2ND EDITION

THE IDA PRO BOOK

THE *UNOFFICIAL GUIDE* TO THE WORLD'S MOST POPULAR DISASSEMBLER

CHRIS EAGLE

"I wholeheartedly recommend The IDA Pro Book *to all IDA Pro users."*
—Ilfak Guilfanov, creator of IDA Pro

no starch press

24

THE IDA DEBUGGER

IDA is most widely known as a disassembler, and it is clearly one of the finest tools available for performing static analysis of binaries. Given the sophistication of modern anti–static analysis techniques, it is not uncommon to combine static analysis tools and techniques with dynamic analysis tools and techniques in order to take advantage of the best of both worlds. Ideally, all of these tools would be integrated into a single package. Hex-Rays made that move when it introduced a debugger in version 4.5 of IDA and solidified IDA's role as a general-purpose reverse engineering tool. With each successive version of IDA, its debugging capabilities have been improved. In its latest version, IDA is capable of local and remote debugging on a number of different platforms and supports a number of different processors. IDA may also be configured to act as a frontend to Microsoft's WinDbg debugger, making it possible to perform Windows kernel debugging.

Over the course of the next few chapters, we will cover the basic features of IDA's debugger, using the debugger to assist with obfuscated code analysis and remote debugging of Windows, Linux, or OS X binaries. While we assume

that the reader possesses some familiarity with the use of debuggers, we will review many of the basic capabilities of debuggers in general as we progress through the features of IDA's debugger.

Launching the Debugger

Debuggers are typically used to perform one of two tasks: examining memory images (core dumps) associated with crashed processes and executing processes in a very controlled manner. A typical debugging session begins with the selection of a process to debug. There are two ways this is generally accomplished. First, most debuggers are capable of *attaching* to a running process (assuming the user has permission to do so). Depending on the debugger being used, the debugger itself may be able to present a list of available processes to choose from. Lacking such capability, the user must determine the ID of the process to which he wishes to attach and then command the debugger to attach to the specified process. The precise manner by which a debugger attaches to a process varies from one operating system to another and is beyond the scope of this book. When attaching to an existing process, it is not possible to monitor or control the process's initial startup sequence, because all of the startup and initialization code will already have completed before you have a chance to attach to the process.

The manner by which you attach to a process with the IDA debugger depends on whether a database is currently open or not. When no database is open, the Debugger ▸ Attach menu is available, as shown in Figure 24-1.

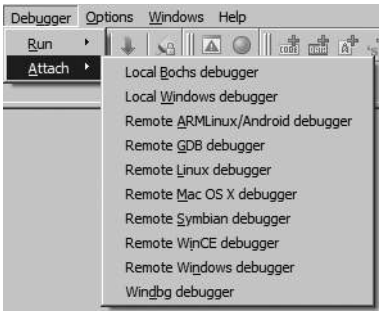

Figure 24-1: Attaching to an arbitrary process

Available options allow selection of different IDA debuggers (remote debugging is covered in Chapter 26). Options vary depending on the platform on which you are running IDA. Selecting a local debugger causes IDA to display a list of running processes to which you may attach. Figure 24-2 shows an example of such a list.

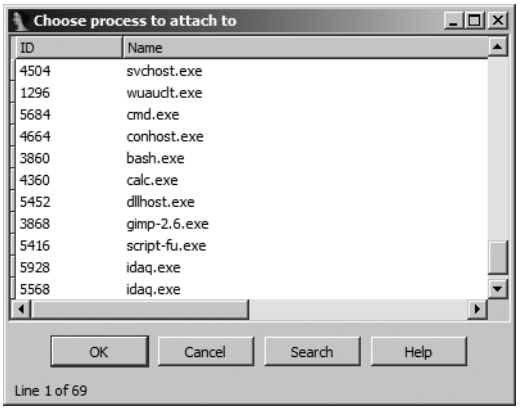

Figure 24-2: Debugger process-selection dialog

Once a process has been selected, the debugger creates a temporary database by taking a memory snapshot of the running process. In addition to the memory image of the running process, the temporary database contains sections for all shared libraries loaded by the process, resulting in a substantially larger and more cluttered database than you may be accustomed to. One drawback to attaching to a process in this manner is that IDA has less information available to disassemble the process because IDA's loader never processes the corresponding executable file image and an automated analysis of the binary is never performed. In fact, once the debugger has attached to the process, the only instructions that will be disassembled in the binary are the instruction referenced by the instruction pointer and those that flow from it. Attaching to a process immediately pauses the process, allowing you the opportunity to set breakpoints prior to resuming execution of the process.

An alternate way to attach to a running process is to open the associated executable in IDA before attempting to attach to the running process. With a database open, the Debugger menu takes on an entirely different form, as shown in Figure 24-3.

If you are not presented with this menu (or one very like it), then you probably have not yet specified a debugger to use for the currently open file type. In such cases, Debugger ▸ Select Debugger will present a list of suitable debuggers given the current file type. Figure 24-4 shows a typical debugger selection dialog.

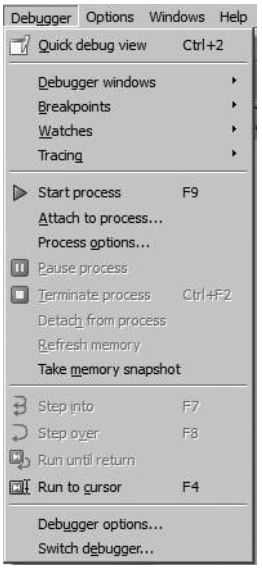

Figure 24-3: Debugger menu with a database open

Figure 24-4: Debugger selection dialog

You may make your selection the default debugger for the current file type by checking the box at the bottom of the dialog. The current default debugger, if any, is noted just above the checkbox. Once you have selected a debugger, you may change debuggers at any time via the Debug ▸ Switch Debugger menu.

When Debugger ▸ Attach to Process is selected, IDA's behavior will vary depending on the type of file opened in the active database. If the file is an executable file, IDA will display a list of all processes that have the same name as the file opened in the database. If IDA can find no process with a matching name, IDA will display a list of every running process and leave it to you to choose the correct process to attach to. In any case, you may attach to any of the displayed processes, but IDA has no way to guarantee that the process was started with same binary image that is loaded in the open IDA database.

IDA behaves differently if the currently open database is a shared library. On Windows systems, IDA will filter the displayed process list to just those processes that have the corresponding *.dll* file loaded. For example, if you are currently analyzing *wininet.dll* in IDA, then when you select Debugger ▸ Attach to Process, you will see only those processes that currently have *wininet.dll* loaded. On Linux and OS X systems, IDA does not have this filtering ability and displays every process to which you have the rights to attach.

