# Malicious Code: Code Obfuscation

**Sebastian Strohhäcker**

**Technische Universität München**

**Zusammenfassung** Dieses Dokument beschreibt Techniken zum Schutz von Algorithmen in Programmen sowie Programmteilen anhand von Codebeispielen. Es werden sowohl Methoden der Programmtransformation als auch Transformationen selbst erörtert.

## 1 Einleitung

Es existieren verschiedene Methoden, ein Programm sowohl vor Analyse als auch Modifikation zu schützen:

- Transformation von Programmteilen
- Verschlüsselung
- spezifisches Verhindern von Debugging, Decompiling, Disassembling und Code Dumping

Durch die Transformation von Programmteilen kann die Analyse eines darin enthaltenen Algorithmus erheblich erschwert werden. Die Transformation eines für eine reale Maschine übersetzten Programms ist jedoch sehr komplex und kann die Geschwindigkeit beeinflussen.
Das Verschlüsseln kann ohne Bezug zum Quelltext eingesetzt werden, in der Regel wird dem Originalprogramm ein Lader hinzugefügt, der das Programm zur Laufzeit entschlüsselt. Somit wird (bis auf den Progammstart) die Geschwindigkeit des Programms nicht beeinflusst. Ein Disassembly eines verschlüsselten Programms ist von geringem Wert.
Techniken wie z.B. Debugging können durch das Ausnutzen unterschiedlichen Verhaltens eines Programms in einem Debugger gezielt erschwert werden, insbesondere Fehlverhalten erweist sich als geeignet.
Kombinationen der Methoden erhöhen den Schutz, z.B. kann der Lader der Verschlüsselung transformiert und gegen Disassembling geschützt sein, sowie der Schlüssel von der Präsenz eines Debuggers abhängen.

## 2 Programmtransformation

### 2.1 Integration in Programmcode

Die in diesem Abschnitt beschriebenen `obfuscating instructions` können manuell durch Makros in Sourcecode eingebunden werden, z.B. kann für das Auslesen und Ändern einer Variablen ein Makroaufruf genutzt werden, die Makros

selbst enthalten entsprechend transformierten Code.
Um eine automatische Transformation zu ermöglichen ist eine komplexe Analyse des Programmflusses nötig, es kann auch ein modifizierter Compiler verwendet werden (hochoptimierende Compiler erzeugen meist Code, der in hohem Maße transformiert ist). Weiter ist dynamische Generierung von Code möglich, wobei Instruktionen erst zur Laufzeit spezifiziert werden (siehe auch Abschnitt 4.2).

### 2.2 Befehle in anderen Befehlen

Instruktionen, die aus mehreren Bytes bestehen, können dazu verwendet werden, andere Befehle zu verstecken. Dabei muss die originale Instruktion nicht ausgeführt werden, sondern verhindert lediglich das Lesen der enthaltenen Befehle (Schutz vor Disassembling). Auftretende Seiteneffekte sind zu berücksichtigen.

```
0x100:  b8 eb 03         mov  ax, 0x3eb
0x103:  eb fc            jmp  0x101
0x105:  9a e9 f7 00 01   call 0x100:0xf7e9
```

Dabei wird in Zeile 0x103 nach 0x101 gesprungen:

```
0x101:  eb 03            jmp  0x106
0x103:  eb fc            jmp  0x101
0x105:  9a e9 f7 00 01   call 0x100:0xf7e9
```

Der call-Befehl wird nie ausgeführt, stattdessen wird weiter zu 0x106 gesprungen:

```
0x106:  e9 f7 00         jmp  0x200
```

### 2.3 Codepfad beeinflussen

Der folgende Codeausschnitt kann *jmp _addr_* ersetzen. Der *call*-Befehl legt den Wert des Instruktionszählers (0x106) auf den Stack, danach wird er in *ax* geladen (es wird kein zugehöriges *ret* verwendet).

```
0x100:  push _addr_
0x103:  call 0x106
0x106:  pop  ax
0x107:  jmp  ax
```

Obiges Beispiel kann erweitert werden, um einen schwer zu analysierenden Codefluss zu erzeugen. Nachfolgend springt *ret* (Adresse 0x10e) an die durch *ax* gegebene Adresse, zuerst nach 0x10a (0x106+4), danach zur Adresse 0x10e (0x106+4+4), zuletzt nach *_addr_*. Adresse 0x109 kann eine überdeckende Instruktion enthalten (siehe Abschnitt 2.2).

```
0x100:  push _addr_
0x103:  call 0x106
0x106:  pop  ax
0x107:  jmp  0x10a
0x109:  nop
0x10a:  add  ax, 4
0x10d:  push ax
0x10e:  ret
```

