

2

DEBUGGERS AND DEBUGGER DESIGN

Debuggers are the apple of the hacker's eye. Debuggers enable you to perform runtime tracing of a process, or *dynamic analysis*. The ability to perform dynamic analysis is absolutely essential when it comes to exploit development, fuzzer assistance, and malware inspection. It is crucial that you understand what debuggers are and what makes them tick. Debuggers provide a whole host of features and functionality that are useful when assessing software for defects. Most come with the ability to run, pause, or step a process; set breakpoints; manipulate registers and memory; and catch exceptions that occur inside the target process.

But before we move forward, let's discuss the difference between a white-box debugger and a black-box debugger. Most development platforms, or IDEs, contain a built-in debugger that enables developers to trace through their source code with a high degree of control. This is called *white-box debugging*. While these debuggers are useful during development, a reverse engineer, or *bug hunter*, rarely has the source code available and must employ black-box debuggers for tracing target applications. A *black-box* debugger

assumes that the software under inspection is completely opaque to the hacker, and the only information available is in a disassembled format. While this method of finding errors is more challenging and time consuming, a well-trained reverse engineer is able to understand the software system at a very high level. Sometimes the folks breaking the software can gain a deeper understanding than the developers who built it!

It is important to differentiate two subclasses of black-box debuggers: user mode and kernel mode. *User mode* (commonly referred to as *ring 3*) is a processor mode under which your user applications run. User-mode applications run with the least amount of privilege. When you launch *calc.exe* to do some math, you are spawning a user-mode process; if you were to trace this application, you would be doing user-mode debugging. *Kernel mode* (*ring 0*) is the highest level of privilege. This is where the core of the operating system runs, along with drivers and other low-level components. When you sniff packets with Wireshark, you are interacting with a driver that works in kernel mode. If you wanted to halt the driver and examine its state at any point, you would use a kernel-mode debugger.

There is a short list of user-mode debuggers commonly used by reverse engineers and hackers: *WinDbg*, from Microsoft, and *OllyDbg*, a free debugger from Oleh Yuschuk. When debugging on Linux, you'd use the standard *GNU Debugger (gdb)*. All three of these debuggers are quite powerful, and each offers a strength that others don't provide.

In recent years, however, there have been substantial advances in *intelligent debugging*, especially for the Windows platform. An intelligent debugger is scriptable, supports extended features such as call hooking, and generally has more advanced features specifically for bug hunting and reverse engineering. The two emerging leaders in this field are PyDbg by Pedram Amini and Immunity Debugger from Immunity, Inc.

PyDbg is a pure Python debugging implementation that allows the hacker full and automated control over a process, entirely in Python. *Immunity Debugger* is an amazing graphical debugger that looks and feels like OllyDbg but has numerous enhancements as well as the most powerful Python debugging library available today. Both of these debuggers get a thorough treatment in later chapters of this book. But for now, let's dive into some general debugging theory.

In this chapter, we will focus on user-mode applications on the x86 platform. We will begin by examining some very basic CPU architecture, coverage of the stack, and the anatomy of a user-mode debugger. The goal is for you to be able create your own debugger for any operating system, so it is critical that you understand the low-level theory first.

2.1 General-Purpose CPU Registers

A *register* is a small amount of storage on the CPU and is the fastest method for a CPU to access data. In the x86 instruction set, a CPU uses eight general-purpose registers: EAX, EDX, ECX, ESI, EDI, EBP, ESP, and EBX. More registers are available to the CPU, but we will cover them only in specific

circumstances where they are required. Each of the eight general-purpose registers is designed for a specific use, and each performs a function that enables the CPU to efficiently process instructions. It is important to understand what these registers are used for, as this knowledge will help to lay the groundwork for understanding how to design a debugger. Let's walk through each of the registers and its function. We will finish up by using a simple reverse engineering exercise to illustrate their uses.

The EAX register, also called the *accumulator register*, is used for performing calculations as well as storing return values from function calls. Many optimized instructions in the x86 instruction set are designed to move data into and out of the EAX register and perform calculations on that data. Most basic operations like add, subtract, and compare are optimized to use the EAX register. As well, more specialized operations like multiplication or division can occur *only* within the EAX register.

As previously noted, return values from function calls are stored in EAX. This is important to remember, so that you can easily determine if a function call has failed or succeeded based on the value stored in EAX. In addition, you can determine the actual *value* of what the function is returning.

The EDX register is the *data register*. This register is basically an extension of the EAX register, and it assists in storing extra data for more complex calculations like multiplication and division. It can also be used for general-purpose storage, but it is most commonly used in conjunction with calculations performed with the EAX register.

The ECX register, also called the *count register*, is used for looping operations. The repeated operations could be storing a string or counting numbers. An important point to understand is that ECX counts downward, not upward. Take the following snippet in Python, for example:

```
counter = 0

while counter < 10:
    print "Loop number: %d" % counter
    counter += 1
```

If you were to translate this code to assembly, ECX would equal 10 on the first loop, 9 on the second loop, and so on. This is a bit confusing, as it is the reverse of what is shown in Python, but just remember that it's always a downward count, and you'll be fine.

In x86 assembly, loops that process data rely on the ESI and EDI registers for efficient data manipulation. The ESI register is the *source index* for the data operation and holds the location of the input data stream. The EDI register points to the location where the result of a data operation is stored, or the *destination index*. An easy way to remember this is that ESI is used for reading and EDI is used for writing. Using the source and destination index registers for data operation greatly improves the performance of the running program.