As an alternative to attaching to an existing process, you may opt to launch a new process under debugger control. With no database open, a new process can be launched via Debugger ▸ Run. When a database is open, a new process can be launched via Debugger ▸ Start Process or Debugger ▸ Run to Cursor. Using the former causes the new process to execute until it hits a breakpoint (which you need to have set prior to choosing Debugger ▸ Start Process) or until you elect to pause the process using Debugger ▸ Pause Process. Using Debugger ▸ Run to Cursor automatically sets a breakpoint at the current cursor location prior to starting the new process. In this case, the new process will execute until the current cursor location is reached or until an

earlier breakpoint is hit. If execution never reaches the current cursor location (or any other breakpoint), the process will continue to run until it is forcibly paused or terminated (Debugger ▸ Terminate Process).

Launching a process under debugger control (as opposed to attaching to an existing process) is the only way to monitor every action the process takes. With breakpoints set prior to process initiation, it becomes possible to closely monitor a process's entire startup sequence. Controlling startup sequences is particularly important in the case of programs that have been obfuscated, because you will often want to pause the process immediately after the de-obfuscation routines complete and before the process begins its normal operations.

Another advantage to launching a process from an open IDA database is that IDA performs its initial autoanalysis on the process image before launching the process. This results in significantly better disassembly quality over that attained when attaching the debugger to an existing process.

IDA's debugger is capable of both local and remote debugging. For local debugging, you can only debug binaries that will run on your platform. There is no emulation layer that allows binaries from alternate platforms or CPU types to be executed within IDA's local debugger. For remote debugging, IDA ships with a number of debugging servers including implementations for Windows 32/64, Windows CE/ARM, Mac OS X 32/64, Linux 32/64/ARM, and Android. The debugging servers are intended to execute alongside the binary that you intend to debug. Once you have a remote debugging server running, IDA can communicate with the server to launch or attach to a target process on the remote machine. For Windows CE ARM devices, IDA communicates with the remote device using ActiveSync and installs the debugging server remotely. IDA is also capable of communicating with the `gdbserver`[1] component of the GNU Debugger[2] (gdb) or with programs that are linked with a suitable gdb remote stub.[3] Finally, for remote debugging on Symbian devices, you must install and configure Metrowerk's App TRK[4] in order for IDA to communicate with the device over a serial port. In any case, IDA is capable of acting as a debugger frontend only for processing running on x86, x64, MIPS, ARM, and PPC processors. Remote debugging is discussed in Chapter 26.

As with any other debugger, if you intend to use IDA's debugger to launch new processes, the original executable file is required to be present on the debugging host, and the original binary will be executed with the full privileges of the user running IDA. In other words, it is not sufficient to have only an IDA database loaded with the binary you wish to debug. This is extremely important to understand if you intend to use the IDA debugger for malware analysis. You can easily infect the debugging target machine if you fail to properly control the malware sample. IDA attempts to warn you of

1. See *http://www.sourceware.org/gdb/current/onlinedocs/gdb/Server.html#Server*.

2. See *http://www.gnu.org/software/gdb/*.

3. See *http://www.sourceware.org/gdb/current/onlinedocs/gdb/Remote-Stub.html#Remote-Stub*.

4. See *http://www.tools.ext.nokia.com/agents/index.htm*.

this possibility anytime you select Debugger ▸ Start Process (or Debugger ▸ Attach to process with an open database) by displaying a debugger warning message stating the following:

> You are going to launch the debugger. Debugging a program means that its code will be executed on your system.
>
> Be careful with malicious programs, viruses and trojans!
>
> REMARK: if you select 'No', the debugger will be automatically disabled.
>
> Are you sure you want to continue?

Selecting No in response to this warning causes the Debugger menu to be removed from the IDA menu bar. The Debugger menu will not be restored until you close the active database.

It is highly recommended that you perform any debugging of malicious software within a sandbox environment. In contrast, the x86 emulator plug-in discussed in Chapter 21 neither requires that the original binary be present nor executes any of the binary's instructions on the machine performing the emulation.

Basic Debugger Displays

Regardless of how you happen to launch the debugger, once your process of interest has been paused under debugger control, IDA enters its debugger mode (as opposed to normal disassembly mode), and you are presented with several default displays. The default debugger display is shown in Figure 24-5.

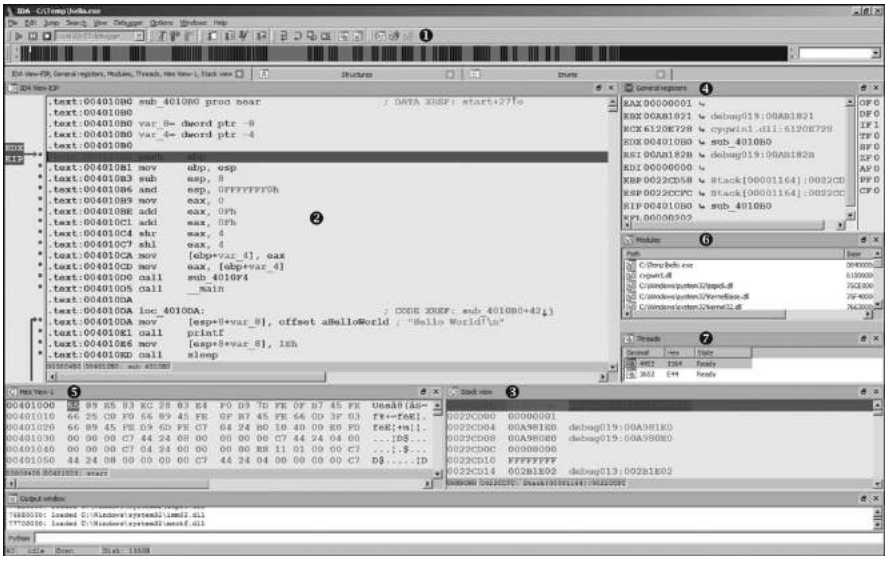

Figure 24-5: IDA debugger display

If you are accustomed to using other Windows debuggers such as OllyDbg[5] or Immunity Debugger,[6] one of your first thoughts might be that not much information is displayed on the screen. This is primarily a result of the fact that IDA defaults to a font size that is actually readable. If you find yourself missing the micro fonts used in other debuggers, you can easily change things via the Options ▸ Font menu. You may also wish to make use of saved IDA desktops (Windows ▸ Save Desktop) if you develop a fondness for a specific layout of your debugger windows.

As shown in the Figure 24-5, the debugger toolbar ❶ replaces the disassembly toolbar. A number of standard (from a debugging standpoint) tools are present, including process control tools and breakpoint manipulation tools.

The IDA View-EIP ❷ disassembly window is a default disassembly listing window when the debugger is active. It also happens to be synchronized with the current value of the instruction pointer register. If IDA detects that a register points to a memory location within the disassembly window, the name of that register is displayed in the left margin, opposite the address to which the register points. In Figure 24-5, the location to which EIP points is flagged in IDA View-EIP (note that EDX also points to the same location in this example). By default, IDA highlights breakpoints in red and the next instruction to be executed (the one to which the instruction pointer points) in blue. Debugger-related disassemblies are generated via the same disassembly process used in standard disassembly mode. Thus, IDA's debugger offers perhaps the best disassembly capability to be found in a debugger. Additionally, if you launched the debugger from an open IDA database, IDA is able to characterize all of the executable content based on analysis performed prior to launching the debugger. IDA's ability to disassemble any library code that has been loaded by the process will be somewhat more limited because IDA has not had a chance to analyze the associated *.dll* file prior to launching the debugger.