Im folgenden Ausschnitt wird an Adresse 0x108 der auf dem Stack bei *[bp]* liegende Wert erhöht, dadurch springt *ret* zuerst nach 0x108 (eine weitere Erhöhung von *[bp]*), danach zu 0x10e, von dort nach *_addr_*. Der Befehl bei 0x10c bewirkt, dass der Stack stets die gleiche Höhe hat.

```
0x100:  push _addr_
0x103:  push 0x102
0x106:  mov  bp, sp
0x108:  add  ss:[bp], 6
0x10c:  mov  sp, bp
0x10e:  ret
```

Wenn im nächsten Codeabschnitt durch *ret* nach *_addr_* gesprungen wird, enthält *ax* den Wert 0x10e. Falls dort ein definierter Wert für *ax* nötig ist, kann statt eines *mov*-Befehls z.B. *add* verwendet werden.

```
0x100:  mov  ax, 0x10e
0x103:  push _addr_
0x106:  call 0x109
0x109:  dec  ax
0x10a:  call ax
0x10c:  inc  ax
0x10d:  ret
```

### 2.4 Selbstmodifizierender Code

Effekte von Code, der sich selbst ändert, sind schwer zu analysieren. Im folgenden wird die *call*-Instruktion durch einen *jmp* überschrieben.

```
0x100:  mov  cs:[0x107], 0x00eb (modifiziere Adresse 0x107)
0x107:  call 0x107
```

...wird zu...

```
0x100:  mov  cs:[0x107], 0x00eb
0x107:  jmp  0x109
```

Selbstmodifizierender Code kann dazu verwendet werden, Dumping zu erschweren. Im nächsten Ausschnitt ist der Wert von *ax*, wenn *call 0x113* in Zeile 0x100 aufgerufen wird, gleich 0x100. Wird danach das Programm durch einen Dump-Prozess gesichert, enthält das Ausführen dieses Programms an Adresse 0x100 bereits *call 0x113*, so dass *ax* einen anderen Wert aufweist.

```
0x100:  call 0x103
0x103:  pop  ax
0x104:  sub  ax, 3
0x107:  add  cs:[0x101], 0x10
0x10d:  jmp  ax
```

...wird zu...

```
0x100:  call 0x113
```

Falls bestimmte Bereiche des Codes nicht mehr besucht werden (z.B. Initialisierungen), können diese zum Speichern von Daten oder für dynamische Codegenerierung genutzt werden. Vorteilhaft ist ein in Abschnitte unterteilter ausführbarer Bereich (Blockweise komprimiert oder verschlüsselt), wobei nur der jeweils benötigte Code geladen ist (siehe auch Abschnitt 4).

### 2.5 Single-Stepping

Bei gesetztem Trap-Flag wird die Befehlsausführung unterbrochen und Interrupt 1 aufgerufen. Dort sind Manipulationen am momentan auszuführenden Code möglich, so dass z.B. Verschlüsselung eine Analyse außerordentlich schwierig gestaltet.
Single-Stepping ermöglicht, da nur jeweils eine Instruktion ausgeführt wird bevor Interrupt 1 aufgerufen wird, eine Vielzahl von Programmfluss-Modifikationen, z.B. kann der Befehlszähler je Instruktion zusätzlich um einen Wert weitergeschaltet werden.

### 2.6 Software-CPU/Scripting

Das Implementieren einer Software-CPU kann die Analyse von Programmteilen effektiv erschweren, da hierzu die Funktionsweise der CPU geklärt werden muss (siehe auch [Anormal1998]). Speziell Verschlüsselungsalgorithmen und kryptographische Algorithmen sind geeignet, in der Sprache einer Software-CPU geschrieben zu werden. Allerdings ist die Implementierung sehr aufwändig und die Programme (im der Regel erheblich) langsamer.
In Tabelle 1 ist der Befehlssatz einer CPU, die ohne Register arbeitet, abgebildet. Dabei bestehen Instruktionen aus 32 Bit (je 8 Byte für Opcode, Parameter1 und Parameter2 sowie den nächsten Wert des Instruktionszählers). Der Befehlsfluss kann sehr komplex gestaltet werden, da Instruktionen nicht aufeinander folgen sondern durch *nextip* gegeben sind.

