Karsten Burger
September 2006
k.burger@burgernet.org

1. Inhaltsverzeichnis

1 Dieses Dokument

Ich möchte in diesem Dokument Anfänger im Reversing – Bereich die Thematik ein wenig näher bringen. Empfohlen sind fundamentale Kenntnisse in der Funktionsweise einer CPU und Kenntnisse in einer Programmiersprache z.B. C++. Im ersten Teil wird allgemeine Theorie vermittelt, wer dabei schon mal was von Assembler gehört hat oder schon selber gebraucht hat, wird keine Schwierigkeiten haben. Im zweiten Teil versuchen wir einige Dinge zu üben. Wir werden mehrere kleine selbst geschriebenes C++ Programm reversen. Dabei steht das Kennenlernen der Tools im Vordergrund. Das Eingehen auf spezifische Reversing-Techniken würde den Umfang dieses Dokumentes sprengen.

Es gibt verschiedene Unterordner. In den Verzeichnissen sind die Quelltexte und EXE's zu den Beispielen zu finden.

2 Was ist Reversing?

Reversing steht für Revers Engeneering (dt. zurück entwickeln). Prinzipiell geht es um die Untersuchung von nicht quelloffenen Software. Als Plattform eignet sich in diesem Fall die Microsoft Windows Reihe hervorragend. Um sich mit der Thematik von Reversing auseinander zu setzen, wird ein fundamentales Verständnis von der Funktionsweise einer CPU vorausgesetzt. Ebenfalls sind Kenntnisse in der hight-level Programmierung von Vorteil. Prinzipiell ist der Prozess des Revers Engineering sehr zeitaufwendig. Ein systematisches Arbeiten auf einer hohen Abstraktionsebene wird verlangt.

2.1 Einsatzgebiet von Reversing?

Es tun die „Bösen“ und die „Guten“. Für Reversing gibt es ein breites Einsatzgebiet. Professionelle Softwarentwickler testen so die Stabilität oder Funktionsweisen von Softwareteilen. Auch Sicherheitsexperten verwenden verschiedene Verfahren zum Aufspüren von Sicherheitslücken in closed-source Software. Die Reversing – Technik kann jedoch auch zum cracken von Software eingesetzt werden.

3 Hight-leven Programmierung

Als hight-level programming oder auf deutsch Hochsprache Programmierung bezeichnet man das Entwickeln von Software auf der Basis von Programmiersprachen, welche für den Menschen vereinfacht wurden. Zu den wohl bekanntesten zählen C und C++. Wobei C als systemnahe Sprache gilt. Hight-leven Sprachen werden mit der Hilfe eines Compilers bzw. Linkers in ausführbare Files übersetzt. Ausnahmen bilden dabei so genannte managed Sprachen, als Beispiel C# und Java. In diesen Sprachen wird der Quelltext in so genannten Bytecode übersetzt, welcher für jede Plattform identisch ist. Im Fall von Java führt die Java Virtual Machine den Bytecode aus. Ein ähnliches Prinzip kommt bei der Microsoft .NET Umgebung zum Zuge.

4 Low-level Programmierung Grundlagen

Entgegen der allgemeinen Meinung handelt es sich beim Maschinencode und dem Assemblercode nicht um zwei unterschiedliche Sprachen. Jeder Assembler Befehl (z.B. MOV Destination, Source) hat seinen einmaligen Operationcode (Opcode). Bei der Ausführung eines Assemblerprogrammes geschieht nichts anderes als das Übersetzen in den Opcode. Gerade bei MOV handelt es sich um einen mehr-byte Befehl. 1. Byte OP-Code, 2. und 3. Byte Destination bzw. Source.

4.1 Register

In der Intel x86 Architektur gibt es acht wichtige interne CPU-Register. Folgende Tabelle soll einen kleinen Überblick verschaffen.

