probably similar to Windows Server 2008 R2.
ReactOS	*untested*	Probably similar to Windows 2000.
Linux (using Wine 1.2)	*untested*	Reported to work on Ubuntu.
Linux (using Wine 1.3)	*untested*	Reported to work on Ubuntu.
Windows + Cygwin	*not supported*	Ctypes under Cygwin doesn't fully support calling Win32 API functions.
Windows Phone	*not supported*	Planned for WinAppDbg 2.0.

Architectures

Version	Status	Notes
Intel (32 bits)	**full**	
Intel (64 bits)	**full**	
ARM	*not supported*	Planned for WinAppDbg 2.0.

2.1.6 Known issues

- Python strings default encoding is 'ascii' since Python 2.5. While I did my best to prevent encoding errors when manipulating binary data, I recommend setting the default to 'latin-1' (ISO 8859-1) instead. You can do this by adding a sitecustomize.py script to your Python installation.
- Step-on-branch mode stopped working since Windows Vista. This is due to a change in the Windows kernel. The next major version of WinAppDbg (2.0) will support this.
- Debugging 32 bit processes from a 64 bit Python VM does not work very well. Debugging 64 bit processes from a 32 bit Python VM does not work at all. This is in part because the Win32 API makes it difficult, but there's also a design problem in WinAppDbg: most of the C struct definitions change from 32 to 64 bits and there's currently no support for having both definitions at the same time. This will change with WinAppDbg 2.0 too.
- Setting hardware breakpoints in the main thread before the process has finished initializing does not work. This is not supported by the Windows API itself, and is not a limitation of WinAppDbg. Future versions of WinAppDbg will try to detect this error and warn about it.

2.1.7 License

This software 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.

2.2 Instrumentation

You can implement process instrumentation in your Python scripts by using the provided set of classes: *System*, *Process*, *Thread*, *Module* and *Window*. 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 threads and modules, containing **Thread** and **Module** objects.

System objects also contain **Window** objects, representing the windows in the current desktop.

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

2.2.1 The System class

The **System** class groups functionality that lets you instrument some global aspects of the machine where you installed *WinAppDbg*. It also behaves like a snapshot of the running processes. It can enumerate processes and perform operations on a batch of processes.

Example #1: knowing on which platform we're running

Download

```
from winappdbg import System, version

# Show the Windows version and the current architecture.
print "WinAppDbg %s" % version
print "Running on %s for the %s architecture." % (System.os, System.arch)
if System.wow64:
    print "Running in 32 bit emulation mode."
print "From this Python VM we can attach to %d-bit processes." % System.bits
```

Example #2: enumerating running processes

Download

```
from winappdbg import System

# 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 #3: 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.
process = system.start_process( command_line ) # see the docs for more options

# Show info on the new process.
print "Started process %d (%d bits)" % ( process.get_pid(), process.get_bits() )
```

The *System* class has many more features, so we'll be coming back to it later on in the tutorial.

2.2.2 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 #4: 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:"
    bits = process.get_bits()
    for module in process.iter_modules():
        print "\t%s\t%s" % (
            HexDump.address( module.get_base(), bits ),
            module.get_filename()
        )
```

Example #5: 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 #6: 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 )

    # You can also change the process memory.
    # process.write( address, "example data" )

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

Example #7: getting the command line for a process

Download

```
from winappdbg import Process

def show_command_line( pid ):

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

    # Print the process command line.
    print process.get_command_line()

    # The same thing could be done with the environment variables.
    #import pprint
    #pprint.pprint( process.get_environment() )
```

Example #8: getting the environment variables for a process

Download

```
from winappdbg import Process

def show_environment( pid ):
    # Instance a Process object.
    process = Process( pid )

    # Get its environment variables.
    environment = process.get_environment()

    # Print the environment variables.
    for variable, value in sorted( environment.items() ):
        print "%s=%s" % (variable, value)
```

Example #9: 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 #10: getting the process memory map

Download

```
from winappdbg import win32, Process, HexDump

def print_memory_map( pid ):

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

    # Find out if it's a 32 or 64 bit process.
    bits = process.get_bits()

    # 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, bits)
        RegionSize  = HexDump.address(mbi.RegionSize,  bits)
        # 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 )
```

Example #11: searching the process memory

Download

```
from winappdbg import Process, HexDump

def memory_search( pid, bytes ):

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

    # Search for the string in the process memory.
    for address in process.search_bytes( bytes ):

        # Print the memory address where it was found.
        print HexDump.address( address )

    # You could also use process.search_regexp to use regular expressions,
    # or process.search_text for Unicode strings,
    # or process.search_hexa for raw bytes represented in hexa.
```

Example #12: dumping ASCII strings from the process memory

Download

```
from winappdbg import Process, HexDump

def strings( pid ):

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

    # For each ASCII string found in the process memory...
    for address, size, data in process.strings():

        # Print the string and the memory address where it was found.
        print "%s: %s" % ( HexDump.address(address), data )
```

2.2.3 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 #13: 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.iter_threads():

        # 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.iter_threads():

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

Example #14: 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 #15: 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 disassembled code.
    print
    print CrashDump.dump_code( code, eip ),
```

2.2.4 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 #16: 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)
```

2.2.5 The Window class

A **Window** object lets you manipulate any window in the current desktop. You can get a *Window* instance by querying a *System* object.

Example #17: enumerate the top-level windows

Download

```
from winappdbg import System, HexDump

# Create a system snaphot.
system = System()

# Now we can enumerate the top-level windows.
for window in system.get_windows():
    handle  = HexDump.integer( window.get_handle() )
    caption = window.get_text()
    if caption is not None:
        print "%s:\t%s" % ( handle, caption )
```

Example #18: minimize all top-level windows

Download

```
from winappdbg import System, HexDump

# Create a system snaphot.
system = System()

# Enumerate the top-level windows.
for window in system.get_windows():

    # Minimize the window.
    if not window.is_minimized():
        window.minimize()

    # You could also maximize, restore, show, hide, enable and disable.
    # For example:
    #
    # if window.is_maximized():
    #     window.restore()
    #
    # if not window.is_visible():
    #     window.show()
    #
    # if not window.is_disabled():
    #     window.enable()
    #
    # ...and so on.
```

Example #19: traverse the windows tree

Download

```
from winappdbg import System, HexDump


def show_window_tree( window, indent = 0 ):

    # Show this window's handle and caption.
    # Use some ASCII art to show the layout. :)
    handle  = HexDump.integer( window.get_handle() )
    caption = window.get_text()
    line = ""
    if indent > 0:
        print "|    " * indent
        line = "|    " * (indent - 1) + "|---"
    else:
        print "|"
    if caption is not None:
        line += handle + ": " + caption
    else:
        line += handle
    print line

    # Recursively show the child windows.
    for child in window.get_children():
        show_window_tree( child, indent + 1 )


def main():