The Stack View ❸ window is another standard disassembly view primarily used to display the data contents of the process's runtime stack. All registers that point to stack locations are noted as such in the General Registers ❹ view (such as EBP in this case). Through the use of comments, IDA makes every attempt to provide context information for each data item on the stack. When the stack item is a memory address, IDA attempts to resolve the address to a function location (this helps highlight the location from which a function was called). When the stack item is a data pointer, a reference to the associated data item is displayed. The remaining default displays include the Hex view ❺, which offers a standard hex dump of memory, the Modules ❻ view, which displays a list of modules currently loaded in the process image, and the Threads ❼ view, which displays a list of threads in the current process. Double-clicking any listed thread causes the IDA View-EIP disassembly

5. See *http://www.ollydbg.de/*.

6. See *http://www.immunityinc.com/products-immdbg.shtml*.

window to jump to the current instruction within the selected thread and updates the General Registers view to reflect the current values for registers within the selected thread.

The General Registers window (also shown in Figure 24-6) displays the current contents of the CPU's general-purpose registers. Additional windows for displaying the contents of the CPU's segment, floating-point, or MMX registers may be opened from the Debugger menu.

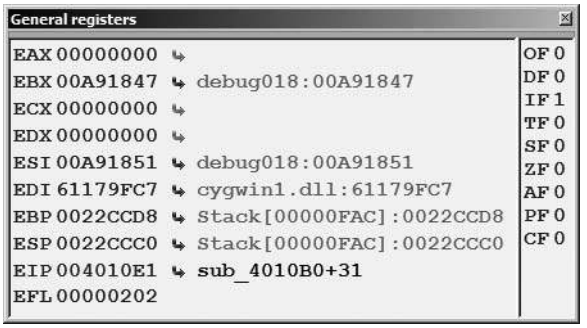

Figure 24-6: The General Registers display

Within the General Registers window, register contents are displayed to the right of the associated register name followed by a description of each register's content. The CPU flag bits are displayed down the rightmost column. Right-clicking a register value or flag bit provides access to a Modify menu item, which allows you to change the contents of any register or CPU flag. Menu options offer quick access to zero a value, toggle a value, increment a value, or decrement a value. Toggling values is particularly useful for changing CPU flag bits. Right-clicking any register value also provides access to the Open Register Window menu item. Selecting Open Register Window causes IDA to open a new disassembly window centered at the memory location held in the selected register. If you ever find that you have inadvertently closed either IDA View-EIP or IDA View-ESP, use the Open Register Window command on the appropriate register to reopen the lost window. If a register appears to point to a valid memory location, then the right-angle arrow control to the right of that register's value will be active and highlighted in black. Clicking an active arrow opens a new disassembly view centered on the corresponding memory location.

The Modules window displays a list of all executable files and shared libraries loaded into the process memory space. Double-clicking any module named in the list opens a list of symbols exported by that module. Figure 24-7 shows an example of the contents of *kernel32.dll*. The symbol list provides an easy way to track down functions within loaded libraries if you wish to set breakpoints on entry to those functions.

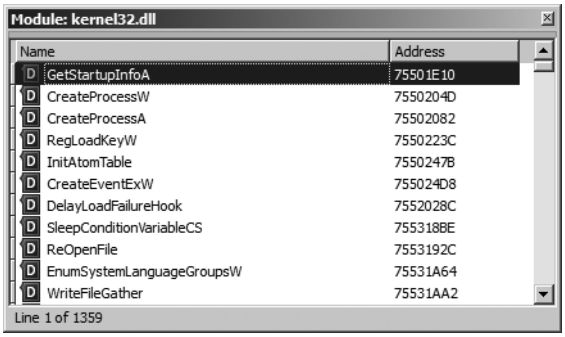

Figure 24-7: The Modules window with associated module contents

Additional debugger displays are accessible using various debugger menu selections. Displays pertaining to debugger operations will be discussed in the following section, "Process Control." Along with the debugger-specific displays, all traditional IDA subviews, such as Functions and Segments, remain available via the Views ▸ Open Subviews command.

Process Control

Perhaps the most important feature of any debugger is the ability to closely control—and modify, if desired—the behavior of the process being debugged. To that end, most debuggers offer commands that allow one or more instructions to be executed before returning control to the debugger. Such commands are often used in conjunction with breakpoints that allow the user to specify that execution should be interrupted when a designated instruction is reached or when a specific condition is met.

Basic execution of a process under debugger control is accomplished through the use of various Step, Continue, and Run commands. Because they are used so frequently, it is helpful to become familiar with the toolbar buttons and hotkey sequences associated with these commands. Figure 24-8 shows the toolbar buttons associated with execution of a process.

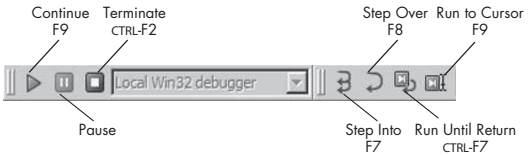

Figure 24-8: Debugger process control tools

The behavior of each of these commands is described in the following list:

Continue Resumes execution of a paused process. Execution continues until a breakpoint is hit, the user pauses or terminates execution, or the process terminates on its own.

Pause Pauses a running process.

Terminate Terminates a running process.

Step Into Executes the next instruction only. If the next instruction is a function call, breaks on the first instruction of the target function. Hence the name *Step Into*, since execution steps into any function being called.

Step Over Executes the next instruction only. If the next instruction is a function call, treats the call as a single instruction, breaking once the function returns. Hence the name *Step Over*, since stepping proceeds over functions rather than through them as with Step Into. Execution may be interrupted prior to completion of the function call if a breakpoint is encountered. Step Over is very useful as a time-saver when the behavior of a function is well known and uninteresting.

Run Until Return Resumes execution of the current function and does not stop until that function returns (or a breakpoint is encountered). This operation is useful when you have seen enough of a function and you wish to get out of it or when you inadvertently step into a function that you meant to step over.

Run to Cursor Resumes execution of the process and stops when execution reaches the current cursor location (or a breakpoint is hit). This feature is useful for running through large blocks of code without the need to set a permanent breakpoint at each location where you wish to pause. Beware that the program may not pause if the cursor location is bypassed or otherwise never reached.

In addition to toolbar and hotkey access, all of the execution control commands are accessible via the Debugger menu. Regardless of whether a process pauses after a single step or hitting a breakpoint, each time the process pauses, all debugger-related displays are updated to reflect the state of the process (CPU registers, flags, memory contents) at the time the process was paused.

