1 Buffer-Overflow

1.1 Literatur

http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/sicher2/website/
http://en.wikipedia.org/wiki/X86_instruction_listings
http://msdn.microsoft.com/en-us/library/1b80826t.aspx
http://en.wikipedia.org/wiki/X86#x86_registers
http://www.intel.com/products/processor/manuals/

1.2 Das Prinzip

Es gibt fast kein Programm, das keinen Fehler besitzt. Gewiefte Hacker können diese Programme angreifen (siehe C'T 2001 Heft 23).
Durch die Analyse eines Programms (z.B. mit dem Debugger), kann ein Hacker z.B. in Erfahrung bringen, daß in diesem Programm nicht genügend Speicher für die Eingabe einer Zeichenkette reserviert wurde und deshalb nichtreservierter Speicher überschrieben wird (Buffer-Overflow). Wenn dadurch die Rücksprungadresse einer Funktion in das Hauptprogramm überschrieben wird, "stürzt" das Programm normalerweise ab.
Der Hacker kann nun eine bestimmte Zeichenkette - einen sogenannten **Exploit** - eingeben, die aus folgenden Teilen besteht:
1) einem sogenannten **Payload** (Virus), das sind Maschinenbefehle des Hackers, die dann später ausgeführt werden.
2) der Adresse des Payload. Diese Adresse muß an der Stelle im Arbeitsspeicher stehen, wo normalerweise die Rücksprungadresse der Funktion in das Hauptprogramm steht.
Kommt das anzugreifende Programm an diese Stelle wird dann nicht in das Hauptprogramm zurückgesprungen, sondern in den Payload verzweigt und dieser ausgeführt.

Um den Hintergrund genau zu verstehen, ist es von Vorteil, jeweils die hier besprochenen Demo-Programme mit dem Debugger auf Assemblerebene (bzw. Maschinencode) zu analysieren.
Hier wurde MS VC++ 6.0 benutzt. Dies kann aber genauso mit einer anderen, passenden Entwicklungsumgebung geschehen.

1.3 Grundlagen und Voraussetzungen

1.3.1 Assembler und höhere Programmiersprache C (C++)

Der Compiler übersetzt ein in C bzw. C++ geschriebenes Unterprogramm nach einem gewissen Schema in die Assemblersprache.
Dies soll an dem folgenden, einfachen Beispiel demonstriert werden.

1.3.1.1 Sourcecode in C

```
void f(char form1, char form2){
  char lok1;
  char lok2;
  lok1 = form1;
  lok2 = form2;
}

void main(){
  f('A', 'B');
}
```

1.3.1.2 In "symbolischen" Assembler umgesetzt

1.3.1.2.1 Bemerkungen

1) Hier wird von einem "symbolischen" Assembler gesprochen, weil Befehle wie z.B.
`mov  [ebp-4][ebp+8]`
in dem zwei Adressen als Operanden vorkommen, im Befehlssatz des 80386 von Intel nicht vorkommen (im Befehlssatz des 80386 gibt es nur Befehle mit maximal einer Adresse im Operanden).
Die Schreibweise mit 2 Operanden in einem Befehl ist kürzer, deswegen wurde sie hier benutzt.

2) Obwohl lok1 und lok2 vom Datentyp char und damit 1 Byte groß sind, reserviert der Compiler für sie jeweils 4 Byte.

