7. Wie man sich gegen Knackmodule schützt

7.1 Einleitung

Im Zuge der in allen Bereichen der Technik um sich greifenden Rationalisierung hat man es vor einigen Monaten geschafft, den ersten Schritt zur Wegrationalisierung der "Knacker" zu tun. Damals erschienen die ersten vollautomatischen Knackmodule. Diese Module bestehen meistens aus einer in ein Gehäuse gepackten Platine mit einem Druckknopf oder Schalter. Man braucht nur das Modul in den Modulport des C64 einzustecken, dann das geschützte Programm zu laden, und nachdem dieses Programm seinen Kopierschutz abgefragt hat, den Knopf zu betätigen. Das Modul speichert dann den gesamten Speicherinhalt des Rechners, oft sogar gepackt und mit einem Schnellader versehen, auf einer Diskette oder Kassette ab, selbstverständlich ohne Kopierschutz. Wenn man das Programm wieder einlädt und startet, wird der Rechner in denselben Zustand wie zum Zeitpunkt des Knopfdruckes versetzt. Wie man sein Programm gegen solche Module sichert, soll in diesem Kapitel beschrieben werden.

7.2 Vorläufer der Knackmodule

Wie kam es überhaupt zu der Entwicklung von Knackmodulen? Viele Raubkopierer, denen es zu mühsam war, sich um einen Programmschutz herumzuwinden, gingen dazu über, mit Hilfe eines RESET-Schalters aus dem laufenden Programm herauszuspringen und die benutzten Teile des Speichers abzuspeichern. Sie brauchten dann nur noch die Startadresse herauszusuchen, und schon war die Kopie fertig. Die Programmierer, die das störte, brachten in ihren Programmen eine 'CBM80'-Kennung unter, wodurch ein RESET wirkungslos wurde. An dieser Stelle tauchten die ersten Module und Betriebssysteme auf, die speziell das "Knacken" unterstützen sollten. Mit ihnen war es möglich, die 'CBM80'-Kennung zu umgehen. Einige unterstützten sogar

das Sichern von Speicherbereichen, die normalerweise bei einem RESET grundsätzlich gelöscht wurden. Folgende Bereiche sind davon betroffen:

```
$0000-$0101 =     0-  257  Systemadressen
$0200-$0802 =   512- 2048  Systemadressen und Bildschirm
$A000       = 40960        durch RAM-Test gelöscht
$D000-$DD0F = 53248-56591  I/O-Bereich
$FD30-$FD4F = 64816-64847  beim Initialisieren der Vektoren
```

Gegen die meisten "Knack"-Betriebssysteme kann man sich schon dadurch schützen, indem man wichtige Programmteile im Bereich $0400 (=1024) bis $07FF (=2048) unterbringt. Dazu muß man aber den Bildschirmspeicher, der normalerweise ab $0400 liegt, in einen freien Teil des Speichers verschieben. Folgendes Programm legt den Bildschirmspeicher nach $CC00 und den Anfang des BASIC-Speichers nach $0400, was zwei Effekte hat: Einerseits erhält man so ein Kilobyte mehr Speicherplatz für seine Programme, andererseits wird bei einem RESET der Anfang des BASIC-Programms unwiederbringlich gelöscht.

```
100 FORI=1TO66STEP15:FORJ=0TO14:READA$:B$=RIGHT$(A$,1)
105 A=ASC(A$)-48:IFA>9THENA=A-7
110 B=ASC(B$)-48:IFB>9THENB=B-7
120 A=A*16+B:C=(C+A)AND255:POKE49151+I+J,A
125 NEXT:READA:IFC=ATHENC=0:NEXT:SYS49152:NEW
130 PRINT"FEHLER IN ZEILE:";PEEK(63)+PEEK(64)*256:STOP
300 DATA78,A2,33,86,01,A0,00,84,FB,A9,D0,85,FC,B1,FB, 153
301 DATAE6,01,91,FB,86,01,C8,D0,F5,E6,FC,A5,FC,C9,E0, 179
302 DATAD0,ED,A9,37,85,01,58,AD,00,DD,29,FC,8D,00,DD, 148
303 DATAA9,35,8D,18,D0,A9,CC,8D,88,02,8C,00,04,C8,84, 187
304 DATA2B,A9,04,85,2C,A9,93,4C,D2,FF,A4,A4,A4,A4,A4, 22
```