    # Create a system snaphot.
    system = System()

    # Get the Desktop window.
    root = system.get_desktop_window()

    # Now show the window tree.
    show_window_tree(root)

    # You can also ge the tree as a Python dictionary:
    # tree = root.get_tree()
    # print tree
```

Example #20: get windows by screen position

Download

```
from winappdbg import System, HexDump
import sys

try:

    # Get the coordinates from the command line.
    x = int( sys.argv[1] )
    y = int( sys.argv[2] )

    # Get the window at the requested position.
    window   = System.get_window_at( x, y )

    # Get the window coordinates.
    rect     = window.get_screen_rect()
    position = (rect.left, rect.top, rect.right, rect.bottom)
    size     = (rect.right - rect.left, rect.bottom - rect.top)

    # Print the window information.
    print "Handle:   %s" % HexDump.integer( window.get_handle() )
    print "Caption:  %s" % window.text
    print "Class:    %s" % window.classname
    print "Style:    %s" % HexDump.integer( window.style )
    print "ExStyle:  %s" % HexDump.integer( window.exstyle )
    print "Position: (%i, %i) - (%i, %i)" % position
    print "Size:     (%i, %i)" % size

except WindowsError:
    print "No window at those coordinates!"
```

Example #21: find windows by class and caption

Download

```
from winappdbg import System, HexDump
import sys


def find_window():

    # If two arguments are given, the first is the classname
    # and the second is the caption text.
    if len(sys.argv) > 2:
        classname = sys.argv[1]
        caption   = sys.argv[2]
        if not classname:
            classname = None
        if not caption:
            caption   = None
        window = System.find_window( classname, caption )

    # If only one argument is given, try the caption text, then the classname.
    else:
        try:
            window = System.find_window( windowName = sys.argv[1] )
        except WindowsError:
            window = System.find_window( className = sys.argv[1] )

    return window


def show_window( window ):

    # Get the window coordinates.
    rect     = window.get_screen_rect()
    position = (rect.left, rect.top, rect.right, rect.bottom)
    size     = (rect.right - rect.left, rect.bottom - rect.top)

    # Print the window information.
    print "Handle:   %s" % HexDump.integer( window.get_handle() )
    print "Caption:  %s" % window.text
    print "Class:    %s" % window.classname
    print "Style:    %s" % HexDump.integer( window.style )
    print "ExStyle:  %s" % HexDump.integer( window.exstyle )
    print "Position: (%i, %i) - (%i, %i)" % position
    print "Size:     (%i, %i)" % size


def main():
    try:
        show_window( find_window() )
    except WindowsError:
        print "No window found!"
```

Example #22: kill a program using its window

Download

```
def user_confirmed():
    print
    answer = raw_input( "Are you sure you want to kill this program? (y/N):" )
    answer = answer.strip().upper()
    return answer.startswith("Y")


def main():

    # Find the window.
    try:
        window = find_window()
    except WindowsError:
        print "No window found!"
        return

    # Show the window info to the user.
    show_window( window )

    # Ask the user for confirmation.
    if user_confirmed():

        # Kill the program.
        window.kill()
```

2.2.6 Back to the System class

As promised, we're back on the **System** class to see more of its features. We'll now see how to access the Windows Registry and work with system services.

Example #23: exporting a Registry key

Download

```
import struct

from winappdbg import System, win32

#RegistryEditorVersion = "REGEDIT4"  # for Windows 95
RegistryEditorVersion = "Windows Registry Editor Version 5.00"

# Helper function to serialize data to hexadecimal format.
def reg_hexa(value, type):
    return "hex(%x):%s" % (type, ",".join( ["%.2x" % ord(x) for x in value] ))

# Registry export function.
def reg_export( reg_path, filename ):

    # Queue of registry keys to visit.
    queue = []

    # Get the registry key the user requested.
    key = System.registry[ reg_path ]

    # Add it to the queue.
    queue.append( key )
```

```
# Open the output file.
with open(filename, "wb") as output:

    # Write the file format header.
    output.write( "%s\r\n" % RegistryEditorVersion )

    # For each registry key in the queue...
    while queue:
        key = queue.pop()

        # Write the key path.
        output.write( "\r\n[%s]\r\n" % key.path )

        # If there's a default value, write it.
        default = str(key)
        if default:
            output.write( "@=\"%s\"\r\n" % default )

        # For each value in the key...
        for name, value in key.iteritems():

            # Skip the default value since we already wrote it.
            if not name:
                continue

            # Serialize the name.
            s_name = "\"%s\"" % name

            # Serialize the value.
            t_value = key.get_value_type(name)
            if t_value == win32.REG_SZ and type(value) == str:
                s_value = "\"%s\"" % value.replace("\"", "\\\"")
            elif t_value == win32.REG_DWORD:
                s_value = "dword:%.8X" % value
            else:
                if t_value == win32.REG_QWORD:
                    value = struct.pack("<Q", value)
                elif t_value == win32.REG_DWORD:
                    value = struct.pack("<L", value)
                elif t_value == win32.REG_DWORD_BIG_ENDIAN:
                    value = struct.pack(">L", value)
                elif t_value == win32.REG_MULTI_SZ:
                    if not value:
                        value = ""
                    elif type(value) == str:
                        value = "\0".join(value)
                    else:
                        value = u"\0".join(value)
                if type(value) == str:
                    s_value = reg_hexa(value, t_value)
                else:
                    s_value = reg_hexa(value.encode("UTF-16"), t_value)

            # Write the name and value.
            output.write( "%s=%s\r\n" % (s_name, s_value) )
```

Example #24: searching the Registry

Download

```
from winappdbg import System, Color

def reg_search( search ):

    # Show the user what we're searching for.
    print "Searching for: %r" % search

    # For each Registry key...
    for path in System.registry.iterkeys():

        # Try to open the key. On error skip it.
        try:
            key = System.registry[ path ]
        except Exception:
            continue

        # Get the default value. On error skip it.
        try:
            default = str(key)
        except KeyError:
            default = ""
        except Exception:
            continue

        # Does the default value match?
        if search in default:
            text = "%s\\@: %s" % ( path, default )
            highlight( search, text )

        # Does the key match?
        elif search in path[ path.rfind("\\") : ]:
            highlight( search, path )

        # For each Registry value...
        for name in key.iterkeys():

            # Try to get the value. On error ignore it.
            try:
                value = key[name]
            except Exception:
                value = ""

            # Registry values can be of many data types.
            # For this search we need to force all values to be strings.
            if type(value) not in (str, unicode):
                value = str(value)

            # Do the name or value match?
            if search in name or search in value:
                text = "%s\\%s: %r" % ( path, name, value )
                highlight( search, text )

# Helper function to print text with a highlighted search string.
def highlight( search, text ):
    if can_highlight:
        try:
            Color.default()
            p = 0
            t = len( text )
            s = len( search )
            while p < t:
                q = text.find( search )
                if q < p:
                    q = t
                sys.stdout.write( text[ p : q ] )
                Color.red()
                Color.light()
                sys.stdout.write( text[ q : q + s ] )
                Color.default()
                sys.stdout.write("\r\n")
                p = q + s
        finally:
            Color.default()
    else:
        print text