Breakpoints

Breakpoints are a debugger feature that goes hand in hand with process execution and interruption (pausing). Breakpoints are set as a means of interrupting program execution at very specific locations within the program. In a sense a breakpoint is a more permanent extension of the Run to Cursor concept in that once a breakpoint is set at a given address, execution will always be interrupted when execution reaches that location, regardless of whether the cursor remains positioned on that location or not. However, while there is only one cursor to which execution can run, it is possible to set many breakpoints all over a program, the arrival at any one of which will

interrupt execution of the program. Breakpoints are set in IDA by navigating to the location at which you want execution to pause and using the F2 hotkey (or right-clicking and selecting Add Breakpoint). Addresses at which breakpoints have been set are highlighted with a red (by default) band across the entire disassembly line. A breakpoint may be removed by pressing F2 a second time to toggle the breakpoint off. A complete list of breakpoints currently set within a program may be viewed via Debugger ▸ Breakpoints ▸ Breakpoint List.

By default, IDA utilizes *software breakpoints*, which are implemented by replacing the opcode byte at the breakpoint address with a software breakpoint instruction. For x86 binaries, this is the `int 3` instruction, which uses opcode value `0xCC`. Under normal circumstances, when a software breakpoint instruction is executed, the operating system transfers control to any debugger that may be monitoring the interrupted process. As discussed in Chapter 21, obfuscated code may take advantage of the behavior of software breakpoints in an attempt to hinder normal operation of any attached debugger.

As an alternative to software breakpoints, some CPUs (such as the x86, actually 386, and later) offer support for *hardware-assisted breakpoints*. Hardware breakpoints are typically configured through the use of dedicated CPU registers. For x86 CPUs, these registers are called DR0–7 (debug registers 0 through 7). A maximum of four hardware breakpoints can be specified using x86 registers DR0–3. The remaining x86 debug registers are used to specify additional constraints on each breakpoint. When a hardware breakpoint is enabled, there is no need to substitute a special instruction into the program being debugged. Instead, the CPU itself decides whether execution should be interrupted or not based on values contained within the debug registers.

Once a breakpoint has been set, it is possible to modify various aspects of its behavior. Beyond simply interrupting the process, debuggers often support the concept of *conditional breakpoints*, which allow users to specify a condition that must be satisfied before the breakpoint is actually honored. When such a breakpoint is reached and the associated condition is not satisfied, the debugger automatically resumes execution of the program. The general idea is that the condition is expected to be satisfied at some point in the future, resulting in interruption of the program only when the condition you are interested in has been satisfied.

The IDA debugger supports both conditional and hardware breakpoints. In order to modify the default (unconditional, software-based) behavior of a breakpoint, you must edit a breakpoint after it has been set. In order to access the breakpoint-editing dialog, you must right-click an existing breakpoint and select Edit Breakpoint. Figure 24-9 shows the resulting Breakpoint Settings dialog.

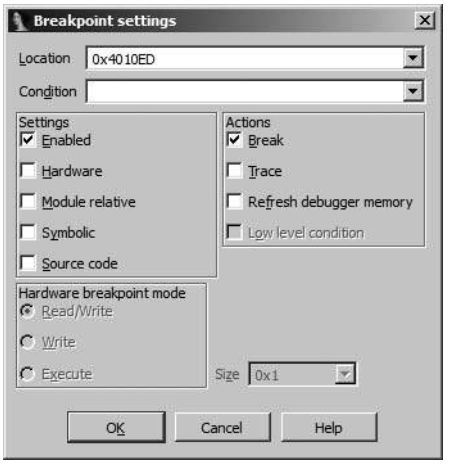

Figure 24-9: The Breakpoint Settings dialog

The Location box indicates the address of the breakpoint being edited, while the Enabled checkbox indicates whether the breakpoint is currently active or not. A breakpoint that is disabled is not honored regardless of any condition that may be associated with the breakpoint. The Hardware checkbox is used to request that the breakpoint be implemented in hardware rather than software.

WARNING *A word of caution concerning hardware breakpoints: Though the x86 only supports four hardware breakpoints at any given time, as of this writing (IDA version 6.1), IDA will happily allow you to designate more than four hardware breakpoints. However, only four of them will be honored. Any additional hardware breakpoints will be ignored.*

When specifying a hardware breakpoint, you must use the Hardware breakpoint mode radio buttons to specify whether the breakpoint behavior is to break on execute, break on write, or break on read/write. The latter two categories (break on write and break on read/write) allow you to create breakpoints that trigger when a specific memory location (usually a data location) is accessed, regardless of what instruction happens to be executing at the time the access takes place. This is very useful if you are more interested in when your program accesses a piece of data than where the data is accessed from.

In addition to specifying a mode for your hardware breakpoint, you must specify a size. For execute breakpoints the size must be 1 byte. For write or read/write breakpoints, the size may be set to 1, 2, or 4 bytes. When the size is set to 2 bytes, the breakpoint's address must be word aligned (a multiple of 2 bytes). Similarly, for 4-byte breakpoints, the breakpoint address must be double-word aligned (a multiple of 4 bytes). A hardware breakpoint's size is combined with its address to form a range of bytes over which the breakpoint may be triggered. An example may help to explain. Consider a 4-byte write

breakpoint set at address 0804C834h. This breakpoint will be triggered by a 1-byte write to 0804C837h, a 2-byte write to 0804C836h, and a 4-byte write to 0804C832h, among others. In each of these cases, at least 1 byte in the range 0804C834h0804C837h is written. More information on the behavior of x86 hardware breakpoints can be found in the *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.[7]

Conditional breakpoints are created by providing an expression in the Breakpoint Settings dialog's Condition field. Conditional breakpoints are a debugger feature, not an instruction set or CPU feature. When a breakpoint is triggered, it is the debugger's job to evaluate any associated conditional expression and determine whether the program should be paused (the condition is met) or whether execution should simply continue (the condition is not met). Therefore, conditions may be specified for both software and hardware breakpoints.

IDA breakpoint conditions are specified using IDC (not Python) expressions. Expressions that evaluate to non-zero are considered true, satisfying the breakpoint condition and triggering the breakpoint. Expressions that evaluate to zero are considered false, failing to satisfy the breakpoint condition and failing to trigger the associated breakpoint. In order to assist in the creation of breakpoint expressions, IDA makes special register variables available within IDC (again, not Python) to provide direct access to register contents in breakpoint expressions. These variables are named after the registers themselves and include EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFL, AX, BX, CX, DX, SI, DI, BP, SP, AL, AH, BL, BH, CL, CH, DL, and DH. These register variables are accessible only when the debugger is active.