Assemblerlisting:

```
C000 SEI          Interrupts sperren
C001 LDX #$33     Speicherkonfiguration umstellen:
C003 STX $01      Zeichen-ROM einschalten
```



```
C005 LDY #$00     Schleifenzähler $FB/$FC auf
C007 STY $FB      $D000 setzen
C009 LDA #$D0
C00B STA $FC
C00D LDA ($FB),Y  Byte aus Zeichen-ROM holen
C00F INC $01      RAM in $D000 einschalten
C011 STA ($FB),Y  Byte in RAM schreiben
C013 STX $01      Zeichen-ROM einschalten
C015 INY          Schleifenzähler LOW erhöhen
C016 BNE $C00D    verzweige, wenn ungleich null
C018 INC $FC      Schleifenzähler HIGH erhöhen
C01A LDA $FC      schon den
C01C CMP #$E0     Bereich $E000 erreicht?
C01E BNE $C00D    verzweige, wenn nein
C020 LDA #$37     alte Speicherkonfiguration
C022 STA $01      wieder herstellen
C024 CLI          Interrupts zulassen
C025 LDA $DD00    VIC auf
C028 AND #$FC     obersten 16K-Bereich
C02A STA $DD00    einstellen
C02D LDA #$35     Bildschirmspeicher bei $CC00 und
C02F STA $D018    Zeichengenerator bei $D000
C032 LDA #$CC     Adresse des Bildschirms dem
C034 STA $0288    Betriebssystem mitteilen
C037 STY $0400    Null an neuen BASIC-Speicher-Anfang
C03A INY          $0401 in
C03B STY $2B      den Zeiger auf den BASIC-Anfang
C03D LDA #$04     $2B/$2C schreiben
C03F STA $2C
C041 LDA #$93     $93 (=147) ist ASCII-Wert für CLR/HOME
C043 JMP $FFD2    BASOUT, hier: Bildschirm löschen
```

Am besten arbeiten Sie mit dem Programm so, daß Sie den NEW-Befehl in Zeile 125 durch einen LOAD-Befehl ersetzen, der das eigentliche Hauptprogramm in den Speicher bringt. Dieses sollte als erstes den Befehl 'CLR' ausführen, damit keine Probleme mit Variablenüberlappungen auftreten können. Außerdem darf dieses Programm ohne das vorhergehende Ladeprogramm nicht lauffähig sein, was sich am einfachsten durch die

Übergabe eines Wertes von dem ersten Programm an das zweite per POKE in eine freie Speicherstelle realisieren läßt. Das ganze könnte zum Beispiel so aussehen: Nach erfolgter Kopierschutzabfrage in dem Teil, der den DATA-Lader enthält, wird der Befehl 'POKE 2,123' benutzt. Im Hauptprogramm wird dann abgefragt, ob der Wert der Speicherstelle Zwei wirklich gleich 123 ist. Falls nein, wird das Programm abgebrochen. Es ist wohl unnötig, zu erwähnen, daß Sie daran denken, Ihr Programm gegen die Einblicke anderer zu schützen.

Gegen gute Knackmodule wird man mit dieser Methode allerdings nicht sehr weit kommen, da diese immer über ein eigenes RAM verfügen, in das sie die Speicherbereiche, die sie benötigen, hinüberretten, also auch den Bildschirmspeicher. Um sich auch hiervor zu sichern, muß man schon schwerere Geschütze auffahren.

