

DEEP Coding

with Autohotkey Script

PROGRAMMABLAUF

DATENTYPEN

POINTER & ADRESSEN

WINDOWS API STRUCT

von Pascal Büttiker (aka IsNull) 2008

INFORMATIONEN

PRERELEASE:MOD-DATE: 02.11.08

VER: 1.1

LICENSE: GNU GPL

DANKSAGUNGEN

Danke an Denick, der schon mal Probe gelesen, und einige Bugs aufdecken konnte.

Danke an SKAN, der mir beim Verstehen der STRUCT-Objekte geholfen hat.

Danke an derRaphael, der mir ebenfalls beim Verstehen der STRUCT-Objekte mit einem Mini-Tut geholfen hat.

Inhalt

1	Vorwort	4
2	Programmablauf	5
2.1	PE FORMAT	5
2.2	Memory	5
2.3	PE Ausführen	6
3	Pointer	7
3.1	AHK und Pointer	7
3.2	Nutzen der Pointer	7
3.3	NumPut & NumGet	8
3.3.1	Grundsätzliches	8
3.3.2	Im Memory jonglieren	9
3.3.3	NULL terminierte Strings aus dem Memory fischen	9
4	Datentypen	10
5	WIN API – C - STRUCT	11
5.1	Voraussetzungen	11
5.2	Einführung	11
5.3	STRUCT	11
5.3.1	STRUCT Member Typen	12
5.3.2	STRUCT erstellen	13
5.3.3	STRUCT lesen	13
5.3.4	STRUCT in AHK	13

1 Vorwort

Dieses Tutorial habe ich zur Selbstvertiefung geschrieben. Es stellt einen Gegensatz zur Grundidee von AHK dar; Hier möchte ich etwas tiefere Einblicke in die Welt der Programme geben.

Der STRUCT Teil ist für mich selber Neuland.

Im engl. AHK Forum wurde mir der Grundsätzliche Aufbau von SKAN erklärt. Auch derRaphael aus dem deutschen AHK Forum hat ein gutes Mini-Tut erstellt.

Nun möchte ich dieses Thema möglichst klar auf Deutsch dokumentieren, bez. erklären.

Ich hoffe es ist einigermaßen verständlich, und enthält keine Grundlegenden Fehler.

2 Programmablauf

Um in diese Programmier-Tiefen hinabzusteigen, muss man wissen, wie das ganze hinter den Kulissen abläuft. Das hier vermittelte Wissen dringt bis an Assembler / Maschinensprache heran. Es hilft dem Verständnis für Programmieren ungemein.

Des Weiteren ist dies ein Bereich, welcher von Code Reversern, Crackern und auch Virenautoren (den guten, nicht Skriptkiddys) gut beherrscht wird, werden muss.

Es ist natürlich auch klar, dass dieses komplexe Thema hier nur gestreift wird, wer mehr wissen will, bekommt aber einige Links.

2.1 PE FORMAT

Eine EXE oder eine DLL liegt im PE (Portable Executable) Format vor. Das PE Format Beschreibt das Programm und enthält natürlich die Daten und den Code. Mit „beschreiben“ ist gemeint, dass dort definiert ist, wie viel Platz die EXE im Memory verbraucht, welche Funktionen importiert werden, und welche exportiert werden. Nur auf exportierte Funktionen kann man zugreifen. DLLs nutzen also vor allem das exportieren, um ihre Funktionen anderen Programmen zugänglich zu machen. Es gibt sehr viele Informationen, die in PE Dateien existieren – ein paar wenige werden im späteren Verlauf noch angedeutet.

Der Code liegt in der EXE/DLL übrigens in der „Maschinentersprache“ vor. Diese für Menschen unlesbare Sprache kann in die (immer noch schwer) lesbare Sprache Assembler 1:1 übersetzt werden. Dieser Vorgang nennt man disassembeln und wird beim Code Reversing verwendet. Code Reversing bezeichnet die Tätigkeit, kompilierte Programme zu analysieren und ihre Prozeduren nachzuvollziehen.

Wer mehr über das PE Format wissen möchte, kann sich folgende Links ansehen:

http://de.wikipedia.org/wiki/Portable_Executable

2.2 Memory

Das Ram, der Arbeitsspeicher, oder eben das Memory dient als schneller Zwischenspeicher für Programme und verwendeten Daten. Der Prozessor kann sehr schnell mit dem Memory kommunizieren. Hier spielt sich das für uns interessante ab.