# Determine if the output is a console or a file.
# Trying to use colors fails if the output is not the console.
can_highlight = Color.can_use_colors()
```

Example #25: listing system services

Download

```
from winappdbg import System, win32

def show_services():

    # Get the list of services.
    services = System.get_services()

    # You could get only the running services instead.
    # services = System.get_active_services()

    # For each service descriptor...
    for descriptor in services:

        # Print the service information, the easy way.
        # print str(descriptor)

        # You can also do it the hard way, accessing its members.
        print "Service name: %s" % descriptor.ServiceName
        print "Display name: %s" % descriptor.DisplayName
        if   descriptor.ServiceType & win32.SERVICE_INTERACTIVE_PROCESS:
            print "Service type: Win32 GUI"
        elif descriptor.ServiceType & win32.SERVICE_WIN32:
            print "Service type: Win32"
        elif descriptor.ServiceType & win32.SERVICE_DRIVER:
            print "Service type: Driver"
        if   descriptor.CurrentState == win32.SERVICE_CONTINUE_PENDING:
            print "Current status: RESTARTING..."
        elif descriptor.CurrentState == win32.SERVICE_PAUSE_PENDING:
            print "Current status: PAUSING..."
        elif descriptor.CurrentState == win32.SERVICE_PAUSED:
            print "Current status: PAUSED"
        elif descriptor.CurrentState == win32.SERVICE_RUNNING:
            print "Current status: RUNNING"
        elif descriptor.CurrentState == win32.SERVICE_START_PENDING:
            print "Current status: STARTING..."
        elif descriptor.CurrentState == win32.SERVICE_STOP_PENDING:
            print "Current status: STOPPING..."
        elif descriptor.CurrentState == win32.SERVICE_STOPPED:
            print "Current status: STOPPED"
        print

# When invoked from the command line,
# call the show_services() function.
```

Example #26: stopping and starting a system service

Download

```
from time import sleep
from winappdbg import System, win32


# Function that restarts a service.
# Requires UAC elevation in Windows Vista and above.
def restart_service( service ):
    try:

        # Get the display name.
        try:
            display_name = System.get_service_display_name( service )
        except WindowsError:
            display_name = service

        # Get the service descriptor.
        descriptor = System.get_service( service )

        # Is the service running?
        if descriptor.CurrentState != win32.SERVICE_STOPPED:

            # Tell the service to stop.
            print "Stopping service \"%s\"..." % display_name
            System.stop_service( service )

            # Wait for the service to stop.
            wait_for_service( service, win32.SERVICE_STOP_PENDING )
            print "Service stopped successfully."

        # Tell the service to start.
        print "Starting service \"%s\"..." % display_name
        System.start_service( service )

        # Wait for the service to start.
        wait_for_service( service, win32.SERVICE_START_PENDING )
        print "Service started successfully."
        # Show the new process ID.
        # This feature requires Windows XP and above.
        descriptor = System.get_service( service )
        try:
            print "New process ID is: %d" % descriptor.ProcessId
        except AttributeError:
            pass

    # On error, show an error message.
    except WindowsError, e:
        print str(e)


# Helper function to wait for the service to change its state.
def wait_for_service( service, wait_state, timeout = 20 ):
    descriptor = System.get_service( service )
    while descriptor.CurrentState == wait_state:
        timeout -= 1
        if timeout <= 0:
            raise RuntimeException( "Error: timed out." )
        sleep( 0.5 )
        descriptor = System.get_service( service )
```

2.3 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*.

2.3.1 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.

When you're finished using the *Debug* object, you must either call its *stop()* method from a *finally* block, or put the *Debug* object inside a *with* statement.

Note: In previous examples we have used a **System.request_debug_privileges()** call to get debug privileges. When using the *Debug* class we don't need to do that - it's taken care of automatically in the constructor.

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

Download

```
from winappdbg import Debug

import sys

# Instance a Debug object.
```

```python
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

```python
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

```python
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 the debugged process when the debugger is closed

Download

```
from winappdbg import Debug

import sys

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

# The user can stop debugging with Control-C.
try:
    print "Hit Control-C to stop debugging..."

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

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

# If the user presses Control-C...
except KeyboardInterrupt:
    print "Interrupted by user."

    # Stop debugging. This kills all debugged processes.
    debug.stop()
```

2.3.2 The interactive debugger

The *Debug* class also contains an implementation of a simple console debugger. It can come in handy when testing your scripts, or to manually handle unexpected situations.

Example #5: running an interactive debugger session

Download

```
from winappdbg import Debug, HexDump

def simple_debugger( argv ):
    # Instance a Debug object.
    debug = Debug()
    try:

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

        # Launch the interactive debugger.
        debug.interactive()

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

2.3.3 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.

Every *Event* object has the following set of common methods to get information from them:

- **get_event_name**: Returns the name of the event.
- **get_event_description**: Returns a user-friendly description of the event.
- **get_event_code**: Returns the event code constant, as defined by the Win32 API.
- **get_pid**: Returns the ID of the process where the event occurred.
- **get_tid**: Returns the ID of the thread where the event occurred.
- **get_process**: Returns the Process object.
- **get_thread**: Returns the Thread object.

Then depending on the event type, you can get more information that's specific to each type.

- **get_filename:** Returns the filename of the EXE or DLL.

  Applicable events: Process creation and destruction, DLL library load and unload.
- **get_exit_code:** Returns the exit code of the process or thread.

  Applicable events: Process and thread destruction.
- **get_exception_name:** Returns the Win32 API constant name for the exception code.

  Applicable events: Exceptions.
- **get_exception_code:** Returns the Win32 API constant value for the exception code.

  Applicable events: Exceptions.
- **get_exception_address:** Returns the memory address where the exception has occurred.

  For exceptions not involving memory operations, the current execution pointer is returned.

  Applicable events: Exceptions.
- **is_system_defined_exception:** Returns *True* if the exception was caused by the operating system rather than the application code.

Most notably, one such exception is always raised when attaching to a process, and then running a process from the debugger (right after process initialization is complete).

Applicable events: Exceptions.

- **is_first_chance:** If *True*, the exception hasn't been passed yet to the exception handlers of the debuggee.

  If *False*, the exception was passed to the exception handlers but none of them could handle it.

  Applicable events: Exceptions.

- ***is_nested**:** If *True*, the exception was raised when handing at least one more exception.

  Many exceptions can be nested that way. Call **get_nexted_exceptions** to get a list of those nested exceptions.

  Applicable events: Exceptions.

- **get_fault_address:** Returns the memory address where the invalid access has occurred.

  Applicable events: Exceptions caused by invalid memory access.

Example #6: handling debug events

Download

```
from winappdbg import Debug, HexDump, win32

def my_event_handler( event ):