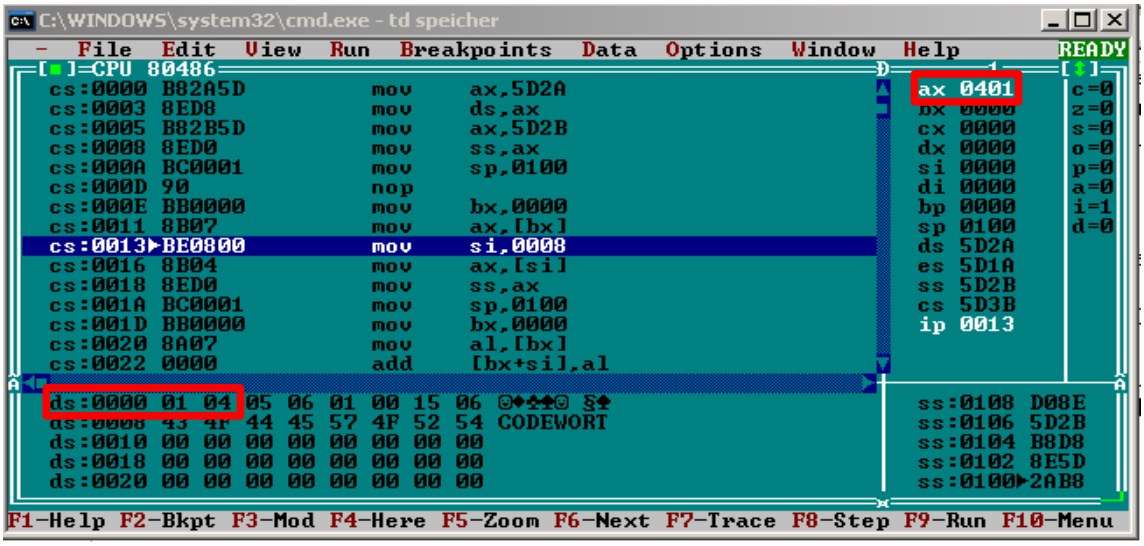
Name	**Breite**	**Beschreibung**
EAX	32 bit	Das wohl wichtigste Register auch Akku genannt.
EBX	32 bit	Lässt sich auch als Pointer einsetzen (EBX + SI)
ECX	32 bit	Wird oftmals als Zähler verwendet z.B. LOOP
EDX	32 bit	Ein gewöhnliches Register
ESI	32 bit	Register welches als Pointer verwendet werden kann, SI = Source Index
EDI	32 bt	Register welches als Pointer verwendet werden kann, DI = Destination Index
EBP	32 bit	Base Poitner
ESP	32 bit	Stack Pointer

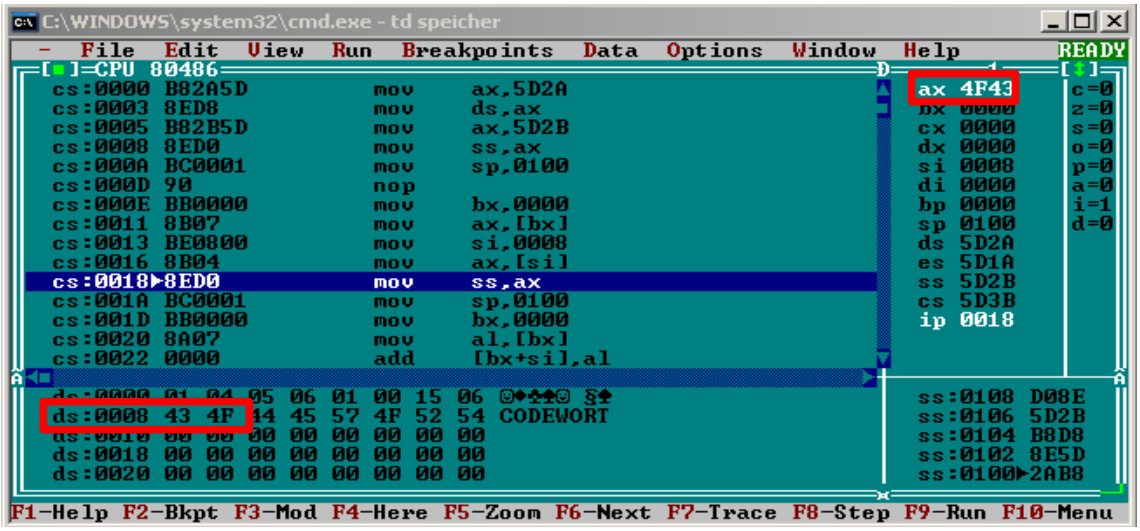
Natürlich gibt es noch weitere Register z.B. DS, SS diese sind an dieser Stelle jedoch nicht relevant. Wer schon mal mit dem Turbo Assembler gearbeitet hat, ist an dieser Stelle eventuell leicht verwirrt. TASM arbeitet mit der 16-Bit Architektur daher heissen die entsprechenden Register dort nur AX, BX, usw.