The ESP and EBP registers are the *stack pointer* and the *base pointer*, respectively. These registers are used for managing function calls and stack operations. When a function is called, the arguments to the function are

pushed onto the stack and are followed by the return address. The ESP register points to the very top of the stack, and so it will point to the return address. The EBP register is used to point to the bottom of the call stack. In some circumstances a compiler may use optimizations to remove the EBP register as a stack frame pointer; in these cases the EBP register is freed up to be used like any other general-purpose register.

The EBX register is the only register that was not designed for anything specific. It can be used for extra storage.

One extra register that should be mentioned is the EIP register. This register points to the current instruction that is being executed. As the CPU moves through the binary executing code, EIP is updated to reflect the location where the execution is occurring.

A debugger must be able to easily read and modify the contents of these registers. Each operating system provides an interface for the debugger to interact with the CPU and retrieve or modify these values. We'll cover the individual interfaces in the operating system-specific chapters.

2.2 The Stack

The *stack* is a very important structure to understand when developing a debugger. The stack stores information about how a function is called, the parameters it takes, and how it should return after it is finished executing. The stack is a First In, Last Out (FILO) structure, where arguments are pushed onto the stack for a function call and popped off the stack when the function is finished. The ESP register is used to track the very top of the stack frame, and the EBP register is used to track the bottom of the stack frame. The stack grows from high memory addresses to low memory addresses. Let's use our previously covered function `my_socks()` as a simplified example of how the stack works.

Function Call in C

```
int my_socks(color_one, color_two, color_three);
```

Function Call in x86 Assembly

```
push color_three  
push color_two  
push color_one  
call my_socks
```

To see what the stack frame would look like, refer to Figure 2-1.

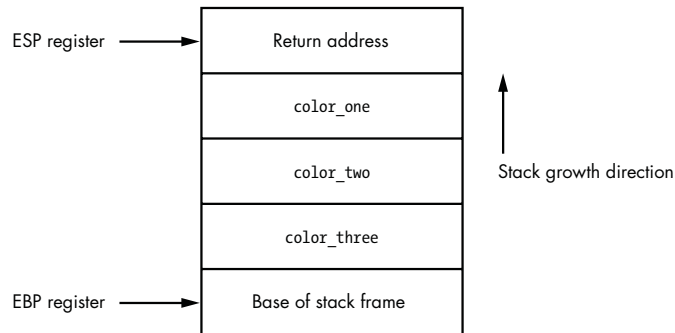


Figure 2-1: Stack frame for the `my_socks()` function call

As you can see, this is a straightforward data structure and is the basis for all function calls inside a binary. When the `my_socks()` function returns, it pops off all the values on the stack and jumps to the return address to continue executing in the parent function that called it. The other consideration is the notion of local variables. *Local variables* are slices of memory that are valid only for the function that is executing. To expand our `my_socks()` function a bit, let's assume that the first thing it does is set up a character array into which to copy the parameter `color_one`. The code would look like this:

```
int my_socks(color_one, color_two, color_three)
{
    char stinky_sock_color_one[10];
    ...
}
```

The variable `stinky_sock_color_one` would be allocated on the stack so that it can be used within the current stack frame. Once this allocation has occurred, the stack frame will look like the image in Figure 2-2.

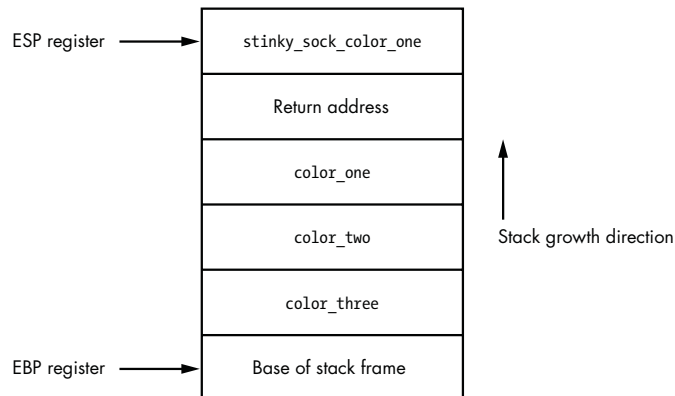


Figure 2-2: The stack frame after the local variable `stinky_sock_color_one` has been allocated

Now you can see how local variables are allocated on the stack and how the stack pointer gets incremented to continue to point to the top of the stack. The ability to capture the stack frame inside a debugger is very useful for tracing functions, capturing the stack state on a crash, and tracking down stack-based overflows.

2.3 Debug Events

Debuggers run as an endless loop that waits for a debugging event to occur. When a debugging event occurs, the loop breaks, and a corresponding event handler is called.

When an event handler is called, the debugger halts and awaits direction on how to continue. Some of the common events that a debugger must trap are these:

- Breakpoint hits
- Memory violations (also called access violations or segmentation faults)
- Exceptions generated by the debugged program

Each operating system has a different method for dispatching these events to a debugger, which will be covered in the operating system-specific chapters. In some operating systems, other events can be trapped as well, such as thread and process creation or the loading of a dynamic library at runtime. We will cover these special events where applicable.

An advantage of a scripted debugger is the ability to build custom event handlers to automate certain debugging tasks. For example, a buffer overflow is a common cause for memory violations and is of great interest to a hacker. During a regular debugging session, if there is a buffer overflow and a memory violation occurs, you must interact with the debugger and manually capture the information you are interested in. With a scripted debugger, you are able to build a handler that automatically gathers all of the relevant information without having to interact with it. The ability to create these customized handlers not only saves time, but it also enables a far wider degree of control over the debugged process.