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

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

    # Find out if it's a 32 or 64 bit process.
    bits = event.get_process().get_bits()

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

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

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

    # If the event is an exception...
    if code == win32.EXCEPTION_DEBUG_EVENT:

        # Get the exception user-friendly description.
        name = event.get_exception_description()

        # Get the exception code.
        code = event.get_exception_code()

        # Get the address where the exception occurred.
        try:
            address = event.get_fault_address()
        except NotImplementedError:
            address = event.get_exception_address()

    # If the event is a process creation or destruction,
    # or a DLL being loaded or unloaded...
    elif code in ( win32.CREATE_PROCESS_DEBUG_EVENT,
                   win32.EXIT_PROCESS_DEBUG_EVENT,
                   win32.LOAD_DLL_DEBUG_EVENT,
                   win32.UNLOAD_DLL_DEBUG_EVENT ):

        # Get the filename.
        filename = event.get_filename()
        if filename:
            name = "%s [%s]" % ( name, filename )

    # Show a descriptive message to the user.
    print "-" * 79
    format_string = "%s (0x%s) at address 0x%s, process %d, thread %d"
    message = format_string % ( name,
                                HexDump.integer(code, bits),
                                HexDump.address(address, bits),
                                pid,
                                tid )
    print message

def simple_debugger( argv ):

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

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

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

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

2.3.4 The Crash and CrashDAO classes

Crashes are exceptions a program can't recover from (also known as second-chance exceptions or last chance exceptions). A **crash dump** is a collection of information from a crash in a program that can (hopefully!) help you reproduce or fix the bug that caused it in the first place.

WinAppDbg provides the *Crash* class to generate and manipulate crash dumps. When instancing a *Crash* object only the most basic information is collected, you have to call the *fetch_extra_data* method to collect more data. This lets you control which information to gather and when - for example you may be interested in gathering more information only under certain conditions, or for certain kinds of exceptions.

Crash objects also support *heuristic signatures* that 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.

Now, the next step would be storing the crash dump somewhere for later examination. The most crude way to do this is using the standard pickle module, or similar modules like cerealizer. This is easy and guaranteed to work, but not very comfortable! Crash dumps stored that way are hard to read outside Python.

A more flexible way to store crash dumps is using the *CrashDAO* class. It uses SQLAlchemy to connect to any supported SQL database, create the required tables if needed, and store multiple crash dumps in it. This is the preferred method, since it's easier to access and manipulate the information outside Python, and you can store crashes from multiple machines into the same database.

Old versions of **WinAppDbg** (1.4 and older) supported DBM databases through the *CrashContainer* class, SQLite databases with the *CrashTable* class, and SQL Server databases with the *CrashTableMSSQL* class. They are now deprecated and, while still present for backwards compatibility (for the time being) its use is not recommended.

Example #7: saving crash dumps

Download

```
from sys import exit

from winappdbg import win32, Debug, HexDump, Crash

try:
    from winappdbg import CrashDAO
except ImportError:
    raise ImportError("Error: SQLAlchemy is not installed!")

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.
    bits = event.get_process().get_bits()
    format_string = "%s (%s) at address %s, process %d, thread %d"
    message = format_string % ( name,
                                HexDump.integer(code, bits),
                                HexDump.address(pc, bits),
                                pid,
                                tid )
    print message

    # If the event is a crash...
    if code == win32.EXCEPTION_DEBUG_EVENT and event.is_last_chance():
        print "Crash detected, storing crash dump in database..."

        # Generate a minimal crash dump.
        crash = Crash( event )
        # You can turn it into a full crash dump (recommended).
        # crash.fetch_extra_data( event, takeMemorySnapshot = 0 ) # no memory dump
        # crash.fetch_extra_data( event, takeMemorySnapshot = 1 ) # small memory dump
        crash.fetch_extra_data( event, takeMemorySnapshot = 2 ) # full memory dump

        # Connect to the database. You can use any URL supported by SQLAlchemy.
        # For more details see the reference documentation.
        dao = CrashDAO( "sqlite:///crashes.sqlite" )
        #dao = CrashDAO( "mysql+MySQLdb://root:toor@localhost/crashes" )

        # Store the crash dump in the database.
        dao.add( crash )

        # If you do this instead, heuristics are used to detect duplicated
        # crashes so they aren't added to the database.
        # dao.add( crash, allow_duplicates = False )

        # You can also launch the interactive debugger from here. Try it! :)
        # event.debug.interactive()

        # Kill the process.
        event.get_process().kill()

def simple_debugger( argv ):

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

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

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

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

2.3.5 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:

- **create_process**:

  What does it mean?: The debugger has attached to a new process.

  When is it received?: 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**:

What does it mean?: A debugee process has finished executing.

When is it received?: When a process terminates by itself or when the *Process.kill* method is called.

- **create_thread**:

  What does it mean?: A debugee process has started a new thread.

  When is it received?: When the process creates a new thread or when the *Process.start_thread* method is called.

- **exit_thread**:

  What does it mean?: A thread in a debugee process has finished executing.

  When is it received?: When a thread terminates by itself or when the *Thread.kill* method is called.

- **load_dll**:

  What does it mean?: A module in a debugee process was loaded.

  When is it received?: When a process loads a DLL module by itself or when the *Process.inject_dll* method is called.

- **unload_dll**:

  What does it mean?: A module in a debugee process was unloaded.

  When is it received?: When a process unloads a DLL module by itself.

- **exception**:

  What does it mean?: An exception was raised by the debugee.

  When is it received?: When a hardware fault is triggered or when the process calls RaiseException().

- **output_string**:

  What does it mean?: The debuggee has sent a debug string.

  When is it received?: When the process calls OutputDebugString().

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:

access_violation:

What does it mean?: An access violation exception was raised by the debugee.

When is it received?: When the debuggee tries to access invalid memory.

ms_vc_exception:

What does it mean?: A C++ exception was raised by the debugee.

When is it received?: When the debuggee calls RaiseException() with a custom exception code. This is what the implementation of throw() of the Visual Studio runtime does.

breakpoint:

What does it mean?: A breakpoint exception was raised by the debugee.

When is it received?: 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:

What does it mean?: A single step exception was raised by the debugee.

When is it received?: 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:

What does it mean?: A guard page exception was raised by the debugee.

When is it received?: 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 what parameters does each call take (using ctypes definitions). 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.

Note: One thing to be careful with when hooking API functions: all pointers should be declared as having the void type. Otherwise ctypes gets too "helpful" and tries to access the memory pointed to by them... and crashes, since those pointers only work in the debugged process.

Example #8: 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]
        bits   = event.get_process().get_bits()
        print "%s: %s" % ( HexDump.address(code[0], bits), code[2].lower() )


def simple_debugger( argv ):
    # Instance a Debug object using the "with" statement.
    # Note how we don't need to call "debug.stop()" anymore.
    with Debug( MyEventHandler(), bKillOnExit = True ) as debug:

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

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

Example #9: intercepting API calls

Download

```
from winappdbg.win32 import *


class MyEventHandler( EventHandler ):


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

        # Hooks for the kernel32 library.
        'kernel32.dll' : [

            #   Function            Parameters
            ( 'CreateFileA'     , (PVOID, DWORD, DWORD, PVOID, DWORD, DWORD, HANDLE) ),
            ( 'CreateFileW'     , (PVOID, DWORD, DWORD, PVOID, DWORD, DWORD, HANDLE) ),

        ],

        # Hooks for the advapi32 library.
        'advapi32.dll' : [

            #   Function            Parameters
            ( 'RegCreateKeyExA' , (HKEY, PVOID, DWORD, PVOID, DWORD, REGSAM, PVOID, PVOID, PVOID) ),
            ( 'RegCreateKeyExW' , (HKEY, PVOID, DWORD, PVOID, DWORD, REGSAM, PVOID, PVOID, PVOID) ),

        ],
    }


    # 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_reg_success( event, retval )

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

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

2.3.6 The EventSift class

If you're debugging more than one process at a time, keeping track of everything can be trickier. For that reason there's also a class called **EventSift**. You can wrap your *EventHandler* class with it to create a new *EventHandler* instance for each debugged process.