1.3.1.2.2 Das Programm

```
// Die ersten 3 Befehle sind der Funktionsaufruf mit den 2 Parameteren
push form2      // gepushter 2. Parameter der Funktion
push form1      // gepushter 1. Parameter der Funktion
call f          // pusht die Adresse des call f folgenden Befehls mov ax, ax auf den
                // Stack
// Irgendein Befehl, der dem Aufruf im Hauptprogramm folgt. Im obigen C-Programm gibt es
// diesen Befehl nicht, aber intern muß es einen Befehl geben, der veranlaßt, daß nach main
// wieder ins Betriebssystem "gesprungen" wird (Betriebssystem wieder Kontrolle bekommt).
mov ax, ax      // Irgendein Befehl
...
...
// Beginn der Funktion f
f:
// sichert den (alten) Wert von ebp, weil ebp im push ebp folgenden Befehl überschrieben wird
push ebp
// hält den Wert des Stackpointers in ebp fest. Über ebp wird dann auf die formalen Parameter
// und lokalen Variablen der Funktion zugegriffen. Dies ist schlecht über den Stackpointer esp
// möglich, da sich dieser (z.B.bei einem push-Befehl) wahrscheinlich oft ändert, während ebp
// innerhalb der Funktion konstant gehalten wird.
// Nach dem Verlassen der Funktion muß der Platz, der für den Stack geschaffen wurde,
// wieder freigegeben werden. Das geschieht weiter unten, indem der Stackpointer dann den
// Wert von ebp bekommt!
mov ebp,esp
// für lok1 und lok2 wird auf dem Stack Platz geschaffen. Da lok1 und lok2 insgesamt 8 Byte
// benötigen, muß der Stackpointer (um mindestens) 8 vermindert werden (Stack wächst in
// Richtung der kleineren Adressen). Ein evtl. weiterer push-Befehl in der Funktion veranlaßt
// dann, daß auf dem Stack nach diesen 8 Byte etwas abgelegt wird!
// Bem: push vermindert ebenfalls den Stackpointer
sub esp,8
// kopiert den formalen Parameter form1 in die lokale Variable lok1
mov [ebp-4][ebp+8]
// kopiert den formalen Parameter form2 in die lokale Variable lok2
mov [ebp-8][ebp+12]
// der Wert des Stackpointers nach dem Aufruf der Funktion wird wieder hergestellt.
mov esp, ebp
// der Wert von ebp nach dem Aufruf der Funktion wird wieder hergestellt.
pop ebp
// ret holt die Rücksprungadresse vom Stack und kopiert sie in den Befehlszähler eip
// und macht dadurch direkt nach call f (also bei mov, ax, ax) weiter.
ret
```

1.3.1.3 In Assembler und Maschinesprache des 80386 umgesetzt

Der Compiler und der Linker machen aus dem obigen C-Programm folgendes Programm:

Adresse Maschinencode Assemblerbefehle

```
00401020 55                   push ebp
00401021 8B EC                mov ebp,esp
// eigentlich würden 8 Byte reichen. Der Compiler reserviert aber mehr.
00401023 83 EC 48             sub esp,48h
// lok1 = form1
00401038 8A 45 08             mov al,byte ptr [ebp+8]
0040103B 88 45 FC             mov byte ptr [ebp-4],al
// lok2 = form2
0040103E 8A 4D 0C             mov cl,byte ptr [ebp+0Ch]
00401041 88 4D F8             mov byte ptr [ebp-8],cl
00401047 8B E5                mov esp,ebp
00401049 5D                   pop ebp
0040104A C3                   ret
...
00401078 6A 42                push 42h // Zeichen B  <---
0040107A 6A 41                push 41h // Zeichen A
0040107C E8 9F FF FF FF       call 00401020
00401081 83 C4 08             add esp,8
```

Bemerkung: Programm beginnt an der mit <--- bezeichneten Stelle.

1.3.1.3.1 Frage zu call

Warum springt der call-Befehl [E8 9F FF FF FF] zur Adresse [00401020] ?

Antwort:
Zum Befehlszähler eip = 0040107C wird die Befehlslänge 5 und dann die Distanz hinzuaddiert. Aber wie groß ist die Distanz ? Dazu schaut man sich den Maschinencode des Befehls an:

E8	9F	FF	FF	FF

Die Bedeutung der einzelnen Bytes ist:

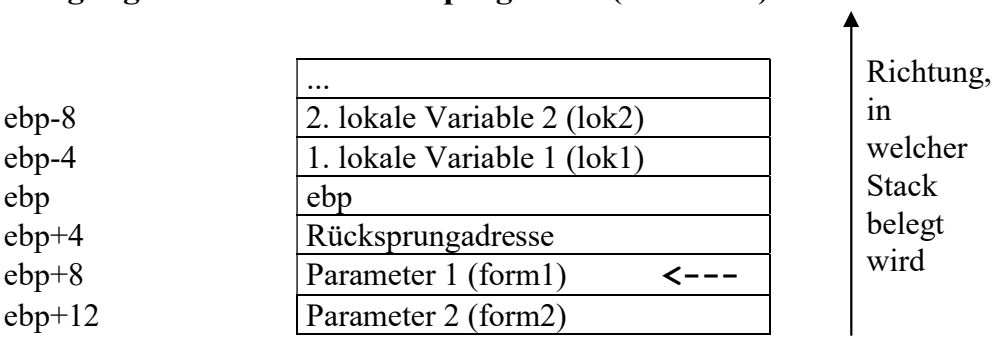
Op-Code	adress-low		adress-high	
	low-Byte	high-Byte	low-Byte	high-Byte