Unfortunately, no variables exist that allow direct access to the processor flag bits. In order to access individual CPU flags, you need to call the GetRegValue function to obtain the value of the desired flag bit, such as CF. If you need a reminder regarding valid register and flag names, refer to the labels along the left and right edges of the General Registers window. A few example breakpoint expressions are shown here:

```
EAX == 100              // break if eax holds the value 100
ESI > EDI               // break if esi is greater than edi
Dword(EBP-20) == 10     // Read current stack frame (var_20) and compare to 10
GetRegValue("ZF")       // break if zero flag is set
EAX = 1                 // Set EAX to 1, this also evaluates to true (non-zero)
EIP = 0x0804186C        // Change EIP, perhaps to bypass code
```

Two things to note about breakpoint expressions are the fact that IDC functions may be called to access process information (as long as the function returns a value) and the fact that assignment can be used as a means of modifying register values at specific locations during process execution. Ilfak himself demonstrated this technique as an example of overriding a function return value.[8]

7. See *http://www.intel.com/products/processor/manuals/*.

8. See *http://www.hexblog.com/2005/11/simple_trick_to_hide_ida_debug.html* and *http://www.hexblog.com/2005/11/stealth_plugin_1.html*.

The last breakpoint options that can be configured in the Breakpoint Settings dialog are grouped into the Actions box on the right side of the dialog. The Break checkbox specifies whether program execution should actually be paused (assuming any associated condition is true) when the breakpoint is reached. It may seem unusual to create a breakpoint that doesn't break, but this is actually a useful feature if all you want to do is modify a specific memory or register value each time an instruction is reached without requiring the program to be paused at the same time. Selecting the Trace checkbox causes a trace event to be logged each time the breakpoint is hit.

Tracing

Tracing offers a means of logging specific events that occur while a process is executing. Trace events are logged to a fixed-size trace buffer and may optionally be logged to a trace file. Two styles of tracing are available: instruction tracing and function tracing. When *instruction tracing* is enabled (Debugger ▸ Tracing ▸ Instruction Tracing), IDA records the address, the instruction, and the values of any registers (other than EIP) that were changed by the instruction. Instruction tracing can slow down a debugged process considerably, because the debugger must single-step the process in order to monitor and record all register values. *Function tracing* (Debugger ▸ Tracing ▸ Function Tracing) is a subset of instruction tracing in which only function calls (and optionally returns) are logged. No register values are logged for function trace events.

Three types of individual trace events are also available: write traces, read/write traces, and execution traces. As their names imply, each allows logging of a trace event when a specific action occurs at a designated address. Each of these individual traces is implemented using nonbreaking breakpoints with the `trace` option set. Write and read/write traces are implemented using hardware breakpoints and thus fall under the same restrictions mentioned previously for hardware breakpoints, the most significant being that no more than four hardware-assisted breakpoints or traces may be active at any given time. By default, execution traces are implemented using software breakpoints, and thus there is no limit on the number of execution traces that can be set within a program.

Figure 24-10 shows the Tracing Options (Debugger ▸ Tracing ▸ Tracing Options) dialog used to configure the debugger's tracing operations.

Options specified here apply to function and instruction tracing only. These options have no effect on individual trace events. The Trace buffer size option specifies the maximum number of trace events that may be displayed at any given time. For a given buffer size *n*, only the *n* most recent trace events are displayed. Naming a log file causes all trace events to be appended to the named file. A file dialog is not offered when specifying a log file, so you must specify the complete path to the log file yourself. An IDC expression may be entered as a stop condition. The condition is evaluated prior to tracing through each instruction. If the condition evaluates to true, execution is immediately paused. The effect of this expression is to act as a conditional breakpoint that is not tied to any specific location.

Figure 24-10: The Tracing Options dialog

The Mark consecutive traced events with same IP option, when checked, causes consecutive trace events originating from the same instruction (*IP* here means *Instruction Pointer*) to be flagged with an equal sign. An example in which consecutive events can originate at the same instruction address occurs when the `REP`[9] prefix is used in `x86` programs. In order for an instruction trace to show each repetition at the same instruction address, the Log if same IP option must also be selected. Without this option selected, an instruction prefixed with `REP` is listed only once each time it is encountered. The following listing shows a partial instruction trace using the default trace settings:

```
  Thread   Address             Instruction    Result
  ------   -------             -----------    ------
❶ 00000150 .text:sub_401320+17 rep movsb      ECX=00000000 ESI=0022FE2C EDI=0022FCF4
  00000150 .text:sub_401320+19 pop esi        ESI=00000000 ESP=0022FCE4
```

Note that the `movsb` instruction ❶ is listed only once.

In the following listing, Log if same IP has been selected, resulting in each iteration of the `rep` loop being logged:

```
Thread   Address             Instruction   Result
------   -------             -----------   ------
000012AC .text:sub_401320+17 rep movsb     ECX=0000000B ESI=0022FE21 EDI=0022FCE9 EFL=00010206 RF=1
000012AC .text:sub_401320+17 rep movsb     ECX=0000000A ESI=0022FE22 EDI=0022FCEA
000012AC .text:sub_401320+17 rep movsb     ECX=00000009 ESI=0022FE23 EDI=0022FCEB
000012AC .text:sub_401320+17 rep movsb     ECX=00000008 ESI=0022FE24 EDI=0022FCEC
000012AC .text:sub_401320+17 rep movsb     ECX=00000007 ESI=0022FE25 EDI=0022FCED
000012AC .text:sub_401320+17 rep movsb     ECX=00000006 ESI=0022FE26 EDI=0022FCEE
000012AC .text:sub_401320+17 rep movsb     ECX=00000005 ESI=0022FE27 EDI=0022FCEF
000012AC .text:sub_401320+17 rep movsb     ECX=00000004 ESI=0022FE28 EDI=0022FCF0
000012AC .text:sub_401320+17 rep movsb     ECX=00000003 ESI=0022FE29 EDI=0022FCF1
```

9. The `REP` prefix is an instruction modifier that causes certain x86 string instructions such as `movs` and `scas` to be repeated based on a count contained in the ECX register.

```
000012AC .text:sub_401320+17 rep movsb     ECX=00000002 ESI=0022FE2A EDI=0022FCF2
000012AC .text:sub_401320+17 rep movsb     ECX=00000001 ESI=0022FE2B EDI=0022FCF3
000012AC .text:sub_401320+17 rep movsb     ECX=00000000 ESI=0022FE2C EDI=0022FCF4 EFL=00000206 RF=0
000012AC .text:sub_401320+19 pop esi       ESI=00000000 ESP=0022FCE4
```