That way, your *EventHandler* can be written as if only a single process was being debugged, but you can attach to as many processes as you want. Each *EventHandler* will only "see" its own debugee.

Example #10: sifting events per process

Download

```
# This class was written assuming only one process is attached.
# If you used it directly it would break when attaching to another
# process, or when a child process is spawned.
class MyEventHandler (EventHandler):

    def create_process(self, event):
        self.first = True
        self.name = event.get_process().get_filename()
        print "Attached to %s" % self.name

    def breakpoint(self, event):
        if self.first:
            self.first = False
            print "First breakpoint reached at %s" % self.name

    def exit_process(self, event):
        print "Detached from %s" % self.name


# Now when debugging we use the EventForwarder to be able to work with
# multiple processes while keeping our code simple. :)
def simple_debugger():

    handler = EventSift(MyEventHandler)
    #handler = MyEventHandler()  # try uncommenting this line...
    with Debug(handler) as debug:
        debug.execl("calc.exe")
        debug.execl("notepad.exe")
        debug.execl("charmap.exe")
        debug.loop()
```

2.3.7 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 #11: setting a breakpoint

Download

```
# This function will be called when our breakpoint is hit.
def action_callback( event ):
    process = event.get_process()
    thread  = event.get_thread()

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

    # Get the return address of the call.
    address = 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, process.get_bits()), 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 #12: hooking a function

Download

```
from winappdbg.win32 import PVOID


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

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

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( 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" )
            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

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

            # 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, signature = signature)
```

Example #13: 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.
    bits = event.get_process().get_bits()
    if bits == 32:
        return_address, hModule, lpProcName = thread.read_stack_dwords( 3 )
    else:
        return_address = thread.read_stack_qwords( 1 )
        registers   = thread.get_context()
        hModule     = registers['Rcx']
        lpProcName = registers['Rdx']

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

    # Show a message to the user.
    message = "%s: GetProcAddress(%s, %r);"
    print message % (
        HexDump.address(return_address, bits),
        HexDump.address(hModule, bits),
        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 the registers).
    registers = event.get_thread().get_context()
    if event.get_process().get_bits() == 32:
        return_value = registers['Eax']
    else:
        return_value = registers['Rax']

    # 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 the registers).
    registers = event.get_thread().get_context()
    if event.get_process().get_bits() == 32:
        return_value = registers['Eax']
    else:
        return_value = registers['Rax']

    # 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 #14: watching a buffer

Download

```
class MyHook (object):

    # Keep record of the buffers we watch.
    def __init__(self):
        self.__watched  = dict()
        self.__previous = None


    # This function will be called when entering the hooked function.
    def entering( self, event, ra, hFile, lpBuffer, nNumberOfBytesToRead, lpNumberOfBytesRead, lpOver

        # Ignore calls using a NULL pointer.
        if not lpBuffer:
            return

        # Show a message to the user.
        print "\nReadFile:\n\tHandle %x\n\tExpected bytes: %d" % ( hFile, nNumberOfBytesToRead )

        # Stop watching the previous buffer.
        if self.__previous:
            event.debug.dont_watch_buffer( self.__previous )
            self.__previous = None

        # Remember the location of the buffer and its size.
        self.__watched[ event.get_tid() ] = ( lpBuffer, lpNumberOfBytesRead )


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

        # If the function call failed ignore it.
        if return_value == 0:
            print "\nReadFile:\n\tStatus: FAIL"
            return

        # Get the buffer location and size.
        tid     = event.get_tid()
        process = event.get_process()
        ( lpBuffer, lpNumberOfBytesRead ) = self.__watched[ tid ]
        del self.__watched[ tid ]
        # Watch the buffer for access.
        pid     = event.get_pid()
        address = lpBuffer
        size    = process.read_dword( lpNumberOfBytesRead )
        action  = self.accessed
        self.__previous = 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.
        #
        # self.__previous = event.debug.stalk_buffer( pid, address, size, action )

        # Show a message to the user.
        print "\nReadFile:\n\tStatus: SUCCESS\n\tRead bytes: %d" % size


    # 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 "%s: %s" % (
            HexDump.address(code[0], thread.get_bits()),
            code[2].lower()
        )


class MyEventHandler( EventHandler ):

    # Called on 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 the function to hook.
            address = module.resolve( "ReadFile" )

            # This is an approximated signature of the function.
            # Pointers must be void so ctypes doesn't try to read from them.
            signature = ( win32.HANDLE, win32.PVOID, win32.DWORD, win32.PVOID, win32.PVOID )

            # Hook the function.
            hook = MyHook()
            event.debug.hook_function( pid, address, hook.entering, hook.leaving, signature = signatu
```

2.3.8 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 Advanced Topics section of this document.

Example #15: 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 #16: 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 )
```

2.4 Helper classes and functions

WinAppDbg provides some helper classes and functions, mostly related to input and output, that can come in handy when reading input from users or writing debugging data.

2.4.1 Console output with colors

The functions from the **Color** static class allow your scripts to write colored text to the console.

Tipically you'll make a call to the **can_use_colors** function to determine if it's possible to write text with colors. This is necessary because color output only works with a real console - if the user has redirected the output to a file or a pipe, trying to use colors will cause an exception to be raised.

The following functions set the console text color:

- **black**
- **white**
- **red**
- **green**
- **blue**
- **cyan**
- **magenta**
- **yellow**

You can also combine the colors with the brightness settings using the **light** and **dark** functions, to get more variations on colors:

```
Color.red()
Color.light()
print "This is printed in light red."
Color.dark()
print "This is printed in dark red."
Color.blue()
print "This is printed in dark blue."
Color.light()
print "This is printed in light blue."
```

The following functions set the console background color:

- **bk_black**
- **bk_white**
- **bk_red**
- **bk_green**
- **bk_blue**
- **bk_cyan**

- **bk_magenta**
- **bk_yellow**

The matching **bk_light** and **bk_dark** functions control the brightness of the background, and they work just like *light* and *dark*.

