

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.

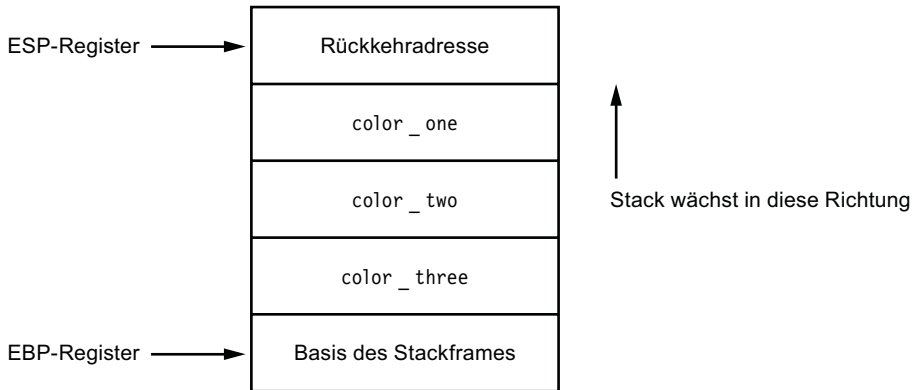


Abb. 2–1 Stackframe für den `my_socks()`-Funktionsaufruf

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.

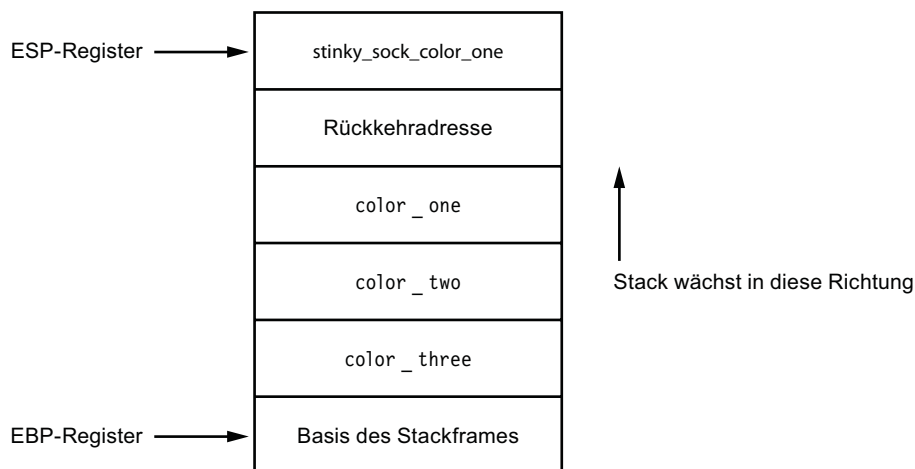


Abb. 2-2 Stackframe nach der Allokation der lokalen Variablen `stinky_sock_color_one`

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:

```
8BC3
```

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.