7.2 Knackmodule neueren Datums

Alle Schutzmethoden gegen Knackmodule haben eines gemeinsam: sie enthalten eine periodisch oder auch nur an wichtigen Stellen des Programms aufgerufene Schutzabfrage, da ein nur einmalig abgefragter Kopierschutz gegen ein solches Modul sinnlos ist. Es hängt unter Umständen stark von dem Programm ab, wie man diese Abfrage installiert. Zum Beispiel könnte man in einem "Adventure" alle 100 Schritte den Spieler fragen, welches Wort sich auf einer bestimmten Seite der Anleitung an einer bestimmten Stelle befindet. Allerdings lassen sich Anleitungen ebenfalls kopieren, wenn auch meist mit größerem Aufwand als Programme. Daher gehen einige Firmen dazu über, ihren Programmen eine spezielle Linse beizufügen, ohne die es nicht möglich ist, die von Zeit zu Zeit verschlüsselt angezeigten Buchstaben zu entziffern. Solche Schutzsysteme stellen nicht nur ein Handikap gegenüber den "Knackern" dar, sondern auch eine Zumutung gegenüber dem Anwender. Außerdem animieren so geschützte Programme gerade dazu, den Schutz zu entfernen, da eine ungeschützte Kopie wesentlich anwenderfreundlicher ist.

Sinnvoller und mit Sicherheit von keinem Knackmodul zu kopieren ist das mehrfache Abfragen des auf der Diskette angebrachten Schutzes. Dies sollte am besten dann geschehen, wenn das Programm Daten nachlädt. Falls man größeren Aufwand treiben will, kann man auch ein komplett neues Diskettenformat verwenden. Man sollte aber darauf achten, daß der Anwender davon nicht betroffen wird, also seine geschützten Disketten nicht öfter wechseln muß als die ungeschützten.

Was aber, wenn Ihr Programm nach Einladen des Hauptprogramms nur noch mit einer Datendiskette arbeitet und die Originaldiskette nicht mehr benötigt? Oder was, wenn Sie Ihr Programm gar nicht auf einer Diskette unterbringen, sondern auf einer Kassette?

Betrachten wir zuerst den Fall, daß eine Diskettenstation vorhanden ist. Dann ist es am einfachsten, nach erfolgter Kopierschutzabfrage einige Bytes, wenn nicht sogar ein komplettes Programm, im Speicher der Floppy abzulegen und regelmäßig das Vorhandensein dieser Bytes abzufragen. Bei normalem Diskettenbetrieb bleiben folgende Speicherstellen ungenutzt:

Hexadezimal	Dezimal
0005	5
0010 / 0011	16 / 17
0014 / 0015	20 / 21
001B	27
001D	29
001F	31
0021	33
0023	35
0035	53
0037	55
003B / 003C	59 / 60
0046	70
0096	150
0100	256
0102 / 0103	258 / 259
02FB	763
02FD	765

Mit folgendem Befehl schreibt man ein oder mehrere Bytes in eine bestimmte Speicheradresse der Floppy:

```
OPEN 1,8,15,"M-W"+CHR$(LOW-Byte)+CHR$(HIGH-Byte)
+CHR$(Anzahl)+CHR$(Byte1)+CHR$(Byte2)+...:CLOSE 1
```

Die Bezeichnungen LOW- und HIGH-Byte beziehen sich auf die Anfangsadresse des zu belegenden Bereichs. Beispiel: Sie wollen in die Speicherstelle 258 den Wert 123 eintragen. Die Anzahl der zu sendenden Bytes beträgt eins, das LOW-Byte von 258 (=$0102) ist zwei, das HIGH-Byte eins. Der Befehl sieht dann so aus:

```
OPEN 1,8,15,"M-W"+CHR$(2)+CHR$(1)+CHR$(1)+CHR$(123):CLOSE 1
```

Um den Inhalt einer Speicherstelle wieder auszulesen, benötigt man folgenden Befehl:

```
OPEN 1,8,15,"M-R"+CHR$(LOW-Byte)+CHR$(HIGH-Byte)
GET #1,A$:A=ASC(A$+CHR$(0)):CLOSE 1
```

A enthält dann den Wert der angesprochenen Speicherstelle. Zu dem oben angegebenen Beispiel sieht die Abfrage dann so aus:

```
OPEN 1,8,15,"M-R"+CHR$(2)+CHR$(1)
GET #1,A$:A=ASC(A$+CHR$(0)):CLOSE 1
```

A muß dann den Wert 123 enthalten.