Man kann sagen, dass „Daten“ und „Code“ im Memory gleich sind. Beide haben gemeinsam, dass sie jeweils über eine eigene Adresse im Memory verfügen. So kann sich der Prozessor bestimmte Werte holen, in dem er dem Memory-Controller sagt, er möchte gerne den Inhalt von Adresse XY haben. Genau so verhält es sich auch mit dem Code. Auch dieser hat für jeden Befehl eine Adresse.

2.3 PE Ausführen

Ein Programm (EXE/DLL Datei) wird ausgeführt, in dem der enthaltene Code und Daten Sektion vom Loader ins Memory kopiert wird. Danach wird Das Programm ausgeführt:

Der Prozessor fängt irgendwo* an, die EXE auszuführen, und klappert dann von oben nach unten jede Adresse ab, und führt den Code aus. **irgendwo: natürlich ist klar definiert wo: Der EP (Entry Point) enthält die Einsprungadresse, wo der Prozessor mit dem ausführen beginnen soll. Der Entrypoint wird ebenfalls im PE Format „beschrieben“ und wird vom Loader ausgelesen.*

Nun, es gibt natürlich Assembler Befehle, welche dieses sture abarbeiten von oben nach unten unterbrechen können: Zum einen Gibt es Prozedur aufrufe, „CALL ADRESSEXY“ dann springt das Programm zur ADRESSEXY im Memory und fährt dort mit der Ausführung fort. Ebenfalls gibt es Jumps, welche zu anderen Adressen springen.

Die klassischen Programmabstürze folgen ebenfalls aus dieser Tatsache, dass ein Sprung zu einer Adresse erfolgt, die nicht existiert.

Wer mehr dazu wissen möchte, muss sich in Assembler etwas einlesen.

3 Pointer

Wer das Kapitel Programmablauf einigermaßen verstanden hat, wird nun Pointer gut verstehen. Pointer sind „Zeiger“, die auf eine Adresse im Memory zeigen. Es gibt in einigen Programmiersprachen (z.B. C/C++) die Möglichkeit, mit Pointern zu arbeiten. Pointer sind allerdings auch gefährlich.

Mit Pointern kann man direkt auf Werte im Memory zugreifen. -> man muss einfach die Adresse im Memory kennen.

3.1 AHK und Pointer

Da AHK von einem C++ Interpreter ausgeführt wird, existiert auch die Unterstützung für Pointer.

```
Var := "Ich bin ein String"  
pVar := &Var
```

Nun, mit dem ersten Befehl weisen wir der Variablen "Var" einen String zu. Dieser String ist nun irgendwo im Memory. Wo?

Dies finden wir heraus, in dem wir von der Variablen `Var` die Adresse im Memory verlangen: Einfach ein "&" vor die Variable stellen, und als Rückgabe Wert erhalten wir unseren Pointer, der nun in einer zweiten Variablen "`pVar`" steht.

Wir können natürlich auch wieder auf den String zugreifen, wenn wir den Pointer (die Adresse im Memory) von ihm haben:

```
Var := *pVar
```

Ein Stern (*) vor einer Variablen veranlasst AHK, den Inhalt der Variablen als Pointer anzuschauen (was es in unserem Falle ja ist) und holt sich den Wert von dort das erste Byte im Memory. Wie man den ganzen String von einer Adresse zurückholt, wird im `NUMGET()` Kapitel erklärt.

Ist der Pointer (wäre hier `pVar`) fehlerhaft und würde ins Nirwana zeigen, knallt das Programm. Ein Programmabsturz ist die Folge.

3.2 Nutzen der Pointer

Sehr oft verlangen Windows API Funktionen einen Pointer zu den Werten als gleich den Wert selber.

Man kann sich Pointer wie ByRef Variablen vorstellen – im Memory muss keine Kopie der Daten angelegt werden, sondern es können die bestehenden Daten verwendet werden.

Auch wenn man mit binären Dateien arbeitet, sind Pointer nötig. Ein String im Memory ist durch ein 0x00 Byte begrenzt. Liest man Daten ein, die unter anderem 0x00 Bytes enthalten, verliert man alle Daten, die nach dem 0x00 vorkommen.