4.2 Speicher auslesen / schreiben [Beispiel 1]

Programme laden sich bei der Ausführung selber in den Arbeitsspeicher des Rechners. Im Speicher sind alle im Programm definierten Variablen zu finden. Natürlich können wir auch aus einem Assemblerprogramm Daten innerhalb unseres zugänglichen Speichers (je nach Mode der CPU) ablegen. Der Zugriff auf diesen Speicher geschieht mit Hilfe von so genannten Pointern. C++ Programmierer kennen diese unter dem Zeichen „*“. In Assembler sind diese durch „[]“ gekennzeichnet. Prinzipiell lässt sich nicht jedes Register als Pointer verwenden. Wir wollen dies anhand eines kleinen Beispiels untersuchen. Der Einfachheit verwende ich hier den Turbo Assembler mit der entsprechenden 16-Bit Architektur.

[Source-Code Beispiel 1 (Speicher)]
[Tools / Turbo Assembler]





Der IP (Instruction Pointer) hat sich wie auch den beiden Screenshots erkennbar um den Wert 5 vergrössert. Der IP zeigt immer auf die aktuelle Position des auszuführenden Befehls. Handelt es sich beim Befehl um eine mehr-byte Operation z.B. 2, vergrössert sich auch der IP entsprechend.

Bei diesen beiden Beispielen handelt es sich um den einfachsten aller Speicherzugriffe. Um in eine Speicheradresse zu schreiben, werden die MOV's einfach vertauscht. Z.B. MOV [SI], AX. Bitte beachten Sie, dass bei MOV Operationen immer beide Register die selbe Breite (Grösse) aufweisen müssen.

4.3 Variablen [Beispiel 2]

Wie in Hochsprachen kennt auch Assembler-Variablen. Über den Variablen-Namen kann die entsprechende Speicheradresse angesprochen werden.

[Source Code Beispiel 2 (Variablen)]

Ich möchte an dieser Stelle die Theorie verlasen und ebenfalls die Beispiele auf 32-Bit Architektur durchführen. Noch nicht behandelte Befehle werde ich direkt beim ersten auftreten erklären.

4.4 Tools

4.4.1 Debugger

4.4.1.1 OllyDBG

Ein Reverser ist ohne die entsprechenden Tools machtlos. Eines der wichtigsten Werkzeuge ist ein Debugger. Er erlaubt an gewissen Codezeilen Haltepunkte (Breakpoints) zu setzen. In unseren Beispielen verwenden wir OllyDbg.

4.4.1.2 IDA

IDA ist ein weiterer Debugger und Disassembler. Er erlaubt die teilweise grafische Darstellung eines Executables. IDA ist in einer Demoversion gratis erhältlich.

[http://www.datarescue.com/idabase/ida.htm]

4.4.2 Disassembler

Disassembler erlauben uns aus compilierten Executables den Assemblercode herauszulösen. Der Code lässt sich anschliessend ohne weiteres abspeichern oder verändern. Eine der wohl bekanntesten und leistungsfähigsten Disassembler ist W32Dasm.

4.4.3 Filanalyzer

PeiD ist wohl der bekannteste und leistungsfähigste Fileanalyzer. Er erlaubt uns den Entrypoint (Eintrittspunkt) eines EXE-Files zu ermitteln. Teilweise kann er uns auch verraten in welcher Sprache das Executable verfasst wurde.

4.4.4 System Monitoring Tools

Monitoring Tools erlauben die Echtzeit-Überwachung des Betriebssystems. So können z.B. alle Registry-Aktivitäten oder Netzwerkverbindungen aufgezeigt werden.

5 Praxis – Beispiel 3

Einige simple Codezeile in C++ können mehrere hunderte Zeilen Assemblercode zur Folge haben. Der Compiler führt automatisch Performance-Optimierungen durch. Um dies zu erreichen, werden optimale Commands für die Aufgabe des Programmes gesucht. Dies ist jedoch vom Compiler und den Einstellung abhängig. Eine der schwierigeren Aufgaben ist es, der für uns relevanten Teil zu finden. Wir wissen, dass unser Programm eine Aufgabe erfüllt, ob diese sinnvoll ist, bleibt im Moment noch eine offene Frage. Wir lassen das Programm einfach mal laufen, um zu schauen ob dies uns Anhaltspunkte geben kann.

[Beispiel 3/ConAppBeispiel3.exe]

Der Text „Drücken Sie eine beliebige Taste ...“ weisst bereits auf ein SYSTEM Pause Einsatz hin.

An dieser Stelle möchte ich das Vorgehen anhand von IDA und OllyDBG zeigen.