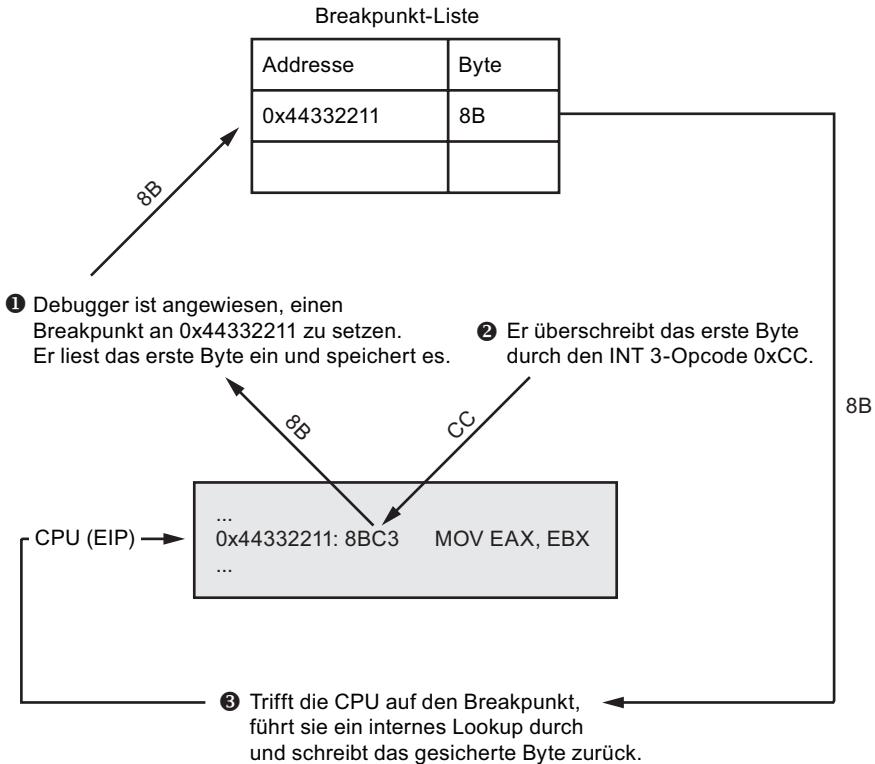


Abb. 2-3 *Setzen eines Software-Breakpunkts*

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.

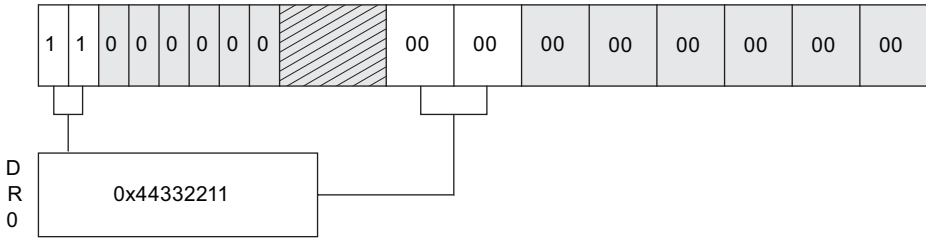
Im Gegensatz zu Software-Breakpunkten, die das INT3-Event verwenden, arbeiten Hardware-Breakpunkte mit Interrupt 1 (INT1). Das INT1-Event ist für Hardware-Breakpunkte und Einzelschritt-Events gedacht. *Einzelschritt* (*single-step*) bedeutet einfach die schrittweise Verarbeitung von Instruktionen, was 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-Registern 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-Registern enthalten, führt die CPU die Instruktion aus und macht dann mit der nächsten Instruktion weiter, führt die Prüfung erneut durch und so weiter.

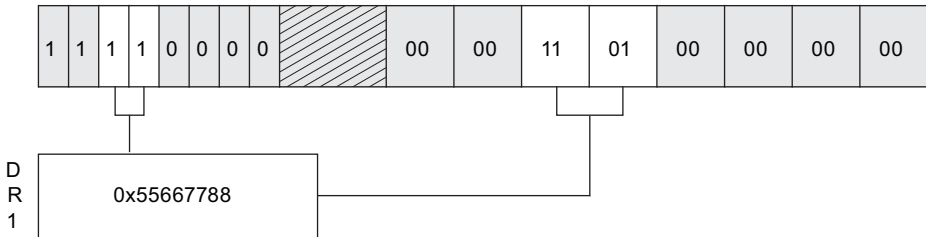
Layout des DR7-Registers

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

DR7 mit gesetztem 1-Byte-Ausführungs-Breakpunkt an 0x44332211



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



Breakpunkt-Flags

Breakpunkt-Längen-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)

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.

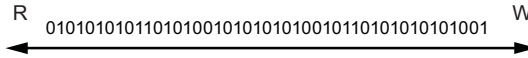
- **Guard Page**

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.



Jede Art des Datenzugriffs auf eine Guard Page löst eine Ausnahme aus. Die eigentliche Datenoperation schlägt fehl.

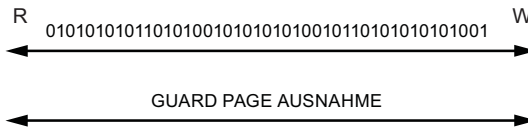


Abb. 2-5 Das Verhalten der verschiedenen Speicherseiten-Zugriffsrechte

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.