Wenn man aber mit Pointer arbeitet, kann man das Programm zwingen, die gewünschten Werte von Adresse XY aus dem Memory zu holen und muss sich nicht mit der automatischen Stringsuche abgeben.

Dazu gibt es die AHK Funktionen `NumGet()` und `NumPut()` mit denen kann man von beliebigen Adressen eine beliebige Anzahl Bytes holen, oder hinschreiben.

3.3 NumPut & NumGet

3.3.1 Grundsätzliches

Diese Befehle benötigt man immer dann, wenn man an einer Bestimmten Memory-Adresse (evtl. mit einem Offset) Daten lesen oder schreiben muss.

Ein ganz simples Beispiel:

Eine Datei `test.bin` hat folgenden Inhalt: (hex Werte:)

```
41 42 00 41
```

Das sind genau 4 Bytes. Der Höchste Wert, für ein Byte ist 0xFF, das entspricht Dezimal 255. So viele unterschiedliche Zeichen können maximal mit einem Byte dargestellt werden.

(schreibt man normalerweise mit dem 0x Prefix)

```
0x41 0x42 0x00 0x41
```

Nun, "0x41" bedeutet laut Ascii Tabelle "A", 0x42 bedeutet "B" und dann folgt ein Null Byte: 0x00 und das Letzte Zeichen ist wieder ein "A"

Öffnet man diese Datei in Notepad, sieht man ca. folgendes:

```
AB[]A
```

Das [] steht für ein nicht darstellbares Zeichen. Das NULL-Byte.

Liest man diese Datei in AHK ein, ...

```
fileread, dump, test.bin
```

...sind diese 4 Bytes im Memory: In der Variablen "dump". Doch lässt man AHK die String-Länge ausgeben:

```
Strlen(dump)
```

...merkt man, dass da was nicht stimmt. Der Rückgabewert wird 2 sein. Nicht 5.

Schon am Rückgabewert von `strlen()`, welcher die Länge der Variable `dump` ermittelt wird die Problematik klar. **Ein String ist beim NULL-BYTE beendet.**

```
Msgbox %dump%
```

Auch eine **msgbox** würde nur „AB“ ausgeben. Das NULL-Byte zeigt das Stringende an, und somit wird 0x00 und das letzte A nicht mehr als String interpretiert.

Wenn wir auf das 5te Zeichen/Byte zugreifen wollen (Unser 2tes „A“), müssen wir `NumGet()` verwenden:

```
fileread, dump, test.bin           ;Datei in Variable dump einlesen
mybyte := NumGet(dump, 5, "Uchar") ;1 Byte (1 = „uchar“) an Stelle 5 auslesen.
```

Liest aus der Variablen `dump` ein Byte an stelle 5. (offset 5) aus.

In `mybyte` ist nun `0x64` enthalten, um das `A` zu bekommen, wenden wir noch `chr(mybyte)` an. `Chr()` schaut in die ASCII Tabelle, und holt den Buchstaben für die zugehörige Nummer/Byte.

```
fileread, dump, test.bin
mybyte := NumGet(dump, 5, "UChar")
mychar := chr(mybyte)
```

3.3.2 Im Memory jonglieren

Hier ein einfaches Beispiel. Mit `VarSetCapacity()` wird einer Variablen 1 Byte Speicher zugewiesen und dieser mit `NULL (0x00)` initialisiert. Danach wird mit `NumPut(...)` der hex Wert von `A`, sprich `0x41` in diese Speicheradresse geschrieben. Die letzte Zeile liest nun von der Adresse das 1 Byte, und gibt somit wieder unser `0x41` zurück.

```
SetFormat, integer, hex ;rechnet mit Hexwerten

VarSetCapacity(Var, 1, 0) ;weist der Variablen Var 1 Byte speicher zu, [0x00]
NumPut(0x41, Var, 0, "UChar") ;schreibt 0x41 [A = chr(0x41)] in das 1 Byte.
msgbox % NumGet(&Var, 0, "UChar") ;Liest das 1 Byte aus. (erhält 0x41)
```

3.3.3 NULL terminierte Strings aus dem Memory fischen

Nehmen wir an, wir haben eine Variable `"var"`, in welcher ein String steht. Wir kennen aber nur die Adresse dieser Variable [Pointer zum 1 Byte des Strings] (und somit die genaue Adresse des ersten Bytes von diesem String), muss man wie folgt vorgehen, um den String aus dem Memory zu extrahieren:

```
var := "Test String" ; Weist der Variablen den String zu.
pvar := &Var ; Schreibt die Adresse von var in pvar
msgbox % "Adresse im Memory ist: " . pvar
```