2.4 Breakpoints

The ability to halt a process that is being debugged is achieved by setting *breakpoints*. By halting the process, you are able to inspect variables, stack arguments, and memory locations without the process changing any of their values before you can record them. Breakpoints are most definitely the most common feature that you will use when debugging a process, and we will cover them extensively. There are three primary breakpoint types: soft breakpoints, hardware breakpoints, and memory breakpoints. They each have very similar behavior, but they are implemented in very different ways.

2.4.1 Soft Breakpoints

Soft breakpoints are used specifically to halt the CPU when executing instructions and are by far the most common type of breakpoints that you will use when debugging applications. A soft breakpoint is a single-byte instruction that stops execution of the debugged process and passes control to the debugger's breakpoint exception handler. In order to understand how this works, you have to know the difference between an *instruction* and an *opcode* in x86 assembly.

An assembly instruction is a high-level representation of a command for the CPU to execute. An example is

```
MOV EAX, EBX
```

This instruction tells the CPU to move the value stored in the register EBX into the register EAX. Pretty simple, eh? However, the CPU does not know how to interpret that instruction; it needs it to be converted into something called an opcode. An *operation code*, or *opcode*, is a machine language command that the CPU executes. To illustrate, let's convert the previous instruction into its native opcode:

```
8BC3
```

As you can see, this obfuscates what's really going on behind the scenes, but it's the language that the CPU speaks. Think of assembly instructions as the DNS of CPUs. Instructions make it really easy to remember commands that are being executed (hostnames) instead of having to memorize all of the individual opcodes (IP addresses). You will rarely need to use opcodes in your day-to-day debugging, but they are important to understand for the purpose of soft breakpoints.

If the instruction we covered previously was at address 0x44332211, a common representation would look like this:

```
0x44332211:    8BC3    MOV EAX, EBX
```

This shows the address, the opcode, and the high-level assembly instruction. In order to set a soft breakpoint at this address and halt the CPU, we have to swap out a single byte from the 2-byte 8BC3 opcode. This single byte represents the interrupt 3 (INT 3) instruction, which tells the CPU to halt. The INT 3 instruction is converted into the single-byte opcode 0xCC. Here is our previous example, before and after setting a breakpoint.

Opcode Before Breakpoint Is Set

```
0x44332211:    8BC3    MOV EAX, EBX
```

Modified Opcode After Breakpoint Is Set

```
0x44332211:    CC C3      MOV EAX, EBX
```

You can see that we have swapped out the 8B byte and replaced it with a CC byte. When the CPU comes skipping along and hits that byte, it halts, firing an INT3 event. Debuggers have the built-in ability to handle this event, but since you will be designing your own debugger, it's good to understand how the debugger does it. When the debugger is told to set a breakpoint at a desired address, it reads the first opcode byte at the requested address and stores it. Then the debugger writes the CC byte to that address. When a breakpoint, or INT3, event is triggered by the CPU interpreting the CC opcode, the debugger catches it. The debugger then checks to see if the *instruction pointer* (EIP register) is pointing to an address on which it had set a breakpoint previously. If the address is found in the debugger's internal breakpoint list, it writes back the stored byte to that address so that the opcode can execute properly after the process is resumed. Figure 2-3 describes this process in detail.

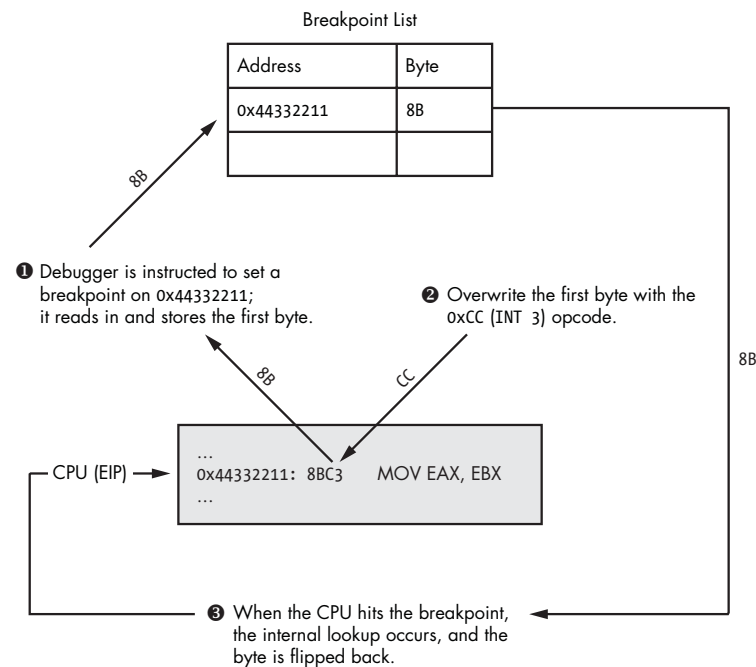


Figure 2-3: The process of setting a soft breakpoint

As you can see, the debugger must do quite a dance in order to handle soft breakpoints. There are two types of soft breakpoints that can be set: one-shot breakpoints and persistent breakpoints. A *one-shot soft breakpoint* means that once the breakpoint is hit, it gets removed from the internal breakpoint list; it's good for only one hit. A *persistent breakpoint* gets restored after the CPU has executed the original opcode, and so the entry in the breakpoint list is maintained.