If you want to go back to the default text color, just call the **default** function. There's also a **bk_default** function for the background color, and a *reset* method that reverts to the default for both at the same time.

Example #1: printing text with colors

Download

```
from winappdbg import Color

# Can we use colors?
if Color.can_use_colors():

    # Let's be polite: put everything in a try/except block
    # so we can reset the console colors before quitting.
    try:

        # Set black background.
        Color.bk_black()

        # For each color...
        for color in ( "red", "green", "blue", "cyan", "magenta", "yellow", "white" ):

            # Set the color.
            function = getattr( Color, color )
            function()

            # For each intensity...
            for intensity in ( "light", "dark" ):

                # Set the intensity.
                function = getattr( Color, intensity )
                function()

                # Print a message.
                print "This is %s %s text on black background." % ( intensity, color )

        # Set black text.
        Color.black()

        # For each color...
        for color in ( "red", "green", "blue", "cyan", "magenta", "yellow", "white" ):

            # Set the background color.
            function = getattr( Color, "bk_" + color )
            function()

            # For each intensity...
            for intensity in ( "light", "dark" ):

                # Set the background intensity.
                function = getattr( Color, "bk_" + intensity )
                function()
                # Print a message.
                print "This is black text on %s %s background." % ( intensity, color )

    # Reset the console colors and quit.
    finally:
        Color.reset()

# No colors available!
else:
    print "Can't use colors! Are you redirecting the output to a file?"
```

2.4.2 Text output in tables

The **Table** class lets you build text tables. Each row is added using the **addRow** method, and the number of columns is automatically inferred. Text justification for each column is defined using the **justify** method.

The **show** method prints the output. If you prefer to get the text table in a string, you can call the **getOutput** method instead. Also, the **getWidth** method tells you the width in characters of the whole table, so you know if it fits in the screen before printing it.

Example #2: printing a text table

Download

```
from winappdbg import Table

# Instance a Table object.
table = Table()

# Add a few rows.
table.addRow( "Right justified column text", "Left justified column text" )
table.addRow( "---------------------------", "--------------------------" )
table.addRow( "example", "text" )
table.addRow( "jabberwocky", "snark" )
table.addRow( "Trillian", "Zaphod", "Arthur Dent" )     # one extra!
table.addRow( "Dalek", "Cyberman" )

# By default all columns are left justified. Let's change that.
table.justify( 0, 1 )  # column 0 is now right justified

# Let's find out how wide the table is.
print "Table width: %d" % table.getWidth()

# Let's find out how many bytes would it be if written to a file.
print "Text size in characters: %d" % len( table.getOutput() )

# Show the table contents on screen.
print
table.show(),
```

2.4.3 Logging

The **Logger** class implements a simple text logger that can send its output to standard output and/or to a file. There are many libraries in Python that can do this, but this one has the advantage of being integrated with *WinAppDbg* objects.

If you want to integrate other logging facilities to your scripts you can also use the functions from the static class **DebugLog**, which contains all the WinAppDbg-related implementation of *Logger*.

Example #3: logging debug events

Download

```
from os.path import basename, splitext

from winappdbg import Debug, EventHandler, Logger, DebugLog

def main( argv ):

    # The log file name will be based on the target executable file name.
    logfile = basename( argv[ 0 ] )
    logfile = splitext( logfile )[ 0 ] + ".log"

    # Instance a global Logger object.
    global logger
    logger = Logger( logfile )

    # Launch the debugger.
    try:
        simple_debugger( argv )

    # On error log the exception and quit.
    except:
        logger.log_exc()

def my_event_handler( event ):

    # Get the Logger object.
    global logger

    # Log the event.
    logger.log_event( event )

def simple_debugger( argv ):

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

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

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

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

2.4.4 Hexadecimal input

The static class **HexInput** contains a collection of functions to parse input data in various formats.

- **integer:** Convert a string to an integer. Supports decimal, hexadecimal (0x prefix), octal (0o prefix) and binary (0b prefix).

  If no prefix is given, this method still does its best to tell if it's hexadecimal or not. If all fails, the number is assumed to be decimal.
- **address:** Read an hexadecimal value from a string. Unlike *integer* no attempt is made to detect other formats. This function was conceived for parsing memory addresses, hence the name.
- **hexadecimal:** Convert a strip of hexadecimal numbers (like OllyDbg's memory view) into binary data.
- **pattern:** Similar to *hexadecimal*, but it also accepts question marks as wildcards for unknown values in fixed positions. The return value is a regular expression that can perform a search for the given byte pattern.
- **is_pattern:** Determine if the given argument is a valid hexadecimal pattern to be used with *pattern*.
- **integer_list_file:** Read a list of integers from a file, assuming a specific file format.

  Check the documentation for HexInput.integer_list_file for details.
- **string_list_file:** Read a list of strings from a file, assuming a specific file format.

  Check the documentation for HexInput.string_list_file for details.
- **mixed_list_file:** Read a list of integers and strings from a file, assuming a specific file format.

  Check the documentation for HexInput.mixed_list_file for details.

2.4.5 Hexadecimal output

Two static classes contain all the functions related to hexadecimal output: **HexOutput** and **HexDump**. The first matches the input functions from *HexInput*, while the second is meant for showing data to the user rather than being parsed by a script.

The following functions are common to both:

- **integer:** Numeric value output, in decimal format.

  The default size depends on the current architecture, but you can override it using the *bits* parameter.
- **address:** Memory address output, in hexadecimal format.

  The default size depends on the current architecture, but you can override it using the *bits* parameter.
- **hexadecimal:** Output binary data as a strip of hexadecimal numbers (like OllyDbg's memory view).

  Currently both implementations are identical.

The *HexOutput* class also has file output functions to match those in *HexInput*:

- **integer_list_file:** Write a list of integers into a file, assuming a specific file format.

  Check the documentation for HexOutput.integer_list_file for details.
- **string_list_file:** Write a list of strings into a file, assuming a specific file format.

  Check the documentation for HexOutput.string_list_file for details.
- **mixed_list_file:** Write a list of integers and strings into a file, assuming a specific file format.

  Check the documentation for HexOutput.mixed_list_file for details.

The *HexDump* class has additional methods for showing hex dumps and binary data to the user in a printable manner:

- **hexblock:** Dump a block of hexadecimal numbers from binary data. Also show a printable text version of the data. The output mimics that of the WinDBG debugger.

- **hexline:** Dump a line of hexadecimal numbers from binary data. This is useful for printing bytes in a console one line at a time.
- **hexa_word:** Convert binary data to a string of hexadecimal WORDs.
- **hexa_dword:** Convert binary data to a string of hexadecimal DWORDs.
- **hexa_qword:** Convert binary data to a string of hexadecimal QWORDs.
- **hexblock_byte:** Dump a block of hexadecimal BYTEs from binary data.
- **hexblock_word:** Dump a block of hexadecimal WORDs from binary data.
- **hexblock_dword:** Dump a block of hexadecimal DWORDs from binary data.
- **hexblock_qword:** Dump a block of hexadecimal QWORDs from binary data.
- **hexblock_cb:** Dump a block of binary data using a callback function to convert each line of text. This allows you to customize the output.

2.4.6 Dumping code, stack and registers

The **CrashDump** static class has functions tipically used from the event handlers to show debug data like the disassembler output, the register contents or the stack trace. Crash dump objects use this class for text output, and pretty many examples in the *Debugging* section of the tutorial use functions from here too.

All functions return a string with the text to print. Here are the most commonly used ones:

dump_code Dump a disassembly. Optionally mark where the program counter is.

dump_registers Dump the x86 processor register values. The output mimics that of the WinDBG debugger.

dump_stack_trace Dump a stack trace using only memory addresses.

dump_stack_trace_with_labels Dump a stack trace using labels instead of memory addresses when possible.

Example #4: dumping code, stack and registers

Download