Wir haben also lediglich `pvar` bez. `&Var` gegeben -> den Pointer zum Stringanfang. Um nun den String aus dem Memory zu holen, machen wir uns zunutze, dass ein String mit dem `NULL-Byte (0x00)` beendet wird. Wir holen nun jedes Zeichen, bis der Inhalt davon `0x00` ist. Dort muss der String fertig sein.

```
Loop ; Initialisiert einen Unendlich Loop
{
    SetFormat, integer, hex ; Alle Zahlen als Hex betrachten
    x := NumGet(&Var, a_index - 1, "UChar") ; holt immer das nächste Byte
    If (0x00 = x) { ; Wenn das geholte Byte ein NULL-Byte ist,
        Break ; dann beende Loop. Der String ist hier fertig.
    } else {
        str .= chr(x) ; ansonsten Wandeln wir das Byte in ASCII um
    }
} ; Loop-Schleife
msgbox % str
```

So erhalten wir unseren Ursprungs-String zurück. Genau so geht auch AHK intern vor (Eigentlich C++), wenn man einen String ausliest. Genau deshalb kann man auf solchen Strings keine „normalen“ String-Operationen ausführen, da diese alles hinter dem ersten `NULL` Byte ignorieren.

4 Datentypen

int64 = 8 byte
int, uint = 4 byte
short, ushort = 2 byte
char, uchar = 1 byte

UShort und Short

Besteht aus 2 Bytes:

Hex: 00 00 bis FF FF

Pseudo Dezimal: 0 0 bis 255 255

Beide Repräsentieren einen Wert von 0x0000 bis 0xffff, der jedoch unterschiedlich interpretiert werden kann. Short kennt negative Zahlen, die mit einem HighBit dargestellt werden

Positive Zahlen wären demnach:

0x0000 bis 0x7fff

Negative Zahlen

0x8000 bis 0xffff

Das lässt sich auch auf andere Typen transferieren:

Unsigned nimmt die volle Bitmöglichkeit als Zahlenraum auf, **signed** hingegen die halbe. Unsigned kennt nur positive Zahlen, signed auch negative:

Char und UChar

UChar:

0x00 bis 0xFF

0 bis 255

Char

0x80-0x7f

-128 bis +128

5 WIN API – C - STRUCT

5.1 Voraussetzungen

Was man schon wissen muss:

- >> Funktionen, Parameter und wie sie in AHK Funktionieren
- >> DLLCALL von „normalen“ DLL-Funktionen
- >> Was sind Pointer/Adressen im Memory?
- >> Wie verwende ich sie in AHK?
- >> DWORD, UInt SHORT usw. sollten ungefähr bekannt sein.

5.2 Einführung

Wer oft DLLCALLS zu Windows-API Funktionen tätigt, wird früher oder später über die sogenannten „STRUCT“ Parameter stolpern. Hier ein Beispiel einer solchen Funktion:

```
HANDLE AddPrinter(
    LPCTSTR *pName,    // server name
    DWORD Level,      // printer information level
    LPBYTE pPrinter   // printer information buffer
);
```

1. Parameter (LPSTR *pName) ein Pointer zu einem String.

Der erste Parameter ist ein „String“. Naja, nicht ganz. Es wird nur ein Pointer zu einer Stelle im Memory erwartet, wo dieser String zu finden ist. Und ein Pointer ist nichts mehr als eine Adresse und somit eine positive Zahl. -> Daher wird es ein UInt-Wert sein, der die Adresse zu unserem String enthält.

2. Parameter (DWORD) Eine Zahl

Der Zweite Parameter ist ein „DWORD“ also eine Zahl. Hier muss man keinen Pointer verwenden, man kann direkt die Zahl übergeben, ebenfalls als UInt.

3. Parameter – STRUCT

So, da ist er nun, der STRUCT Parameter. Hier wird ein Pointer zu einem STRUCT erwartet. Was das nun genau ist, soll hier geklärt werden.