5.1 IDA

Wir laden das EXE-File einfach mal in IDA. IDA zeigt uns nun schon den relevanten Quelltext an.

```
; Attributes: bp-based frame

; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

var_18= dword ptr -18h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 18h        ; char *
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_C], eax
mov     eax, [ebp+var_C]
call    sub_40D080
call    sub_40CCC0
mov     [ebp+var_4], 0AAAAh
mov     eax, [ebp+var_4]
add     eax, 1111h
mov     [ebp+var_8], eax
mov     [esp+18h+var_18], offset aPause ; "PAUSE"
call    system
mov     eax, 0
leave
retn
_main endp
```

Ein Wert wird nach ebp+var_4 geladen, der Wert wird nach EAX geladen

ADD (Addition) EAX, 1111h

Die Pause

Diese Programm ist zwar sehr banal, jedoch konnten wir die Funktion nachvollziehen. Es wird eine

Variabel mit dem Wert AAAAh angelegt. Zu dieser wird 1111h addiert und ebenfalls in eine Variabel abgespeichert.

5.2 OllyDBG

Auch im OllyDBG sind die selben Statements zu finden.

```
00401390  $ 55              PUSH EBP
00401391  . 89E5            MOV EBP,ESP
00401393  . 83EC 18         SUB ESP,18
00401396  . 83E4 F0         AND ESP,FFFFFFF0
00401399  . B8 00000000     MOV EAX,0
0040139E  . 83C0 0F         ADD EAX,0F
004013A1  . 83C0 0F         ADD EAX,0F
004013A4  . C1E8 04         SHR EAX,4
004013A7  . C1E0 04         SHL EAX,4
004013AA  . 8945 F4         MOV DWORD PTR SS:[EBP-C],EAX
004013AD  . 8B45 F4         MOV EAX,DWORD PTR SS:[EBP-C]
004013B0  . E8 CBBC0000     CALL Kopie_vo.0040D080
004013B5  . E8 06B90000     CALL Kopie_vo.0040CCC0
004013BA  . C745 FC AAAA0(  MOV DWORD PTR SS:[EBP-4],0AAAA
004013C1  . 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
004013C4  . 05 11110000     ADD EAX,1111
004013C9  . 8945 F8         MOV DWORD PTR SS:[EBP-8],EAX
004013CC  . C70424 000044(  MOV DWORD PTR SS:[ESP],Kopie_vo.0044000(   ASCII "PAUSE"
004013D3  . E8 98F20000     CALL <JMP.&msvcrt.system>                  system
004013D8  . B8 00000000     MOV EAX,0
004013DD  . C9              LEAVE
```

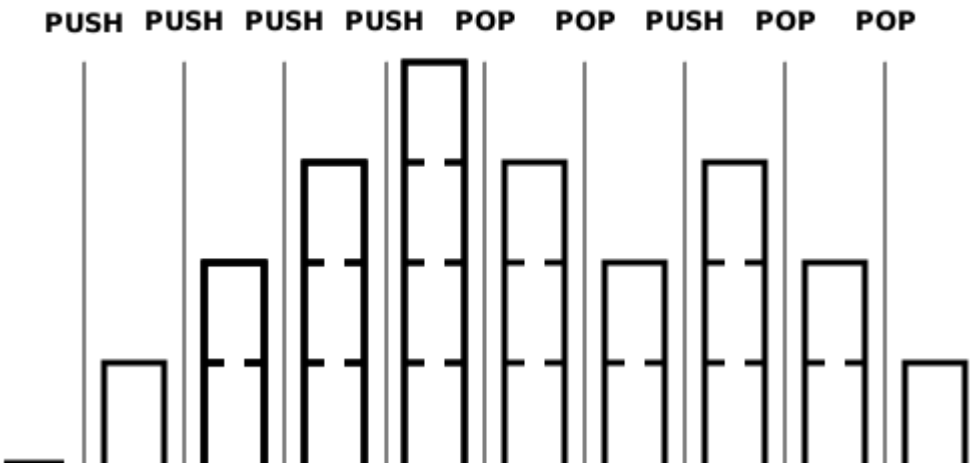
5.3 Auflösung

An dieser Stelle möchte ich Ihnen zum ersten simplen reverse gratulieren. Natürlich bietet dieses Beispiel keine Anwendung in der Praxis. Sie haben nun aber gelernt fundamental mit den beiden Tools zu umzugehen. Der original Quelltext bestätigt unsere Vermutungen.

[Source-Code Beispiel 3]

6 Stack

Der Stack ist ein Speicher welcher den meisten heute gängigen Mikroprozessoren zu Verfügung steht. Im deutschen Sprachgebrauch spricht man häufig auch vom Stapelspeicher oder Kellerspeicher. Der Speicher ist nach dem First In Last Out (FILO) Prinzip organisiert. Sie können dies mit z.B. Umzugskisten vergleichen.