```
from winappdbg import Thread, HexDump, CrashDump, System

def print_state( process_name ):

    # Request debug privileges.
    System.request_debug_privileges()

    # Find the first process that matches the requested name.
    system = System()
    process, filename = system.find_processes_by_filename( process_name )[ 0 ]

    # Suspend the process execution.
    process.suspend()
    try:

        # For each thread in the process...
        for thread in process.iter_threads():

            # Get the thread state.
            tid     = thread.get_tid()
            eip     = thread.get_pc()
            code    = thread.disassemble_around( eip )
            context = thread.get_context()

            # Display the thread state.
            print
            print "-" * 79
            print "Thread: %s" % HexDump.integer( tid )
            print
            print CrashDump.dump_registers( context )
            print CrashDump.dump_code( code, eip ),
            print "-" * 79

    # Resume the process execution.
    finally:
        process.resume()
```

2.4.7 Pathname and filename handling

The **PathOperations** static class provides functions to manipulate pathnames and filenames. It's somewhat similar to the standard *os.path* module - except that it works by using only the Win32 API instead of manually parsing the filenames, which provides better compatibility with Windows (UNC path support, for example).

- **path_is_relative:** Returns True if the path is relative.
- **path_is_absolute:** Returns True if the path is absolute.
- **make_relative:** Converts an absolute to a relative path.
- **make_absolute:** Converts a relative to an absolute path.
- **split_filename:** Split the file from the directory where it resides.
- **split_extension:** Split the file name from the file extension.
- **split_path:** Split each component of a path.
- **join_path:** Join back the components of a path.
- **native_to_win32_pathname:** Converts an NT Native path to a standard Win32 path.

Example #5: pathname and filename handling

Download

```
import sys

from winappdbg import PathOperations

# Get the command line argument.
path = sys.argv[ 1 ]
print "Path: %s" % path

# If it's a relative path...
if PathOperations.path_is_relative( path ):
    print "Path is relative."

    # Convert to absolute.
    absolute = PathOperations.make_absolute( path )
    print "Absolute path: %s" % absolute
```

```
# If it's an absolute path...
elif PathOperations.path_is_absolute( path ):
    print "Path is absolute."

    # Convert to relative.
    relative = PathOperations.make_relative( path )
    print "Relative path: %s" % relative

# If it's neither...
else:
    print "Path is invalid."
```

2.5 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*).

All handles returned by API calls are wrapped around the *Handle* class. This allows you to use the **with** statement to ensure proper cleanup, and causes handles to be closed automatically when they go out of scope, thus preventing handle leaks.

2.5.1 Example #1: finding a DLL in the search path

`Download`

```
import sys

from winappdbg import win32

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

```
except WindowsError, e:
    if e.winerror != win32.ERROR_FILE_NOT_FOUND:
        raise
    fullpath, basename = win32.SearchPath( None, sys.argv[1], '.exe' )

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

2.5.2 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()
```

2.5.3 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
```

2.5.4 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
```

2.5.5 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)
```

2.6 More examples

2.6.1 Set a debugging timeout

Sometimes you'll want to set a maximum time to debug your target, especially when fuzzing or analyzing malware. 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

# Using the Debug object in a "with" context ensures proper cleanup.
with Debug( bKillOnExit = True ) as dbg:

    # Run the Windows Calculator (calc.exe).
    dbg.execl('calc.exe')

    # For the extra paranoid: this makes sure calc.exe dies
    # even if our own process is killed from the Task Manager.
    System.set_kill_on_exit_mode(True)

    # The execution time limit is 5 seconds.
    maxTime = time() + 5
    # Loop while calc.exe is alive and the time limit wasn't reached.
    while dbg and time() < maxTime:
        try:

            # Get the next debug event.
            dbg.wait(1000)   # 1 second accuracy

            # Show the current time on screen.
            print time()

        # If wait() times out just try again.
        # On any other error stop debugging.
        except WindowsError, e:
            if e.winerror in (win32.ERROR_SEM_TIMEOUT,
                              win32.WAIT_TIMEOUT):
                continue
            raise

        # Dispatch the event and continue execution.
        try:
            dbg.dispatch()
        finally:
            dbg.cont()
```

2.6.2 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:]:
    print "Looking for: %s" % filename
    for process, pathname in system.find_processes_by_filename(filename):
        pid  = process.get_pid()
        bits = process.get_bits()
        print "Dumping memory for process ID %d (%d bits)" % (pid, bits)
```

```
# 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, bits),
            winappdbg.HexDump.address(BaseAddress + RegionSize, bits)
        )
        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."
```

2.6.3 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.

Note that in 64 bit processors most memory addresses are not alphanumeric, so this example is meaningful for 32 bits only.

Download

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

# Iterator of alphanumeric executable addresses.
def iterate_alnum_jump_addresses(process):

    # Determine the size of a pointer in the current architecture.
    if System.bits == 32:
        fmt = 'L'
    elif System.bits == 64:
        fmt = 'Q'
        print "Warning! 64 bit addresses are not likely to be alphanumeric!"
    else:
        raise NotImplementedError

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

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

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

        # Get the module that owns this memory region, if any.
        address = mbi.BaseAddress
        module  = process.get_module_at_address(address)

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

        # For each executable alphanumeric address...
        for address, packed, module in iterate_alnum_jump_addresses(process):

            # Format the address for printing.
            numeric = HexDump.address(address, process.get_bits())
            ascii   = repr(packed)

            # Format the module name for printing.
            if module:
                modname = module.get_name()
            else:
                modname = ""

            # Try to disassemble the code at this location.
            try:
                code = process.disassemble(address, 16)[0][2]
            except NotImplementedError:
                code = ""

            # Print it.
            print numeric, ascii, modname, code

    # 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()
```

2.6.4 Show processes DEP settings

Beginning with Windows XP SP3, it's possible to query a process and find out its Data Execution Prevention (DEP) settings. It may have DEP enabled or disabled, DEP-ATL thunking emulation enabled or disabled, and these settings may be changeable on runtime or permanent for the lifetime of the process.

This example shows all 32 bits processes the current user has permission to access and shows their DEP settings.