Soft breakpoints have one caveat, however: when you change a byte of the executable in memory, you change the running software's *cyclic redundancy check (CRC)* checksum. A CRC is a type of function that is used to determine if data has been altered in any way, and it can be applied to files, memory, text, network packets, or anything you would like to monitor for data alteration. A CRC will take a range of values—in this case the running process's memory—and hash the contents. It then compares the hashed value against a known CRC checksum to determine whether there have been changes to the data. If the checksum is different from the checksum that is stored for validation, the CRC check fails. This is important to note, as quite often malware will test its running code in memory for any CRC changes and will kill itself if a failure is detected. This is a very effective technique to slow reverse engineering and prevent the use of soft breakpoints, thus limiting dynamic analysis of its behavior. In order to work around these specific scenarios, you can use hardware breakpoints.

2.4.2 Hardware Breakpoints

Hardware breakpoints are useful when a small number of breakpoints are desired and the debugged software itself cannot be modified. This style of breakpoint is set at the CPU level, in special registers called *debug registers*. A typical CPU has eight debug registers (registers DR0 through DR7), which are used to set and manage hardware breakpoints. Debug registers DR0 through DR3 are reserved for the addresses of the breakpoints. This means you can use only up to four hardware breakpoints at a time. Registers DR4 and DR5 are reserved, and DR6 is used as the status register, which determines the type of debugging event triggered by the breakpoint once it is hit. Debug register DR7 is essentially the on/off switch for the hardware breakpoints and also stores the different breakpoint conditions. By setting specific flags in the DR7 register, you can create breakpoints for the following conditions:

- Break when an instruction is executed at a particular address.
- Break when data is written to an address.
- Break on reads or writes to an address but not execution.

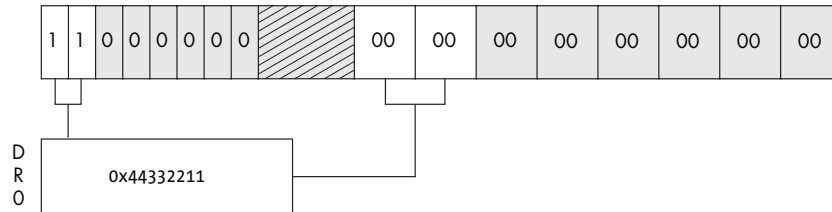
This is very useful, as you have the ability to set up to four very specific conditional breakpoints without modifying the running process. Figure 2-4 shows how the fields in DR7 are related to the hardware breakpoint behavior, length, and address.

Bits 0–7 are essentially the on/off switches for activating breakpoints. The L and G fields in bits 0–7 stand for local and global scope. I depict both bits as being set. However, setting either one will work, and in my experience I have not had any issues doing so during user-mode debugging. Bits 8–15 in DR7 are not used for the normal debugging purposes that we will be exercising. Refer to the Intel x86 manual for further explanation of those bits. Bits 16–31 determine the type and length of the breakpoint that is being set for the related debug register.

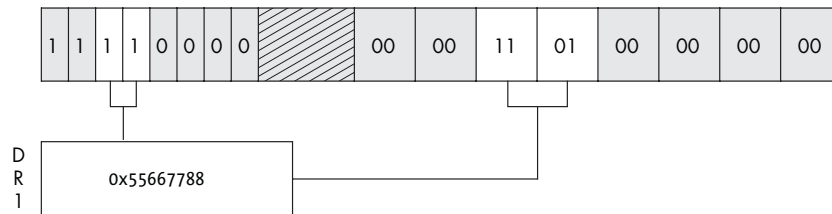
Layout of DR7 Register

L	G	L	G	L	G	L	G		Type	Len	Type	Len	Type	Len	Type	Len									
D	D	D	D	D	D	D	D		DR	DR	DR	DR	DR	DR	DR	DR									
R	R	R	R	R	R	R	R		0	0	1	1	2	2	3	3									
0	0	1	1	2	2	3	3																		
Bits	0	1	2	3	4	5	6	7	8 - 15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

DR7 with 1-byte Execution Breakpoint Set at 0x44332211



DR7 with Additional 2-byte Read/Write Breakpoint at 0x55667788



Breakpoint Flags	Breakpoint Length Flags
00 – Break on execution	00 – 1 byte
01 – Break on data writes	01 – 2 bytes (WORD)
11 – Break on reads or writes but not execution	11 – 4 bytes (DWORD)

Figure 2-4: You can see how the flags set in the DR7 register dictate what type of breakpoint is used.

Unlike soft breakpoints, which use the INT3 event, hardware breakpoints use interrupt 1 (INT1). The INT1 event is for hardware breakpoints and single-step events. *Single-step* simply means going one-by-one through instructions, allowing you to very closely inspect critical sections of code while monitoring data changes.

Hardware breakpoints are handled in much the same way as soft breakpoints, but the mechanism occurs at a lower level. Before the CPU attempts to execute an instruction, it first checks to see whether the address is currently enabled for a hardware breakpoint. It also checks to see whether any of the instruction operators access memory that is flagged for a hardware breakpoint. If the address is stored in debug registers DR0–DR3 and the read, write, or

execute conditions are met, an INT1 is fired and the CPU halts. If the address is not currently stored in the debug registers, the CPU executes the instruction and carries on to the next instruction, where it performs the check again, and so on.

Hardware breakpoints are extremely useful, but they do come with some limitations. Aside from the fact that you can set only four individual breakpoints at a time, you can also only set a breakpoint on a maximum of four bytes of data. This can be limiting if you want to track access to a large section of memory. In order to work around this limitation, you can have the debugger use memory breakpoints.

2.4.3 Memory Breakpoints

Memory breakpoints aren't really breakpoints at all. When a debugger is setting a memory breakpoint, it is changing the permissions on a region, or *page*, of memory. A memory page is the smallest portion of memory that an operating system handles. When a memory page is allocated, it has specific access permissions set, which dictate how that memory can be accessed. Some examples of memory page permissions are these:

Page execution This enables execution but throws an access violation if the process attempts to read or write to the page.