Finally, in the following listing, the Mark consecutive traced events with same IP option has been enabled, resulting in special markings that highlight the fact that the instruction pointer has not changed from one instruction to the next:

```
Thread   Address              Instruction  Result
------   -------              -----------  ------
000017AC .text:sub_401320+17 rep movsb    ECX=0000000B ESI=0022FE21 EDI=0022FCE9 EFL=00010206 RF=1
=        =                    =           ECX=0000000A ESI=0022FE22 EDI=0022FCEA
=        =                    =           ECX=00000009 ESI=0022FE23 EDI=0022FCEB
=        =                    =           ECX=00000008 ESI=0022FE24 EDI=0022FCEC
=        =                    =           ECX=00000007 ESI=0022FE25 EDI=0022FCED
=        =                    =           ECX=00000006 ESI=0022FE26 EDI=0022FCEE
=        =                    =           ECX=00000005 ESI=0022FE27 EDI=0022FCEF
=        =                    =           ECX=00000004 ESI=0022FE28 EDI=0022FCF0
=        =                    =           ECX=00000003 ESI=0022FE29 EDI=0022FCF1
=        =                    =           ECX=00000002 ESI=0022FE2A EDI=0022FCF2
=        =                    =           ECX=00000001 ESI=0022FE2B EDI=0022FCF3
=        =                    =           ECX=00000000 ESI=0022FE2C EDI=0022FCF4 EFL=00000206 RF=0
000017AC .text:sub_401320+19 pop esi      ESI=00000000 ESP=0022FCE4
```

The last two options we will mention concerning tracing are Trace over debugger segments and Trace over library functions. When Trace over debugger segments is selected, instruction and function call tracing is temporarily disabled anytime execution proceeds to a program segment outside any of the file segments originally loaded into IDA. The most common example of this is a call to a shared library function. Selecting Trace over library functions temporarily disables function and instruction tracing anytime execution enters a function that IDA has identified as a library function (perhaps via FLIRT signature matching). Library functions linked into a binary should not be confused with library functions that a binary accesses via a shared library file such as a DLL. Both of these options are enabled by default, resulting in better performance while tracing (because the debugger does not need to step into library code) as well as a substantial reduction in the number of trace events generated, since instruction traces through library code can rapidly fill the trace buffer.

Stack Traces

A *stack trace* is a display of the current call stack, or sequence of function calls that have been made in order for execution to reach a particular location within a binary. Figure 24-11 shows a sample stack trace generated using the Debugger ▸ Stack Trace command.

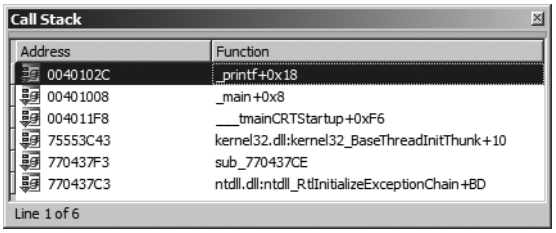

Figure 24-11: A sample stack trace

The top line in a stack trace lists the name of the function currently executing. The second line indicates the function that called the current function and the address from which that call was made. Successive lines indicate the point from which each function was called. A debugger is able to create a stack trace display by walking the stack and parsing each stack frame that it encounters, and it typically relies on the contents of the frame pointer register (EBP for x86) to locate the base of each stack frame. When a stack frame is located, the debugger can extract a pointer to the next stack frame (the saved frame pointer) as well as the saved return address, which is used to locate the call instruction used to invoke the current function. IDA's debugger cannot trace through stack frames that do not utilize EBP as a frame pointer. At the function (rather than individual instruction) level, stack traces are useful for answering the question, "How did I get here?" or, more correctly, "What sequence of function calls led to this particular location?"

Watches

While debugging a process, you may wish to constantly monitor the value contained in one or more variables. Rather than requiring you to navigate to the desired memory locations each time the process is paused, many debuggers allow you to specify lists of memory locations whose values should be displayed each time the process is stopped in the debugger. Such lists are called *watch lists*, because they allow you to watch as the contents of designated memory locations change during program execution. Watch lists are simply a navigational convenience; they do not cause execution to pause like a breakpoint.

Because they are focused on data, watch points (addresses designated to be watched) are most commonly set in the stack, heap, or data sections of a binary. Watches are set in the IDA debugger by right-clicking a memory item of interest and selecting Add Watch. Determining exactly which address to set a watch on may require some thought. Determining the address of a global variable is somewhat less challenging than determining the address of a local variable because global variables are allocated and assigned fixed addresses at compile time. Local variables, on the other hand, don't exist until runtime, and even then they exist only once the function in which they are declared has been called. With the debugger active, once you have stepped into a function, IDA is capable of reporting the addresses of local variables within that function. Figure 24-12 shows the result of mousing over a local variable named `arg_0` (actually a parameter passed into the function).

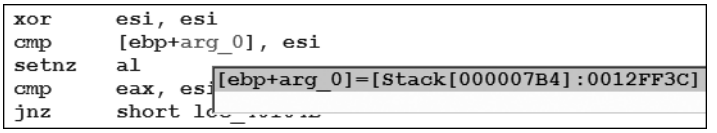

Figure 24-12: Debugger resolution of a local variable address

Double-clicking a local variable within an active function causes IDA to jump the main IDA View window to the address of that local variable. Having arrived at the variable's address, you may then add a watch on that address using the Add Watch context-sensitive menu option, though you will need to manually enter the address into the Watch Address dialog. If, instead, you take the time to name the memory location, IDA will automatically add a watch if you apply the same menu option to the name rather than the address.

You can access a list of all watches currently in effect via Debugger ▸ Watches ▸ Watch List. You can delete individual watches by highlighting the desired watch in the watch list and pressing DELETE.

Automating Debugger Tasks

In Chapters 15 through 19, we covered the basics of IDA scripting and the IDA SDK and demonstrated the usefulness of these capabilities during static analysis of binaries. Launching a process and working in the more dynamic environment of a debugger doesn't make scripting and plug-ins any less useful. Interesting uses for the automation provided by scripts and plug-ins include analyzing runtime data available while a process is being debugged, implementing complex breakpoint conditions, and implementing measures to subvert anti-debugging techniques.

Scripting Debugger Actions

All of the IDA scripting capabilities discussed in Chapter 15 continue to be accessible when you are using the IDA debugger. Scripts may be launched from the File menu, associated with hotkeys, and invoked from the IDA scripting command line. In addition, user-created IDC functions may be referenced from breakpoint conditions and tracing termination expressions.

Basic scripting functions offer the capability to set, modify, and enumerate breakpoints and the ability to read and write register and memory values. Memory access is provided by the `DbgByte`, `PatchDbgByte`, `DbgWord`, `PatchDbgWord`, `DbgDword`, and `PatchDbgDword` functions (analogous to the `Byte`, `Word`, `Dword`, and `Patch`*`XXX`* functions described in Chapter 15). Register and breakpoint manipulation is made possible by the following functions (please see the IDA help file for a complete list).