[Bild Quelle: Wikipedia]

Von einem „push“ spricht man, ein Element (Kiste) auf dem Stack abzulegen. „pop“ bedeutet das genaue Gegenteil, der gespeicherte Wert wird vom Stack gelesen und wieder gelöscht. Der Befehl „peek“ holt das Element, entfernt es aber nicht vom Stack. Zu den weiteren Anwendungsgebieten des Stacks werden wir in den folgenden Kapiteln zu sprechen kommen.

7 Unterprozeduren

Sie haben sich sicherlich schon beim Praxisbeispiel 3 gefragt, was wohl der Befehl „call“ bewirkt. Bei „call“ handelt es sich um den Aufruf einer so genannten Unterprozedur. Das Programm springt an die entsprechende Adresse und führt die Befehle bis zum Command „retn“ aus. Der Stack spielt dabei eine wichtige Rolle, die Rücksprungadresse wird bei einem „call“ auf dem Stack abgelegt. So müssen die PUSH und POP Operationen innerhalb einer Prozedur ausgeglichen sein. Ein nicht beachten dieser Regel hätte beim return einen Sprung an eine falsche Adresse zur Folge. Das Programm würde nicht mehr ordnungsgemäss weiter ausgeführt werden.

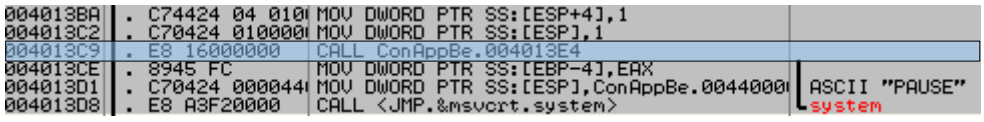
8 Praxis - Beispiel 4

Wir möchten nun anhand eines Beispiels die Funktionsweise und Möglichkeiten von Stacks und Unterprozeduren kennen lernen. Dazu gibt es wieder ein Beispiel.

[Beispiel4/ConAppBeispiel4.exe]

Zuerst lassen wir das Programm einfach mal rennen. Bitte beachten Sie, wenn Sie sich über die Herkunft und Funktion nicht sicher sind, sollten Sie dies nicht tun. Auf Ihrem PC könnte Schadcode ausgeführt werden.

Wir laden das EXE-File wieder in OllyDBG und scrollen bis zur Pause. In den paar Codezeilen oberhalb finden wir einige wichtige Hinweise.



In der ersten Zeile wird der Wert 1 auf das Stack Segment mit der Adresse Stack Pointer + 4 geschrieben. Das Selbe geschieht in der zweiten Zeile an die Adresse des SP's. Nun folgt ein Call, wir verfolgen mal diesen Aufruf indem wir Rechtsklick -> Follow auswählen.

PUSH EBP ; Wert wird auf Stack gepusht

MOV EBP,ESP ; Stack Pointer wird nach Base-Pointer kopiert

MOV EAX,DWORD PTR SS:[EBP+C] ; Wert wird vom Stack-Segment abgeholt

ADD EAX,DWORD PTR SS:[EBP+8] ; zwei Werte werden addiert

POP EBP ; POP und PUSH ausgleichen

RETN ; Rücksprung

Anhand dieser wenigen Zeilen erfahren wir viel über das Programm.

- Es handelt sich um eine Funktion bzw. Methode

- Es werden zwei Parameter übergeben
- Die beiden Wert werden Addiert
- Es geschieht wieder ein Rücksprung an die ursprüngliche Position
- Der Rückgabewert (via Register EAX) wird abgespeichert.

Ein C/C++ Programmierer könnte aus diesen Erkenntnissen schnell folgendes Programm evaluieren.

[Source Code ConAppBeispiel4.exe]

Wir haben nun eine zusätzliche Funktion des Stack-Segments bzw. Speichers kennen gelernt. Parameter können in dieser Form übergeben werden, indem ein Wert im Speicher abgelegt wird. Dies kann durchaus auch über den Stack direkt geschehen (push, pop). Beachten Sie bitte, dass beim verwenden des Schlüsselwortes „inline“ in C++ kein Call-Aufruf erfolgen muss.

9 Schlusswort

Ich hoffe Sie haben ein wenig Einsicht in das grosse Gebiet des „Reverse Engineering“ erhalten. Ich würde mich um ein Feedback sehr freuen. Ich bin ebenfalls für Verbesserungsvorschläge und Korrekturen offen.

mailto:k.burger@burgernet.org

Weblog: http://weblog.burgernet.org

Homepage: http://www.analyzernet.net.ms