5.3 STRUCT

Ein STRUCT Parameter ist ein spezielles ARRAY-Element. Im Grunde ein Bytehaufen, wo diverse Informationen einfach aneinander gereiht werden. Also könnte man diese STRUCT als einen Haufen Parameterwerte bezeichnen. Diese STRUCT's werden normalerweise Dokumentiert und es liegt meist eine „Definition“ bei: Ein vereinfachtes Beispiel:

Eine Definition des STRUCT „Farben“

```
STRUCT Farben(
    LPCTSTR Farbe
    LPCTSTR Farbe2
);
```

Farbe soll den String „grün“ haben, Farbe2 „blau“.

Hier könnte der STRUCT so aussehen „grünblau“. Nun hat man aber das Problem, dass man nicht weiß wo der eine Parameter anfängt, und wo er aufhört. Daher ist bei jedem der Einträge (Member) angegeben, welchen Typ der Parameter hat.

5.3.1 STRUCT Member Typen

Deswegen arbeitet man mit Werten, wo man die Grösse in etwa festlegen kann. Bei Strings ist das schwierig, daher verwendet man deshalb „Pointer“ die auf den eigentlichen String zeigen. Die kleinen „p“ vor dem Eintrag deuten schon mal auf Pointer hin:

```
STRUCT Farben(  
    LPTSTR pFarbe  
    LPTSTR pFarbe2  
);
```

In dem Beispiel werden zwei Pointer erwartet. Nun, die Pointer werden als **UInt** Zahlen erwartet. Und genau hier liegt der Grundgedanke dieses STRUCT-ARRAY:

Die Werte können von einander getrennt bez. unterschieden werden, da man die Grösse weiss, welche sie im Memory einnehmen. Die Reihenfolge bestimmt die STRUCT Definition.

Eine Liste mit den Bytegrössen:

```
int64          = 8 byte  
int, uint      = 4 byte  
short, ushort = 2 byte  
char, uchar   = 1 byte
```

Unser STRUCT hat zwei LPTSTR Werte (LPTSTR entspricht normalen AHK Strings), wo er jeweils einen Pointer dafür erwartet. D.h. unser STRUCT enthält zweimal ein UInt Wert, und wird deshalb $2 * 4 = 8$ Bytes gross sein.

Ein STRUCT ist also ein Bytehaufen, bei welchem die einzelnen Elemente eine feste Länge aufweisen. Die Elemente müssen genau in der richtigen Reihenfolge sein, und genau die richtige Grösse aufweisen.

5.3.2 STRUCT erstellen

Das Grundsätzliche Vorgehen ist es, zuerst die STRUCT Variable mit der gewünschten Grösse zu erstellen (hier sind es 8 Bytes), und nachher die Werte (immer UInts!?) dort einzufüllen.

Leeren STRUCT:

Position	Byte(0x)	Kommentar
1	00	Hier beginnt UInt 1
2	00	
3	00	
4	00	
5	00	Hier beginnt UInt 2
6	00	
7	00	
8	00	

Mit `varsetcapacity(STRUCT, 8, 0)` erstellt man diese leere Struktur, und füllt sie mit binären Nullen. (0x00).

Nun muss man die einzelnen UInt's mittels **NumPut(...)** in diese Struktur einfügen. Mit dem richtigen Offset versteht sich. Dabei muss das erste UInt an Stelle 1 sein, und der zweite UInt an Stelle 5 beginnen.

```
NumPut( pFarbe,      USERDATA, 0, "UInt" ) Einfüge Pointer @ offset 0
NumPut( pFarbe2,    USERDATA, 4, "UInt" ) Einfüge Pointer @ offset 4
```

Achtung, Position 1 ist der Pointer bei 0, und bei 5 bei 4.

5.3.3 STRUCT lesen

Um einen STRUCT zu lesen, muss man genau umgekehrt vorgehen wie beim erstellen. Man muss wissen, in welcher Reihenfolge die Werte abgelegt wurden und welchen Platz die einzelnen Werte einnehmen. (Datentyp)

5.3.4 STRUCT in AHK

Da AHK keine Native Unterstützung für STRUCT-Objekte bietet muss man diese umständliche, Low-Level Lösung verwenden.

„**corrupt**“ aus dem englischen AHK-Forum hat aber eine Funktionsbibliothek erstellt, die den Umgang mit STRUCT-Objekten erleichtern. Zu finden ist sie in diesem Topic:

<http://www.autohotkey.com/forum/topic4888.html>