`long GetRegValue(string reg)`

Returns the value of the named register, such as EAX, as discussed previously. In IDC only, register values may also be easily accessed by using the desired register's name as a variable within an IDC expression.

bool SetRegValue(number val, string name)
Sets the value of the named register, such as EAX. If you are using IDC, register values may also be modified directly by using the desired register name on the left side of an assignment statement.

bool AddBpt(long addr)
Adds a software breakpoint at the indicated address.

bool AddBptEx(long addr, long size, long type)
Adds a breakpoint of the specified size and type at the indicated address. Type should be one of the `BPT_xxx` constants described in *idc.idc* or the IDA help file.

bool DelBpt(long addr)
Deletes a breakpoint at the specified address.

long GetBptQty()
Returns the number of breakpoints set within a program.

long GetBptEA(long bpt_num)
Returns the address at which the indicated breakpoint is set.

long/string GetBptAttr(long addr, number attr)
Returns an attribute associated with the breakpoint at the indicated address. The return value may be a number or a string depending on which attribute value has been requested. Attributes are specified using one of the `BPTATTR_xxx` values described in *idc.idc* or the IDA help file.

bool SetBptAttr(long addr, number attr, long value)
Sets the specified attribute of the specified breakpoint to the specified value. Do not use this function to set breakpoint condition expressions (use `SetBptCnd` instead).

bool SetBptCnd(long addr, string cond)
Sets the breakpoint condition to the provided conditional expression, which must be a valid IDC expression.

long CheckBpt(long addr)
Gets the breakpoint status at the specified address. Return values indicate whether there is no breakpoint, the breakpoint is disabled, the breakpoint is enabled, or the breakpoint is active. An active breakpoint is a breakpoint that is enabled while the debugger is also active.

The following script demonstrates how to install a custom IDC breakpoint-handling function at the current cursor location:

```
#include <idc.idc>
/*
 * The following should return 1 to break, and 0 to continue execution.
 */
static my_breakpoint_condition() {
   return AskYN(1, "my_breakpoint_condition activated, break now?") == 1;
}
```

```
/*
 * This function is required to register my_breakpoint_condition
 * as a breakpoint conditional expression
 */
static main() {
   auto addr;
   addr = ScreenEA();
   AddBpt(addr);
   SetBptCnd(addr, "my_breakpoint_condition()");
}
```

The complexity of `my_breakpoint_condition` is entirely up to you. In this example, each time the breakpoint is hit, a dialog will be displayed asking the user if she would like to continue execution of the process or pause at the current location. The value returned by `my_breakpoint_condition` is used by the debugger to determine whether the breakpoint should be honored or ignored.

Programmatic control of the debugger is possible from both the SDK and through the use of scripts. Within the SDK, IDA utilizes an event-driven model and provides callback notifications to plug-ins when specific debugger events occur. Unfortunately, IDA's scripting capabilities don't facilitate the use of an event-driven paradigm within scripts. As a result, Hex-Rays introduced a number of scripting functions that allow for synchronous control of the debugger from within scripts. The basic approach required to drive the debugger using a script is to initiate a debugger action and then wait for the corresponding debugger event code. Keep in mind that a call to a synchronous debugger function (which is all you can do in a script) blocks all other IDA operations until the call completes. The following list details several of the debugging extensions available for scripts:

`long GetDebuggerEvent(long wait_evt, long timeout)`
Waits for a debugger event (as specified by `wait_evt`) to take place within the specified number of seconds (–1 waits forever). Returns an event type code that indicates the type of event that was received. Specify `wait_evt` using a combination of one or more `WFNE_xxx` (WFNE stands for Wait For Next Event) flags. Possible return values are documented in the IDA help file.

`bool RunTo(long addr)`
Runs the process until the specified location is reached or until a breakpoint is hit.

`bool StepInto()`
Steps the process one instruction, stepping into any function calls.

`bool StepOver()`
Steps the process one instruction, stepping over any function calls. This call may terminate early if a breakpoint is hit.

`bool StepUntilRet()`
Runs until the current function call returns or until a breakpoint is hit.

bool EnableTracing(long trace_level, long enable)

Enables (or disables) the generation of trace events. The trace_level parameter should be set to one of the TRACE_*xxx* constants defined in *idc.idc*.

long GetEvent*XXX*()

A number of functions are available for retrieving information related to the current debug event. Some of these functions are valid only for specific event types. You should test the return value of GetDebuggerEvent in order to make sure that a particular GetEvent*XXX* function is valid.

GetDebuggerEvent must be called after each function that causes the process to execute in order to retrieve the debugger's event code. Failure to do so may prevent follow-up attempts to step or run the process. For example, the following code fragment will step the debugger only one time because GetDebuggerEvent does not get called to clear the last event type in between invocations of StepOver.

```
StepOver();
StepOver();    //this and all subsequent calls will fail
StepOver();
StepOver();
```

The proper way to perform an execution action is to follow up each call with a call to GetDebuggerEvent, as shown in the following example:

```
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
```

The calls to GetDebuggerEvent allow execution to continue even if you choose to ignore the return value from GetDebuggerEvent. The event type WFNE_SUSP indicates that we wish to wait for an event that results in suspension of the debugged process, such as an exception or a breakpoint. You may have noticed that there is no function that simply resumes execution of a suspended process.[10] However, it is possible to achieve the same effect by using the WFNE_CONT flag in a call to GetDebuggerEvent, as shown here:

```
GetDebuggerEvent(WFNE_SUSP | WFNE_CONT, -1);
```

This particular call waits for the next available suspend event after first resuming execution by continuing the process from the current instruction.

10. In reality, there is a macro named ResumeProcess that is defined as GetDebuggerEvent(WFNE_CONT|WFNE_NOWAIT, 0).

Additional functions are provided for automatically launching the debugger and attaching to running processes. See IDA's help file for more information on these functions.

An example of a simple debugger script for collecting statistics on the addresses of each executed instruction (provided the debugger is enabled) is shown here:

```
static main() {
   auto ca, code, addr, count, idx;
❶  ca = GetArrayId("stats");
   if (ca != -1) {
      DeleteArray(ca);
   }
   ca = CreateArray("stats");
❷  EnableTracing(TRACE_STEP, 1);
❸  for (code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1); code > 0;
          code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1)) {
❹     addr = GetEventEa();
❺     count = GetArrayElement(AR_LONG, ca, addr) + 1;
❻     SetArrayLong(ca, addr, count);
   }
   EnableTracing(TRACE_STEP, 0);
❼  for (idx = GetFirstIndex(AR_LONG, ca);
          idx != BADADDR;
          idx = GetNextIndex(AR_LONG, ca, idx)) {
      count = GetArrayElement(AR_LONG, ca, idx);
      Message("%x: %d\n", idx, count);
   }
❽  DeleteArray(ca);
}
```

The script begins ❶ by testing for the presence of a global array named `stats`. If one is found, the array is removed and re-created so that we can start with an empty array. Next ❷, single-step tracing is enabled before entering a loop ❸ to drive the single-stepping process. Each time a debug event is generated, the address of the associated event is retrieved ❹, the current count for the associated address is retrieved from the global array and incremented ❺, and the array is updated with the new count ❻. Note that the instruction pointer is used as the index into the sparse global array, which saves time looking up the address in some other form of data structure. Once the process completes, a second loop ❼ is used to retrieve and print all values from array locations that have valid values. In this case, the only array indexes that will have valid values represent addresses from which instructions were fetched. The script finishes off ❽ by deleting the global array that was used to gather the statistics. Example output from this script is shown here:

```
401028: 1
40102b: 1
40102e: 2
```

```
401031: 2
401034: 2
401036: 1
40103b: 1
```