Der Schutz beruht einfach darauf, daß Knackmodule den Inhalt des Floppy-Speichers nicht sichern. Denken Sie bitte immer daran, daß das Schreiben des Bytes nur ein einziges Mal im Programm vorkommen darf, nämlich bei der Kopierschutzabfrage, und daß der Test, ob das Byte vorhanden ist, regelmäßig durchzuführen ist, am besten vor dem Aufruf oft gebrauchter Programmteile.

Was jedoch läßt sich tun, wenn Sie keine Diskettenstation zur Verfügung haben? Gibt es eine Methode, Knackmodule auszutricksen, die völlig speicherintern abläuft? Die Antwort ist: Ja, es gibt sie. Die Idee ist einfach die, daß im C64 einige Bausteine vorhanden sind, in deren Register man zwar einen Wert hineinschreiben kann, aber aus denen sich dieser Wert nicht direkt wieder auslesen läßt.

Das erste Beispiel, das wir in diesem Zusammenhang betrachten, ist der Soundchip des C64, der sog. *SID*. Dort läßt sich für jede der drei erzeugbaren Stimmen eine Hüllkurve festlegen. Dabei handelt es sich um den Lautstärkeverlauf eines gespielten Tones. Normalerweise kann man die einmal in ein Register geschriebenen Hüllkurvenwerte nicht wieder auslesen, was das Knackmodul daran hindert, diese Werte zu kopieren. Dummerweise bedeutet das aber auch, daß man selbst nicht testen kann, ob die Werte stimmen. Für Stimme Drei existiert allerdings ein Leseregister, das es ermöglicht, den Lautstärkeverlauf eines gerade gespielten Tones mitzuverfolgen. Die Abfrage der Hüllkurve geschieht also einfach durch Anspielen dieser Stimme. Für diesen Schutz benötigen wir folgende Register:

Hex	Dezimal	Funktion
$D412	54290	Steuerregister für Stimme Drei. Wird in diesem Register das Bit 1 von Null auf Eins geschaltet, so wird ein Ton gepielt.
$D413	54291	Attack/Decay: Bits 0 bis 3: Zeit, in der Lautstärke vom Maximum auf den Sustain-Pegel abfällt. Bits 4 bis 7: Zeit, in der Lautstärke von Null auf das Maximum ansteigt.
$D414	54292	Sustain/Release: Bits 0 bis 3: Zeit, in der Lautstärke vom Sustain-Pegel auf Null abfällt. Bits 4 bis 7: Sustain-Pegel: Lautstärke, die kurz nach Anspielen eines Tones erreicht wird.
$D41C	54300	augenblickliche Lautstärke der Stimme Drei

Beim Anspielen der Stimme Drei braucht man weder die Gesamtlautstärke des SIDs noch eine Wellenform für diese Stimme einzuschalten, wodurch der Ton unhörbar bleibt. Am einfachsten ist es, die Attack-, Decay- und Releasezeiten auf null zu stellen und den Sustainpegel auf einen Wert, der sich durch Auslesen von $D41C (=54300) feststellen läßt. Das Einstellen könnte dann so aussehen:

```
10 A=10:REM (änderbarer) Sustain-Wert
20 POKE54291,0:POKE54292,A*16
```