Page read This enables the process only to read from the page; any writes or execution attempts cause an access violation.

Page write This allows the process to write into the page.

Guard page Any access to a guard page results in a one-time exception, and then the page returns to its original status.

Most operating systems allow you to combine these permissions. For example, you may have a page in memory where you can read and write, while another page may allow you to read and execute. Each operating system also has intrinsic functions that allow you to query the current memory permissions in place for a particular page and modify them if so desired. Refer to Figure 2-5 to see how data access works with the various memory page permissions set.

The page permission we are interested in is the *guard page*. This type of page is quite useful for such things as separating the heap from the stack or ensuring that a portion of memory doesn't grow beyond an expected boundary. It is also quite useful for halting a process when it hits a particular section of memory. For example, if we are reverse engineering a networked server application, we could set a memory breakpoint on the region of memory where the payload of a packet is stored after it's received. This would enable us to determine when and how the application uses received packet contents, as any accesses to that memory page would halt the CPU, throwing a guard page debugging exception. We could then inspect the instruction that accessed the buffer in memory and determine what it is

doing with the contents. This breakpoint technique also works around the data alteration problems that soft breakpoints have, as we aren't changing any of the running code.

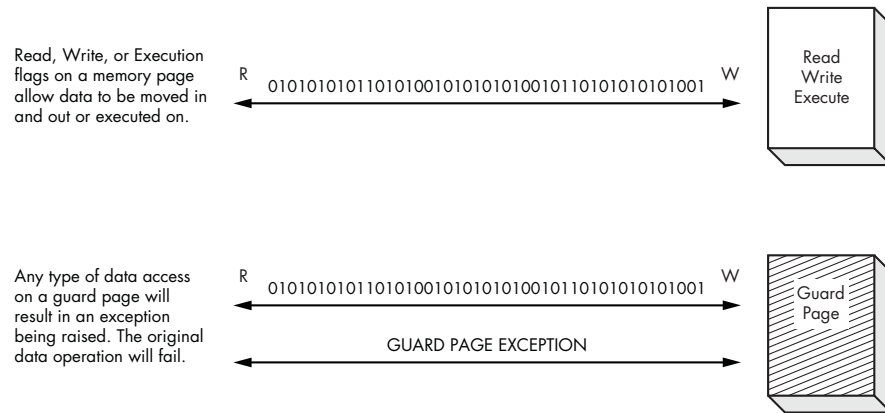


Figure 2-5: The behavior of the various memory page permissions

Now that we have covered some of the basic aspects of how a debugger works and how it interacts with the operating system, it's time to begin coding our first lightweight debugger in Python. We will begin by creating a simple debugger in Windows where the knowledge you have gained in both ctypes and debugging internals will be put to good use. Get those coding fingers warmed up.

4

PYDBG—A PURE PYTHON WINDOWS DEBUGGER

If you've made it this far, then you should have a good understanding of how to use Python to construct a user-mode debugger for Windows. We'll now move on to learning how to harness the power of PyDbg, an open source Python debugger for Windows. PyDbg was released by Pedram Amini at Recon 2006 in Montreal, Quebec, as a core component in the PaiMei¹ reverse engineering framework. PyDbg has been used in quite a few tools, including the popular proxy fuzzer Taof and a Windows driver fuzzer that I built called ioctlizer. We will start with extending breakpoint handlers and then move into more advanced topics such as handling application crashes and taking process snapshots. Some of the tools we'll build in this chapter can be used later on to support some of the fuzzers we are going to develop. Let's get on with it.

¹The PaiMei source tree, documentation, and development roadmap can be found at <http://code.google.com/p/paimei/>.

4.1 Extending Breakpoint Handlers

In the previous chapter we covered the basics of using event handlers to handle specific debugging events. With PyDbg it is quite easy to extend this basic functionality by implementing user-defined callback functions. With a user-defined callback, we can implement custom logic when the debugger receives a debugging event. The custom code can do a variety of things such as read certain memory offsets, set further breakpoints, or manipulate memory. Once the custom code has run, we return control to the debugger and allow it to resume the debuggee.

The PyDbg function to set soft breakpoints has the following prototype:

```
bp_set(address, description="", restore=True, handler=None)
```

The `address` parameter is the address where the soft breakpoint should be set; the `description` parameter is optional and can be used to uniquely name each breakpoint. The `restore` parameter determines whether the breakpoint should automatically be reset after it's handled, and the `handler` parameter specifies which function to call when this breakpoint is encountered. Breakpoint callback functions take only one parameter, which is an instance of the `pydbg()` class. All context, thread, and process information will already be populated in this class when it is passed to the callback function.