A slight alteration of the preceding example can be used to gather statistics on what types of instructions are executed during the lifetime of a process. The following example shows the modifications required in the first loop to gather instruction-type data rather than address data:

```
   for (code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1); code > 0;
        code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1)) {
      addr = GetEventEa();
❶     mnem = GetMnem(addr);
❷     count = GetHashLong(ht, mnem) + 1;
❸     SetHashLong(ht, mnem, count);
   }
```

Rather than attempting to classify individual opcodes, we choose to group instructions by mnemonics ❶. Because mnemonics are strings, we make use of the hash-table feature of global arrays to retrieve the current count associated with a given mnemonic ❷ and save the updated count ❸ back into the correct hash table entry. Sample output from this modified script is shown here:

```
add:   18
and:   2
call:  46
cmp:   16
dec:   1
imul:  2
jge:   2
jmp:   5
jnz:   7
js:    1
jz:    5
lea:   4
mov:   56
pop:   25
push:  59
retn:  19
sar:   2
setnz: 3
test:  3
xor:   7
```

In Chapter 25 we will revisit the use of debugger-interaction capabilities as a means to assist in de-obfuscating binaries.

Automating Debugger Actions with IDA Plug-ins

In Chapter 16 you learned that IDA's SDK offers significant power for developing a variety of compiled extensions that can be integrated into IDA and that have complete access to the IDA API. The IDA API offers a superset of all the capabilities available in IDC, and the debugging extensions are no exception. Debugger extensions to the API are declared in *<SDKDIR>/dbg.hpp* and include C++ counterparts to all of the IDC functions discussed thus far, along with a complete asynchronous debugger interface capability.

For asynchronous interaction, plug-ins gain access to debugger notifications by hooking the `HT_DBG` notification type (see *loader.hpp*). Debugger notifications are declared in the `dbg_notification_t` enum found in *dbg.hpp*.

Within the debugger API, commands for interacting with the debugger are typically defined in pairs, with one function used for synchronous interaction (as with scripts) and the second function used for asynchronous interaction. Generically, the synchronous form of a function is named `COMMAND()`, and its asynchronous counterpart is named `request_COMMAND()`. The `request_XXX` versions are used to queue debugger actions for later processing. Once you finish queuing asynchronous requests, you must invoke the `run_requests` function to initiate processing of your request queue. As your requests are processed, debugger notifications will be delivered to any callback functions that you may have registered via `hook_to_notification_point`.

Using asynchronous notifications, we can develop an asynchronous version of the address-counting script from the previous section. The first task is to make sure that we hook and unhook debugger notifications. We will do this in the plug-in's `init` and `term` methods, as shown here:

```
//A netnode to gather stats into
❶ netnode stats("$ stats", 0, true);

int idaapi init(void) {
   hook_to_notification_point(HT_DBG, dbg_hook, NULL);
   return PLUGIN_KEEP;
}

void idaapi term(void) {
   unhook_from_notification_point(HT_DBG, dbg_hook, NULL);
}
```

Note that we have also elected to declare a global netnode ❶, which we will use to collect statistics. Next we consider what we want the plug-in to do when it is activated via its assigned hotkey. Our example plug-in `run` function is shown here:

```
void idaapi run(int arg) {
   stats.altdel();   //clear any existing stats
❶  request_enable_step_trace();
❷  request_step_until_ret();
❸  run_requests();
}
```

Since we are using asynchronous techniques in this example, we must first submit a request to enable step tracing ❶ and then submit a request to resume execution of the process being debugged. For the sake of simplicity, we will gather statistics on the current function only, so we will issue a request to run until the current function returns ❷. With our requests properly queued, we kick things off by invoking `run_requests` to process the current request queue ❸.

All that remains is to process the notifications that we expect to receive by creating our `HT_DBG` callback function. A simple callback that processes only two messages is shown here:

```
int idaapi dbg_hook(void *user_data, int notification_code, va_list va) {
   switch (notification_code) {
❶      case dbg_trace:  //notification arguments are detailed in dbg.hpp
          va_arg(va, thid_t);
❷         ea_t ea = va_arg(va, ea_t);
          //increment the count for this address
❸         stats.altset(ea, stats.altval(ea) + 1);
          return 0;
❹      case dbg_step_until_ret:
          //print results
❺         for (nodeidx_t i = stats.alt1st(); i != BADNODE; i = stats.altnxt(i)) {
              msg("%x: %d\n", i, stats.altval(i));
          }
          //delete the netnode and stop tracing
❻         stats.kill();
❼         request_disable_step_trace();
❽         run_requests();
          break;
   }
}
```

The `dbg_trace` notification ❶ will be received for each instruction that executes until we turn tracing off. When a trace notification is received, the address of the trace point is retrieved from the args list ❷ and then used to update the appropriate netnode array index ❸. The `dbg_step_until_ret` notification ❹ is sent once the process hits the `return` statement to leave the function in which we started. This notification is our signal that we should stop tracing and print any statistics we have gathered. A loop is used ❺ to iterate through all valid index values of the `stats` netnode before destroying the netnode ❻ and requesting that step tracing be disabled ❼. Since this example uses asynchronous commands, the request to disable tracing is added to the queue, which means we have to issue `run_requests` ❽ in order for the queue to be processed. An important warning about synchronous versus asynchronous interaction with the debugger is that you should never call the synchronous version of a function while actively processing an asynchronous notification message.

Synchronous interaction with the debugger using the SDK is done in a manner very similar to scripting the debugger. As with many of the SDK functions we have seen in previous chapters, the names of debugger-related

functions typically do not match the names of related scripting functions, so you may need to spend some time combing through *dbg.hpp* in order to find the functions you are looking for. The biggest disparity in names between scripting and the SDK is the SDK's version of `GetDebuggerEvent`, which is called `wait_for_next_event` in the SDK. The other major difference between script functions and the SDK is that variables corresponding to the CPU registers are not automatically declared for you within the SDK. In order to access the values of CPU registers from the SDK, you must use the `get_reg_val` and `set_reg_val` functions to read and write registers, respectively.

Summary

IDA may not have the largest share of the debugger market, but its debugger is powerful and integrates seamlessly with the disassembly side of IDA. While the debugger's user interface, like that of any debugger, requires some initial getting used to, it offers all of the fundamental features that users require in a basic debugger. Strong points include scripting and plug-in capabilities along with the familiar user interface of IDA's disassembly displays and the power of its analysis capabilities. Together the unified disassembler/debugger combination provides a solid tool for performing static analysis, dynamic analysis, or a combination of both.