Damit hat man dann die folgende Distanz: FF FF FF 9F
Damit bekommt man dann:

```
  0040107C
+        5
+ FFFFFF9F
----------
  00401020
```

Bemerkung:
Das mit MS VC++ 6.0 erzeugte Programm macht (umständlicherweise ?) einen call und dann auch noch einen jmp. Deswegen stimmt es mit obigem Programm (am Anfang) nicht genau überein

Belegung des Stacks im Unterprogramm (Funktion)



Allgemein:
Beim Aufruf eines Unterprogramms wird auf dem Stack ein sogenannter Frame "eingerichtet", in dem die formalen Parameter, die lokalen Variablen und die Rücksprungadresse gespeichert werden.

Bemerkung:
Der Stack wächst in Richtung der niederen Adressen.

Konkret:
C++ pusht die Parameter von rechts nach links auf den Stack, d.h. zuerst wird dorm2 (der Buchstabe B) gepusht, dann form1 (der Buchstabe A).
Danach wird die Rücksprungadresse des nächsten Befehls, der dem Aufruf der Funktion (im Hauptprogramm main) folgt (der Befehl nach call f ist mov ax, ax) auf den Stack gepusht.
Der letzte Befehl ret in der Funktion f holt dann diese Rücksprungadresse vom Stack und macht dadurch direkt nach call f (also bei mov, ax, ax) weiter.
Der erste Befehl in dem durch call f aufgerufenen Unterprogramm (entspricht der Funktion f) sichert (pusht) den Inhalt des Registers ebp auf dem Stack, der zweite kopiert esp nach ebp (Framepointer).
In ebp steht nun der aktuelle Wert des Stackpointers esp (esp kann sich in der Funktion zwar noch z.B. durch weiter push-Befehle) ändern, aber ebp ist dann immer noch gleich.
Deswegen kann man mit ebp auf die formalen Parameter und lokalen Variablen innerhalb der Funktion problemlos zugreifen.
Mit "ebp + negative Werte" wird auf die lokalen Variablen, mit "ebp + positive Werte" wird auf die formalen Parameter zugegriffen.

Bemerkung:
Der Pfeil oben gibt die Stelle an, auf die der Stackpointer zeigt (Wert von esp), nach der Ausführung der Funktion f (nach Ausführung des Befehl ret in f), d.h. wenn der erste Befehl nach dem Rücksprung in das Hauptprogramm main gemacht wird, zeigt esp auf diese Stelle.

1.4 Buffer-Overflow: Ein konkretes Beispiel:

1.4.1 Das anzugreifende Programm

```
#include "stdafx.h"
#include <string.h>
#include <stdio.h>

void f();

void f(){
  char lok1[8];
  char lok2[18];
  printf("Bitte Zeichen einlesen\n");
  scanf("%s",&lok2);
  strcpy(lok1, lok2);
}

void main(){
  f();
}
```

1.4.2 Der Angriff

Dieses obige Programm hat einen Programmierfehler:
Wenn der Anwender z.B. 15 Zeichen über Tastatur eingibt, werden diese in das Feld lok1 kopiert (durch strcpy), obwohl dort nur Speicher für 8 Bytes reserviert wurden. Es wird also nicht reservierter Speicher überschrieben. Der Trick, dieses Programm anzugreifen, besteht nun darin, in diesen reservierten und nichtreservierten Speicher "geeignete" Bytes, d.h. ein **Programm** einzugeben.

Konkret:
Dieses obige Programm wird dadurch angegriffen, daß dort über Tastatur (bei der Eingabeaufforderung dieses Programms) die ASCII-Zeichen folgender ASCII-Werte eingegeben werden:
235 254 ? ? ? ? ? ? # # # # 36 255 18 0

Bemerkungen:
1) ? bzw. # steht für einen beliebiges ASCII-Zeichen (oder ASCII-Wert).

2) Zum Beispiel kann das ASCII-Zeichen mit dem ASCII-Wert 235 dadurch eingegeben werden, daß man die ALT-Taste gedrückt hält und schnell hintereinander im numerischen Block die Ziffern 235 eingibt.