| opcode | mnemonic | param1 | param2 | nextip | Aufgabe |
|---|---|---|---|---|---|
| 0 | xor | orig1 | dorig2 | nextip | [orig1] := [orig1] xor dorig2 |
| 2 | add | orig1 | dorig2 | nextip | [orig1] := [orig1] + dorig2 |
| 4 | mov | @orig | dest | nextip | [[orig]] := dest (doppelte Indirektion) |
| 6 | stp | nil | nil | nil | Programm beenden |
| 8 | jnz | orig1 | nextip1 | nextip2 | if [orig1]!=0 jmp ip1 else jmp ip2 |

**Tabelle 1.** Opcodes einer Software-CPU

Es folgt die Hauptschleife, welche einen Befehl einliest und ausführt, sowie die Implementierung zweier Befehle der CPU (xor und jnz):

```
initCpu:
    mov  si, offset memory  ; Instruktionszeiger initialisieren
    mov  di, si             ; Start des Speichers
    sub  bx, bx
nextCycle:
    mov  eax, [si]          ; Opcode holen
    mov  bl, al
    call [bx+offset opcodeTable]   ; Instruktion ausfuehren
    shr  eax, 8
    mov  si, ax             ; si:=nextip
    add  si, di             ; nextip relativ zu Speicherstart
    jmp  nextCycle

opXor:
    mov  bl, ah             ; Parameter 1 holen
    mov  cl, [bx+di]        ; Parameter 1 dereferenzieren
    shr  eax, 16
    mov  bl, al             ; Parameter 2 holen
xorit:
    xor  [bx+di], cl        ; xor-Operation ausfuehren
    ret

opJnz:
    mov  bl, ah             ; Parameter 1 holen
    shr  eax, 16
    cmp  [bx+di], bh        ; Parameter 1 dereferenzieren
                            ; und auf 0 (==bh) testen
    je   not0
    mov  ah, al             ; nextip aendern wegen Sprung
not0:
    ret
```

Falls auch Code, der außerhalb der Software-CPU liegt, verfügbar sein soll, können Callbacks verwendet werden (z.B. ein dedizierter Opcode mit entspre-

chenden Parametern), so dass Ein- und Ausgabe möglich ist.
Architekturelle Aspekte der CPU wie Anzahl und Aufgabe von Registern können beliebig gestaltet werden, es seien Stack-Maschinen erwähnt, die arithmetische Operationen ähnlich der FPU ausführen.
Eine Variation der Software-CPUs sind Skript-Sprachen, d.h. bestimmte Teile der Ausführung werden durch Skript-Befehle (und in das Skript integrierte Daten) gesteuert.

# 3 Debugger erkennen

## 3.1 Backdoor commands

Einige Debugger benutzen ein definiertes Interface, um mit Komponenten und Programmen zu kommunizieren. Nachfolgend wird die Schnittstelle zu BoundsChecker (BCHK) benutzt, um SoftIce zu identifizieren.

```
mov     ebp, 0x4243484B        ; 'BCHK'
mov     eax, 4
int     3
cmp     al, 4
jnz     @bad
```

## 3.2 Unemulierte Befehle

Manche Debugger/Disassembler können z.B. FPU-Befehle nicht dekodieren oder zeigen falsches Verhalten:

```
icebp (opcode 0xf1)
```

Der undokumentierte Befehl IceBP löst auf x86-CPUs Interrupt 1 aus (SingleStep), nicht Interrupt 6 (Unknown Opcode).

Auch werden Debug-Register nicht in allen Fällen korrekt emuliert:

```
mov  eax, 0xffff0000
xor  ecx, ecx
mov  dr7, eax
mov  ebx, dr7
mov  dr7, ecx
cmp  ebx, eax
je   @bad
```

In diesem Fall wird überprüft, ob bestimmte Bits des Debug Control Registers (dr7), die normalerweise nicht veränderbar sind, gesetzt werden können.