Download

```
from winappdbg import System, Table
from winappdbg.win32 import PROCESS_DEP_ENABLE, \
                            PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION, \
                            ERROR_ACCESS_DENIED

# Prepare the table.
header = ( " PID ", "DEP ", "DEP-ATL ", "Permanent ", "Filename " )
separator = [ " " * len(x) for x in header ]
table = Table()
table.addRow( *header )
table.addRow( *separator )
```

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

# Scan for running processes.
system = System()
try:
    system.scan_processes()
    #system.scan_process_filenames()
except WindowsError:
    system.scan_processes_fast()

# For each running process...
for process in system.iter_processes():
    try:

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

        # Skip "special" process IDs.
        if pid in (0, 4, 8):
            continue

        # Skip 64 bit processes.
        if process.get_bits() != 32:
            continue

        # Get the DEP policy flags.
        flags, permanent = process.get_dep_policy()

        # Determine if DEP is enabled.
        if flags & PROCESS_DEP_ENABLE:
            dep = " X"
        else:
            dep = ""

        # Determine if DEP-ATL thunk emulation is enabled.
        if flags & PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION:
            atl = ""
        else:
            atl = "   X"

        # Determine if the current DEP flag is permanent.
        if permanent:
            perm = "    X"
        else:
            perm = ""

    # Skip processes we don't have permission to access.
    except WindowsError, e:
        if e.winerror == ERROR_ACCESS_DENIED:
            continue
        raise

    # Get the filename.
    filename = process.get_filename()

    # Add the process to the table.
    table.addRow( pid, dep, atl, perm, filename )
```

```
# Print the table.
table.show()
```

2.6.5 Choose the disassembler you want to use

WinAppDbg supports several disassembler engines. When more than one compatible engine is installed a default one is picked. However, you can manually select which one you want to use.

This example shows you how to list the supported disassembler engines for the desired architecture and pick one.

Download

```
from sys import argv

from winappdbg import Disassembler, HexInput, CrashDump

# If there are no command line arguments...
if len( argv ) == 1:

    # Show the help message.
    print "Usage:"
    print "  %s <file> [offset] [size] [arch] [engine]" % argv[0]

    # Show the available disassembler engines.
    print
    print "Supported disassembler engines:"
    print "-------------------------------"
    for engine in Disassembler.engines:
        print
        print "Name: %s" % engine.name
        print "Description: %s" % engine.desc
        print "Supported architectures: %s" % ", ".join( engine.supported )

# If there are command line arguments...
else:

    # Get the arguments from the command line.
    filename = argv[1]
    try:
        offset = HexInput.address( argv[2] )
    except IndexError:
        offset = 0
    try:
        size = HexInput.integer( argv[3] )
    except IndexError:
        size = 0
    try:
        arch = argv[4]
    except IndexError:
        arch = None
    try:
        engine = argv[5]
    except IndexError:
        engine = None

    # Load the requested disassembler engine.
    disasm = Disassembler( arch, engine )
    # Load the binary code.
    with open( filename, 'rb' ) as fd:
        fd.seek( offset )
        if size:
            code = fd.read( size )
        else:
            code = fd.read()

    # Disassemble the code.
    disassembly = disasm.decode( offset, code )

    # Show the disassembly.
    print CrashDump.dump_code( disassembly, offset )
```

2.6.6 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 XP (Vista and 7 don't seem to be using them anymore). 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
```

2.7 Advanced topics

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

2.7.1 About the heuristic crash signatures

The signature is currently implemented as a tuple of some elements that 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.

The goal is to detect seemingly duplicated crashes in a large set of them. This tipically happens when fuzzing an application with little or no robustness, or when fuzzing a robust application with a very large fuzzing farm - the amount of crashes quickly becomes hard to manage. Depending on how many crashes are generated and how valuable each crash is, you may want to simply use this for classification, or to directly filter out potential duplicates before sending them to the database.

This simple implementation should suit most users needs, however if your project requires something more elaborated, just derive from the *Crash* class and reimplement the *signature()* method with your own custom algorithm.

These are the elements included in the signature:

- **Processor architecture**:

  For different platforms the locations in the code would change. So if the locations are the same, it may be just a coincidence rather than the same crash.

  But most importantly, even if it were the same crash we'd like to know we can trigger it in multiple platforms.

- **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 ASLR in Vista and above and DLL relocations in XP and below, a *label* is used instead of a memory address 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 ASLR in Vista and above and DLL relocations in XP and below, a *label* is used instead of a memory address 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 storing them in the database.

These are the elements **NOT** included in the signature:

- **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.

- **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. And the chances of collision are still slow thanks to all the other elements that factor in the signature.

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

  Both are most likely to contain garbage we're not interested in (for the signature, that is) 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. But the main goal of the signature is to eliminate noise when fuzzing an application that crashes too often, so false positives are not much of an issue. In a scenario when a crash is rare we wouldn't want filtering by signature at all.

- **Operating system version**:

  Doesn't tell if it's the same crash or not, unless we're fuzzing the OS itself - and in that case we'd be more interested in the names and versions of the binary files.

2.7.2 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:

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

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.
- 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. To get the debugging symbols you need to first install the Microsoft Debugging Tools, and then either install the debugging symbols for your version of Windows or set up your system to connect to the Microsoft Symbol Server.

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 stric
aProcessInstance.split_method( "CreateFileA" )                       # instance method, using the fuz
```

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 ignore
(None, None, 512)
>>> winappdbg.Process.split_label_strict(' ! 0x200')                       # meaningless ! is ignore
(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 extensi
('kernel32', None, None)
>>> aProcess.split_label_fuzzy( "CreateFileA" )                         # can tell a module from a f
(None, 'CreateFileA', None)
>>> aProcess.split_label_strict( "0x7c800000" )                         # strict mode can't tell bas
```

```
(None, None, 2088763392)
>>> aProcess.split_label_fuzzy( "0x7c800000" )                          # fuzzy mode can tell base ac
('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 loac
(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 inser
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
```

2.7.3 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:

- **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*.

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

- **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:

- **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:

- **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:

- **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:** *(Static, for page breakpoints only)* Align the given address to the start of the page it occupies.
- **align_address_to_page_end:** *(Static, 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.

- **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).

2.8 Command line tools

WinAppDbg 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 SQL 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*.

2.8.1 Crash logger

- `crash_logger.py`:

  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 (SQLite, MySQL, SQL Server, etc.) and/or log it through standard output.

  A *heuristic signature* 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 attaching to system services, 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.

  Settings are defined in a Unix-style configuration file. Here's a `template file` you can use, where all options are explained.

- `crash_report.py`:

  Shows the contents of the crashes database to standard output.

2.8.2 Process tools

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

- `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.

- `ptrace.py`:

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

- `pinject.py`:

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

- `pfind.py`:

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

- `plist.py`:

  Shows a list of all currently running processes.

- `pmap.py`:

  Shows a map of 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.

- `pstrings.py`:

  Dumps all ASCII strings from a live process.

2.8.3 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.