3) Die Hex-Dasrtellung der obigen ASCII-Werte sind
EB FE ? ? ? ? ? ? # # # # 24 FF 12 00

4) Um sich die mühsame Arbeit der Eingabe der 16 Zeichen (ohne sich zu verschreiben) über Tastatur zu sparen, kann man diese 16 Zeichen mit Hilfe eines Hexeditors in eine Datei (z.B. schnell1.txt) schreiben und in der Eingabeaufforderung (DOS-Fenster) aufrufen:
`demo_overflow1 < schnell1.txt`

1.4.2.1 Beispiel (kein Angriff: nur reservierter Speicher wird überschrieben)

Adressen	Beschreibung	Beispiel
0x0012FF10	lokale Variable lok2 (reservierter Speicher)	a
0x0012FF11		b
0x0012FF12		c
0x0012FF13		d
0x0012FF14		e
0x0012FF15		0
0x0012FF16		
0x0012FF17		
0x0012FF18		
0x0012FF19		
0x0012FF1A		
0x0012FF1B		
0x0012FF1C		
0x0012FF1D		
0x0012FF1E		
0x0012FF1F		
0x0012FF20		
0x0012FF21		
0x0012FF22		
0x0012FF23		
0x0012FF24	lokale Variable lok1 (reservierter Speicher)	a
0x0012FF25		b
0x0012FF26		c
0x0012FF27		d
0x0012FF28		e
0x0012FF29		0
0x0012FF2A		
0x0012FF2B		
0x0012FF2C	alter Wert von Register Register ebp	
0x0012FF2D		
0x0012FF2E		
0x0012FF2F		
0x0012FF30	Rücksprung-adresse in main	
0x0012FF31		
0x0012FF32		
0x0012FF33		

1.4.2.2 Beispiel (Angriff: nichtreservierter Speicher wird überschrieben)

Adressen	Inhalt	Beschreibung	Befehle	Maschinencode
0x0012FF10	Exploit	Speicher für lokale Variable lok2. Hier werden die über Tastatur eingegebenen Zeichen zuerst abgespeichert. Hier gibt der Hacker seine spezielle Zeichenkette ein, den Exploit	M: jmp M	EB
0x0012FF11				FE
0x0012FF12			irgendwelche Zeichen. Diese werden nur zum Auffüllen benutzt	?
0x0012FF13				?
0x0012FF14				?
0x0012FF15				?
0x0012FF16				?
0x0012FF17				?
0x0012FF18				#
0x0012FF19				#
0x0012FF1A				#
0x0012FF1B				#
0x0012FF1C			Anfangsadresse des Exploit in der Variable lok1	24
0x0012FF1D				FF
0x0012FF1E				12
0x0012FF1F				00
0x0012FF20	nicht genutzter Speicherbereich			
0x0012FF21				
0x0012FF22				
0x0012FF23				
0x0012FF24	kopierter Exploit	Speicher für lokale Variable lok1. Hier wird der Speicher von lok2 hinein kopiert.		EB
0x0012FF25				FE
0x0012FF26				?
0x0012FF27				?
0x0012FF28				?
0x0012FF29				?
0x0012FF2A				?
0x0012FF2B				?
0x0012FF2C		alter Wert (*) von Register ebp	(*) wird überschrieben durch: #### (siehe rechts)	#
0x0012FF2D				#
0x0012FF2E				#
0x0012FF2F				#
0x0012FF30		Rücksprung-adresse (**) in main	(**) wird überschrieben durch: 24FF1200 (siehe rechts)	**24**
0x0012FF31				**FF**
0x0012FF32				**12**
0x0012FF33				**00**

In dem gestrichelt gezeichneten Speicherbereich findet der Buffer-Overflow statt !!

Der letzte Befehl ret in dem Unterprogramm (in der Funktion f) veranlaßt, daß der Wert, der gerade auf dem Stack liegt (Rücksprungadresse 0012FF24) in den Befehlszähler eip geladen wird, was bedeutet, daß der nächste auszuführende Befehl an der Adresse 0012FF24 steht. Dort befindet sich aber der Befehl jmp M, der in eine Endlosschleife geht.
Bemerkung: Der Wert des Stackpointer esp ist dann esp = 0x0012FF34.