### 3.3 Structured Exception Handling

SEH wird in Windows verwendet, um es einem Programm zu ermöglichen, auf kritische Fehler (z.B. Protection Violation) zu reagieren. Viele Debugger behandeln einige der Fehler selbst, ohne den benutzerdefinierten SEH-Handler aufzurufen [Owl1998]. Dadurch ist es möglich, dass sich ein Programm bei Präsenz eines Debuggers anders verhält. Zudem kann durch Verändern des Kontextes (der als Parameter an den SEH-Handler übergeben wird) die Privilegstufe auf 0 gesetzt werden (ring0).
Die Adresse der `EXCEPTION_REGISTRATION`-Struktur, die einen Zeiger auf den vorherigen sowie den benutzerdefinierten SEH-Handler enthält, ist für jeden Prozess in fs:[0] zu finden.

```
      call inst_handler          ; Adresse von seh_handler auf Stack
seh_handler:
      mov  ebx,[esp+0ch]
      add  dword ptr [ebx+0b8h],02h          ; int 1 ueberspringen
      xor  eax,eax
      ret
inst_handler:
      push dword ptr fs:[0]   ; vorhergehenden SEH-Handler auf Stack
      mov  dword ptr fs:[0],esp           ; SEH-Handler registrieren
      xor  eax,eax
      int  1                ; Schutzverletzung (ruft SEH-Handler auf)
      inc  eax
      inc  eax
      or   eax,eax
      jz   bad
```

Durch Kontextmanipulation sind auch Techniken wie Single-Stepping (siehe Abschnitt 2.5) realisierbar, sowie das Rücksetzen von Debug-Registern.

## 4 Verschlüsselungstechniken

### 4.1 Programmvorspann

Das Verschlüsseln von kompilierten Programmen kann durch ein externes Programm vorgenommen werden. Dabei wird dem Programm ein zusätzliches Codestück vorangestellt, das zum Programmstart die Entschlüsselung vornimmt. Um ein automatisches Entfernen dieses Laders zu verhindern, kann dieser polymorph implementiert werden (siehe Abschnitt 4.2).
Die folgende Schleife benutzt den *xor*-Befehl, um einen Speicherblock zu verschlüsseln:

```
0x100:  mov  bx, _size_
0x103:  mov  di, _start_addr_
```

```
0x106:  xor  cs:[di], 97
0x10a:  inc  di
0x10b:  dec  bx
0x10c:  jnz  0x106
0x10e:  jmp  _code_start_
```

### 4.2 Polymorphismus

Polymorphe Routinen haben zu verschiedenen Zeitpunkten unterschiedliche Ausprägung des Codes, jedoch die selbe Wirkungsweise. Dadurch ist ihre Analyse wesentlich komplexer.
Ein Dekodierer, der zur Laufzeit generiert wird, kann polymorph implementiert werden, indem z.B. je Aufruf verschiedene Register verwendet werden oder andere Transformationen angewendet werden. Auch können verschiedene Basisalgorithmen zur Verfügung gestellt werden, aus denen ausgewählt wird.
Um Polymorphismus effizient implementieren zu können, werden Architekturspezifika der Zielhardware ausgenutzt. Der x86-Befehlssatz ermöglicht durch Bitfield-Layout einfache Variation der benutzten Register. Nach Tabelle 2 wird *add eax, ecx* als 0x03 0xC1 codiert, *add eax, edx* als 0x03 0xC2. Das Erzeugen polymorphen Codes kann somit die Register beliebig nutzen.

| **bits** | 76 543 210 | **bits** | 76 543 210 |
|---|---|---|---|
| | 00 abc 011 | | 11 def ghi |

| **abc**: | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| opcode: | add | or | adc | sbb | and | sub | xor | cmp |
| | | | | | | | | |
| **def/hji**: | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| reg: | eax | ecx | edx | ebx | esp | ebp | esi | edi |

**Tabelle 2.** Beispiel für Bitfield-Layout

## Literatur

[CTL1999] Christian Collberg, Clark Thomborson, Douglas Low: A Taxonomy of Obfuscating Transformations, Technical Report #148 (1999)
[Intel2004] Intel: IA-32 Software Developer's Manual (2004) http://developer.intel.com/design/Pentium4/documentation.htm
[Yoda2004] Yoda: Yoda's PE crypter source (2004) http://yodap.cjb.net
[Gabler1999] Christoph Gabler: INSIDER - FAQ Edition 9 (1999)
[Daemon2002] ˆDAEMONˆ: sice detector sourcecode (2002) http://daemon.anticrack.de/ (unavailable as of december 2004)
[Owl1998] The Owl: Inside Structured Exception Handling (1998) http://www.anticracking.sk/EliCZ/infos/SEHall.zip
[Anormal1998] anormal/kindergarten: Design your own CPU (1998) http://hackjaponaise.cosm.co.jp/archives/websites/fravia/new_anor.htm