Setzt man in Zeile 10 A auf einen anderen Wert zwischen 0 und 15, erhält man dementsprechend einen anderen Sustain-Pegel. Gleiches gilt auch für die Abfrage:

```
10 A=10:REM (AENDERBRER) SUSTAIN-WERT
20 POKE 54290,0:POKE 54290,1:REM TON SPIELEN
30 FORI=1TO50:NEXT:REM ATTACK/DECAY-ZEIT ABWARTEN
40 IF INT(PEEK(54300)/16)<>A THEN PRINT "DU RAUBKOPIERER!"
```

Sie können selbstverständlich auch kompliziertere Hüllkurven programmieren und abfragen. Allerdings wird durch diesen Schutz die Stimme Drei belegt. Was aber, wenn man die Schutzabfrage zu einem Zeitpunkt benötigt, bei dem gerade diese Stimme schon anderweitig benutzt wird? Nun, es gibt noch andere Register, die sich normal nicht auslesen lassen, deren Wert man aber mit einem Trick erfahren kann: die Alarmzeiten der Echtzeituhren.

Der C64 besitzt zwei Chips, genannt CIAs, die beide eine Echtzeituhr besitzen. Die folgenden Erklärungen beziehen sich nur auf den ersten der beiden Chips, dessen Basisadresse bei $DC00 (=56320) liegt. Wenn Sie den anderen Chip benutzen wollen, so denken Sie bitte daran, daß sich dessen Basisadresse bei $DD00 (=56576) befindet.

Der Trick, mit dem man die Alarmzeiten überprüfen kann, ist einfach der, daß man die Echtzeituhr auf einen Wert knapp vor dem erwarteten Alarm setzt und kontrolliert, ob der Alarm kurz darauf ausgelöst wird. Dieser Schutz ist besonders schwer zu kopieren. Bei der Hüllkurve wäre es ja denkbar, daß das Knackmodul ebenfalls die Stimme Drei spielt und aufgrund des ausgelesenen Lautstärkeverlaufs die Werte für Attack, Decay, Sustain und Release herausfindet. Bei der Echtzeituhr müßte man aber, wenn man die Alarmzeit nicht schon kennt, die Uhr bis zu 24 Stunden laufen lassen, um die Zeit zu rekonstruieren.

Folgende Register braucht man zur Programmierung der Uhr:

Hex	Dezimal	Funktion
$DC08	56328	Bit 0 bis 3: Zehntelsekunden
		Bit 4 bis 7: müssen null sein
$DC09	56329	Bit 0 bis 3: Einersekunden
		Bit 4 bis 6: Zehnersekunden
		Bit 7: muß null sein
$DC0A	56330	Bit 0 bis 3: Einerminuten
		Bit 4 bis 6: Zehnerminuten
		Bit 7: muß null sein
$DC0B	56331	Bit 0 bis 3: Einerstunden
		Bit 4: Zehnerstunde
		Bit 5 bis 6: müssen null sein
		Bit 7: 0=vormittags (AM)
		1=nachmittags (PM)
$DC0D	56333	Bit 2: 1=Gleichheit von Uhrzeit und Alarmzeit; Achtung: beim Lesen dieses Registers werden alle Bits gelöscht.
$DC0F	56335	Bit 7: 0=Schreibzugriffe auf die vorhergehenden Register beziehen sich auf die Echtzeituhr. 1=Schreibzugriffe auf die vorhergehenden Register beziehen sich auf die Alarmzeit

Die Echtzeituhr wird gestellt, indem man Bit 7 von $DC0F auf null setzt und dann die Uhrzeit, beginnend mit der Stunde, in die Register einträgt. Die Uhr hält beim Ansprechen des Stundenregisters an und läuft erst beim Setzen der Zehntelsekunden weiter. Entsprechend sollte man beim Auslesen ebenfalls zuerst auf das Stundenregister zugreifen, da dann die Uhr scheinbar bis zum Lesen der Zehntelsekunden gestoppt wird. Intern läuft sie allerdings weiter. Die Alarmzeit wird genauso festgesetzt, nur muß Bit 7 von $DC0F den Wert eins haben.