Using our `printf_loop.py` script, let's implement a user-defined callback function. For this exercise, we will read the value of the counter that is used in the `printf` loop and replace it with a random number between 1 and 100. One neat thing to remember is that we are actually observing, recording, and manipulating live events inside the target process. This is truly powerful! Open a new Python script, name it `printf_random.py`, and enter the following code.

printf_random.py

```
from pydbg import *
from pydbg.defines import *

import struct
import random

# This is our user defined callback function
def printf_randomizer(dbg):

    # Read in the value of the counter at ESP + 0x8 as a DWORD
    parameter_addr = dbg.context.Esp + 0x8
    counter = dbg.read_process_memory(parameter_addr,4)

    # When we use read_process_memory, it returns a packed binary
    # string. We must first unpack it before we can use it further.
```

```

counter = struct.unpack("L",counter)[0]
print "Counter: %d" % int(counter)

# Generate a random number and pack it into binary format
# so that it is written correctly back into the process
random_counter = random.randint(1,100)
random_counter = struct.pack("L",random_counter)[0]

# Now swap in our random number and resume the process
dbg.write_process_memory(parameter_addr,random_counter)

return DBG_CONTINUE

# Instantiate the pydbg class
dbg = pydbg()

# Now enter the PID of the printf_loop.py process
pid = raw_input("Enter the printf_loop.py PID: ")

# Attach the debugger to that process
dbg.attach(int(pid))

# Set the breakpoint with the printf_randomizer function
# defined as a callback
printf_address = dbg.func_resolve("msvcrt","printf")
dbg.bp_set(printf_address,description="printf_address",handler=printf_randomizer)

# Resume the process
dbg.run()

```

Now run both the *printf_loop.py* and the *printf_random.py* scripts. The output should look similar to what is shown in Table 4-1.

Table 4-1: Output from the Debugger and the Manipulated Process

Output from Debugger	Output from Debugged Process
Enter the printf_loop.py PID: 3466	Loop iteration 0!
...	Loop iteration 1!
...	Loop iteration 2!
...	Loop iteration 3!
Counter: 4	Loop iteration 32!
Counter: 5	Loop iteration 39!
Counter: 6	Loop iteration 86!
Counter: 7	Loop iteration 22!
Counter: 8	Loop iteration 70!
Counter: 9	Loop iteration 95!
Counter: 10	Loop iteration 60!

You can see that the debugger set a breakpoint on the fourth iteration of the infinite `printf` loop, because the counter as recorded by the debugger is set to 4. You will also notice that the `printf_loop.py` script ran fine until it reached iteration 4; instead of outputting the number 4, it output the number 32! It is clear to see how our debugger records the real value of the counter and sets the counter to a random number before it is output by the debugged process. This is a simple yet powerful example of how you can easily extend a scriptable debugger to perform additional actions when debugging events occur. Now let's take a look at handling application crashes with PyDbg.

4.2 Access Violation Handlers

An access violation occurs inside a process when it attempts to access memory it doesn't have permission to access or in a particular way that it is not allowed. The faults that lead to access violations range from buffer overflows to improperly handled null pointers. From a security perspective, every access violation should be reviewed carefully, as the violation might be exploited.

When an access violation occurs within a debugged process, the debugger is responsible for handling it. It is crucial that the debugger trap all information that is relevant, such as the stack frame, the registers, and the instruction that caused the violation. You can now use this information as a starting point for writing an exploit or creating a binary patch.

PyDbg has an excellent method for installing an access violation handler, as well as utility functions to output all of the pertinent crash information. Let's first create a test harness that will use the dangerous C function `strcpy()` to create a buffer overflow. Following the test harness, we will write a brief PyDbg script to attach to and handle the access violation. Let's start with the test script. Open a new file called `buffer_overflow.py`, and enter the following code.

buffer_overflow.py

```
from ctypes import *

msvcrt = cdll.msvcrt

# Give the debugger time to attach, then hit a button
raw_input("Once the debugger is attached, press any key.")

# Create the 5-byte destination buffer
buffer = c_char_p("AAAAA")

# The overflow string
overflow = "A" * 100

# Run the overflow
msvcrt.strcpy(buffer, overflow)
```

Now that we have the test case built, open a new file called *access_violation_handler.py*, and enter the following code.

access_violation_handler.py

```
from pydbg import *
from pydbg.defines import *

# Utility libraries included with PyDbg
import utils

# This is our access violation handler
def check_accessv(dbg):

    # We skip first-chance exceptions
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    dbg.terminate_process()

    return DBG_EXCEPTION_NOT_HANDLED

pid = raw_input("Enter the Process ID: ")

dbg = pydbg()
dbg.attach(int(pid))
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, check_accessv)
dbg.run()
```

Now run the *buffer_overflow.py* file and take note of its PID; it will pause until you are ready to let it run. Execute the *access_violation_handler.py* file, and enter the PID of the test harness. Once you have the debugger attached, hit any key in the console where the harness is running, and you will see output similar to Listing 4-1.

❶ python25.dll:1e071cd8 mov ecx,[eax+0x54] from thread 3376 caused access violation when attempting to read from 0x41414195

❷ CONTEXT DUMP

```
EIP: 1e071cd8 mov ecx,[eax+0x54]
EAX: 41414141 (1094795585) -> N/A
EBX: 00b055d0 ( 11556304) -> @U`" B`0x,`0 )Xb@|V`"L{0+H}$6 (heap)
ECX: 0021fe90 ( 2227856) -> !$4|7|4|@%,\!$H8|!OGGBG)00S\o (stack)
EDX: 00a1dc60 ( 10607712) -> VO`w`W (heap)
EDI: 1e071cd0 ( 503782608) -> N/A
ESI: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
EBP: 1e1cf448 ( 505214024) -> enable() -> NoneEnable automa (stack)
ESP: 0021fe74 ( 2227828) -> 2? BUH` 7|4|@%,\!$H8|!OGGBG) (stack)
```

```

+00: 00000000 (      0) -> N/A
+04: 1e063f32 ( 503725874) -> N/A
+08: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
+0c: 00000000 (      0) -> N/A
+10: 00000000 (      0) -> N/A
+14: 00b055c0 ( 11556288) -> @F@U`" B`0x,`0 )Xb@|V`"L{0+H}$ (heap)