Ein Programm, das beispielsweise die Alarmzeit 7 Uhr 35 Minuten 11 Sekunden und 1 Zehntelsekunde nachmittags setzen soll, sieht so aus:

```
10 POKE56335,PEEK(56335)OR128:REM BIT 7 setzen
20 POKE56331,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330,3*16+5:REM 35 MINUTEN
40 POKE56329,1*16+1:REM 11 SEKUNDEN
50 POKE56328,1:REM 1 ZEHNTELSEKUNDE
```

Beim Test auf die Alarmzeit stellen wir die Uhr auf eine Zehntelsekunde vor der erwarteten Zeit, warten diese Zehntelsekunde ab und testen dann Bit 2 von Register $DC0D. Da dieses Register beim ersten Lesezugriff wieder gelöscht wird, müssen wir verhindern, daß das Betriebssystem nach erfolgtem Alarm noch vor uns auf $DC0D zugreift. Man erreicht das durch Abschalten des Timer-IRQs (siehe auch: CIA-Beschreibung im Teil "Kassettenkopierschutz"), indem man nach $DC0D (=56333) den Wert 1 schreibt. Hinterher muß man den Timer-IRQ durch 'POKE 56333,129' wieder einschalten. Bei der CIA zwei ($DD00) kann man sich diese Prozedur sparen, da sie nicht vom Betriebssystem benutzt wird.

Hier also das Programm, welches die oben eingetragene Alarmzeit testet:

```
10 POKE56335,PEEK(56335)AND128:Bit 7 LOESCHEN
20 POKE56331,128+7:REM 7 UHR NACHMITTAGS
30 POKE56330,3*16+5:REM 35 MINUTEN
40 POKE56329,1*16+1:REM 11 SEKUNDEN
50 POKE56328,0:REM 0 ZEHNTELSEKUNDEN
60 POKE56333,1:REM TIMER-IRQ ABSCHALTEN
70 FORI=1TO200:NEXT:REM ALARM ABWARTEN
80 A=PEEK(56333):POKE56333,129:REM ALARM TESTEN, IRQ EIN
90 IF (A AND 4)<>4 THEN PRINT"VERFLIXT, EIN KNACKMODUL!"
```


Zur Benutzung dieser Programme gilt das gleiche wie bei den vorhergehenden Beispielen. Die Alarm-Setz-Routine sollte ein einziges Mal in Verbindung mit der Kopierschutzabfrage ausgeführt werden, die Testroutine dagegen an allen wichtigen Stellen im Programm. Denken Sie daran, daß ein guter Kopierschutz nur in Verbindung mit einem mindestens genauso guten Programmschutz sinnvoll ist. Es gibt schließlich noch genügend "Knacker", die sich nicht auf ein Knackmodul verlassen. Gerade in BASIC-Programmen lassen sich besonders leicht Schutzabfragen finden. Kompilieren Sie daher nach Möglichkeit Ihr Programm, um es den Raubkopierern nicht leichter zu machen als nötig.

7.3 Knackmodule: Zukunftsaussichten

Es ist abzusehen, daß in Zukunft Knackmodule auf dem Markt erscheinen werden, die sich durch die beschriebenen speicherinternen Tricks nicht abschrecken lassen. Theoretisch könnte nämlich ein Modul den Adreß- und Datenbus des C64 überwachen und so alle Schreibzugriffe auf nicht lesbare Register zusätzlich in einem eigenen RAM nachvollziehen. Außerdem ist ein Modul-System denkbar, welches das Sichern des Floppy-Speichers ermöglicht. In beiden Fällen ist aber der notwendige Hardware-Aufwand wesentlich höher als bei den jetzigen Knackmodulen. Der sicherste Schutz ist und bleibt daher eine mehrfache Abfrage der Originaldiskette. Es bleibt dabei nur zu hoffen, daß nicht doch eines Tages das "alles kopierende" Kopierprogramm entwickelt wird.