```

③ disasm around:

```

0x1e071cc9 int3
0x1e071cca int3
0x1e071ccb int3
0x1e071ccc int3
0x1e071ccd int3
0x1e071cce int3
0x1e071ccf int3
0x1e071cd0 push esi
0x1e071cd1 mov esi,[esp+0x8]
0x1e071cd5 mov eax,[esi+0x4]
0x1e071cd8 mov ecx,[eax+0x54]
0x1e071cdb test ch,0x40
0x1e071cde jz 0x1e071cff
0x1e071ce0 mov eax,[eax+0xa4]
0x1e071ce6 test eax,eax
0x1e071ce8 jz 0x1e071cf4
0x1e071cea push esi
0x1e071ceb call eax
0x1e071ced add esp,0x4
0x1e071cf0 test eax,eax
0x1e071cf2 jz 0x1e071cff

```

④ SEH unwind:

```

0021ffe0 -> python.exe:1d00136c jmp [0x1d002040]
fffffff -> kernel32.dll:7c839aa8 push ebp

```

Listing 4-1: Crash output using PyDbg crash binning utility

The output reveals many pieces of useful information. The first portion ❶ tells you which instruction caused the access violation as well as which module that instruction lives in. This information is useful for writing an exploit or if you are using a static analysis tool to determine where the fault is. The second portion ❷ is the context dump of all the registers; of particular interest is that we have overwritten EAX with 0x41414141 (0x41 is the hexadecimal value of the capital letter A). As well, we can see that the ESI register points to a string of A characters, the same as for a stack pointer at ESP+08. The third section ❸ is a disassembly of the instructions before and after the faulting instruction, and the final section ❹ is the list of *structured exception handling (SEH)* handlers that were registered at the time of the crash.

You can see how simple it is to set up a crash handler using PyDbg. It is an incredibly useful feature that enables you to automate the crash handling and postmortem of a process that you are analyzing. Next we are going to use PyDbg's internal process snapshotting capability to build a process rewinder.

4.3 Process Snapshots

PyDbg comes stocked with a very cool feature called *process snapshotting*. Using process snapshotting you are able to freeze a process, obtain all of its memory, and resume the process. At any later point you can revert the process to the point where the snapshot was taken. This can be quite handy when reverse engineering a binary or analyzing a crash.

4.3.1 Obtaining Process Snapshots

Our first step is to get an accurate picture of what the target process was up to at a precise moment. In order for the picture to be accurate, we need to first obtain all threads and their respective CPU contexts. As well, we need to obtain all of the process's memory pages and their contents. Once we have this information, it's just a matter of storing it for when we want to restore a snapshot.

Before we can take the process snapshots, we have to suspend all threads of execution so that they don't change data or state while the snapshot is being taken. To suspend all threads in PyDbg, we use `suspend_all_threads()`, and to resume all the threads, we use the aptly named `resume_all_threads()`. Once we have suspended the threads, we simply make a call to `process_snapshot()`. This automatically fetches all of the contextual information about each thread and all memory at that precise moment. Once the snapshot is finished, we resume all of the threads. When we want to restore the process to the snapshot point, we suspend all of the threads, call `process_restore()`, and resume all of the threads. Once we resume the process, we should be back at our original snapshot point. Pretty neat, eh?

To try this out, let's use a simple example where we allow a user to hit a key to take a snapshot and hit a key again to restore the snapshot. Open a new Python file, call it *snapshot.py*, and enter the following code.

snapshot.py

```
from pydbg import *
from pydbg.defines import *

import threading
import time
import sys

class snapshotter(object):

    def __init__(self, exe_path):

        self.exe_path = exe_path
        self.pid = None
        self.dbg = None
        self.running = True
```

```

❶ # Start the debugger thread, and loop until it sets the PID
# of our target process
pydbg_thread = threading.Thread(target=self.start_debugger)
pydbg_thread.setDaemon(0)
pydbg_thread.start()

while self.pid == None:
    time.sleep(1)

❷ # We now have a PID and the target is running; let's get a
# second thread running to do the snapshots
monitor_thread = threading.Thread(target=self.monitor_debugger)
monitor_thread.setDaemon(0)
monitor_thread.start()

❸ def monitor_debugger(self):

    while self.running == True:

        input = raw_input("Enter: 'snap', 'restore' or 'quit'")
        input = input.lower().strip()

        if input == "quit":
            print "[*] Exiting the snapshotter."
            self.running = False
            self.dbg.terminate_process()

        elif input == "snap":

            print "[*] Suspending all threads."
            self.dbg.suspend_all_threads()

            print "[*] Obtaining snapshot."
            self.dbg.process_snapshot()

            print "[*] Resuming operation."
            self.dbg.resume_all_threads()

        elif input == "restore":

            print "[*] Suspending all threads."
            self.dbg.suspend_all_threads()

            print "[*] Restoring snapshot."
            self.dbg.process_restore()

            print "[*] Resuming operation."
            self.dbg.resume_all_threads()

❹ def start_debugger(self):

    self.dbg = pydbg()

```

```

        pid = self.dbg.load(self.exe_path)
        self.pid = self.dbg.pid

        self.dbg.run()
❷ exe_path = "C:\\WINDOWS\\System32\\calc.exe"
    snapshotter(exe_path)

```

So the first step ❶ is to start the target application under a debugger thread. By using separate threads, we can enter snapshotting commands without forcing the target application to pause while it waits for our input. Once the debugger thread has returned a valid PID ❹, we start up a new thread that will take our input ❷. Then when we send it a command, it will evaluate whether we are taking a snapshot, restoring a snapshot, or quitting ❸—pretty straightforward. The reason I picked Calculator as an example application ❺ is that we can actually see this snapshotting process in action. Enter a bunch of random math operations into the calculator, enter **snap** into our Python script, and then do some more math or hit the Clear button. Then simply type **restore** into our Python script, and you should see the numbers revert to our original snapshot point! Using this technique you can walk through and rewind certain parts of a process that are of interest without having to restart the process and get it to that exact state again. Now let's combine some of our new PyDbg techniques to create a fuzzing assistance tool that will help us find vulnerabilities in software applications and automate crash handling.

4.3.2 Putting It All Together

Now that we have covered some of the most useful features of PyDbg, we will build a utility program to help root out (pun intended) exploitable flaws in software applications. Certain function calls are more prone to buffer overflows, format string vulnerabilities, and memory corruption. We want to pay particular attention to these dangerous functions.

The tool will locate the dangerous function calls and track hits to those functions. When a function that we deemed to be dangerous gets called, we will dereference four parameters off the stack (as well as the return address of the caller) and snapshot the process in case that function causes an overflow condition. If there is an access violation, our script will rewind the process to the last dangerous function hit. From there it single-steps the target application and disassembles each instruction until we either throw the access violation again or hit the maximum number of instructions we want to inspect. Anytime you see a hit on a dangerous function that matches data you have sent to the application, it is worth taking a look at whether you can manipulate the data to crash the application. This is the first step toward creating an exploit.

Warm up your coding fingers, open a new Python script called *danger_track.py*, and enter the following code.

danger_track.py

```
from pydbg import *
from pydbg.defines import *

import utils

# This is the maximum number of instructions we will log
# after an access violation
MAX_INSTRUCTIONS = 10

# This is far from an exhaustive list; add more for bonus points
dangerous_functions = {
    "strcpy" : "msvcrt.dll",
    "strncpy" : "msvcrt.dll",
    "sprintf" : "msvcrt.dll",
    "vsprintf": "msvcrt.dll"
}

dangerous_functions_resolved = {}
crash_encountered = False
instruction_count = 0

def danger_handler(dbg):

    # We want to print out the contents of the stack; that's about it
    # Generally there are only going to be a few parameters, so we will
    # take everything from ESP to ESP+20, which should give us enough
    # information to determine if we own any of the data
    esp_offset = 0
    print "[*] Hit %s" % dangerous_functions_resolved[dbg.context.Eip]
    print "======"

    while esp_offset <= 20:
        parameter = dbg.smart_dereference(dbg.context.Esp + esp_offset)
        print "[ESP + %d] => %s" % (esp_offset, parameter)
        esp_offset += 4

    print "=====\n"

    dbg.suspend_all_threads()
    dbg.process_snapshot()
    dbg.resume_all_threads()

    return DBG_CONTINUE

def access_violation_handler(dbg):
    global crash_encountered

    # Something bad happened, which means something good happened :)
    # Let's handle the access violation and then restore the process
    # back to the last dangerous function that was called

    if dbg.dbg.u.Exception.dwFirstChance:
```

```

        return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    if crash_encountered == False:
        dbg.suspend_all_threads()
        dbg.process_restore()
        crash_encountered = True

    # We flag each thread to single step
    for thread_id in dbg.enumerate_threads():

        print "[*] Setting single step for thread: 0x%08x" % thread_id
        h_thread = dbg.open_thread(thread_id)
        dbg.single_step(True, h_thread)
        dbg.close_handle(h_thread)

    # Now resume execution, which will pass control to our
    # single step handler
    dbg.resume_all_threads()

    return DBG_CONTINUE
else:
    dbg.terminate_process()

return DBG_EXCEPTION_NOT_HANDLED

def single_step_handler(dbg):
    global instruction_count
    global crash_encountered

    if crash_encountered:

        if instruction_count == MAX_INSTRUCTIONS:

            dbg.single_step(False)
            return DBG_CONTINUE
        else:

            # Disassemble this instruction
            instruction = dbg.disasm(dbg.context.Eip)
            print "%d\t0x%08x : %s" % (instruction_count, dbg.context.Eip,
                instruction)
            instruction_count += 1
            dbg.single_step(True)

    return DBG_CONTINUE

dbg = pydbg()

pid = int(raw_input("Enter the PID you wish to monitor: "))

```

```

dbg.attach(pid)

# Track down all of the dangerous functions and set breakpoints
for func in dangerous_functions.keys():

    func_address = dbg.func_resolve( dangerous_functions[func],func )
    print "[*] Resolved breakpoint: %s -> 0x%08x" % ( func, func_address )
    dbg.bp_set( func_address, handler = danger_handler )
    dangerous_functions_resolved[func_address] = func

dbg.set_callback( EXCEPTION_ACCESS_VIOLATION, access_violation_handler )
dbg.set_callback( EXCEPTION_SINGLE_STEP, single_step_handler )
dbg.run()

```

There should be no big surprises in the preceding code block, as we have covered most of the concepts in our previous PyDbg endeavors. The best way to test the effectiveness of this script is to pick a software application that is known to have a vulnerability,² attach the script, and then send the required input to crash the application.

We have taken a solid tour of PyDbg and a subset of the features it provides. As you can see, the ability to script a debugger is extremely powerful and lends itself well to automation tasks. The only downside to this method is that for every piece of information you wish to obtain, you have to write code to do it. This is where our next tool, Immunity Debugger, bridges the gap between a scripted debugger and a graphical debugger you can interact with. Let's carry on.

² A classic stack-based overflow can be found in WarFTP 1.65. You can still download this FTP server from <http://support.jgaa.com/index.php?cmd=DownloadVersion&ID=1>.