Lehrstuhl Systemsicherheit
Prof. Dr. Thorsten Holz
Dipl.-Inf. Ralf Hund
Dipl.-Inf. Carsten Willems

Vorlesung Programmanalyse
Herbst-/Wintersemester 2011/2012
Ruhr-Universität Bochum

2. Übung

10. November 2011
Abgabe der Hausaufgaben: per Moodle bis zum 24. November 2011
Bei Fragen und Problemen können Sie sich per E-Mail/Moodle an die Tutoren wenden.

Aufgaben

Aufgabe 1: Rekonstruktion einer C-Funktion (1 Punkt)

Rekonstruieren Sie die C-Funktion, die zu folgenden Assembleranweisungen kompiliert wurde.

```
00401290  PUSH  EBP
00401291  MOV   EBP, ESP
00401293  SUB   ESP, 4
00401296  MOV   DWORD PTR [EBP-4], 1

0040129D  CMP   DWORD PTR [EPB+8], 9
004012A1  JG    SHORT 004012BD
004012A3  LEA   EAX, [EBP+12]
004012A6  DEC   DWORD PTR [EAX]
004012A8  CMP   DWORD PTR [EBP+12], 9
004012AC  JG    SHORT 004012B0
004012AE  JMP   SHORT 004012BD

004012B0  MOV   EDX, [EBP+8]
004012B3  LEA   EAX, [EBP-4]
004012B6  ADD   DWORD PTR [EAX], EDX
004012B8  INC   DWORD PTR [EBP+8]
004012BB  JMP   SHORT 0040129D

004012BD  MOV   EAX, DWORD PTR [EBP-4]
004012C0  MOV   ESP, EBP
004012C2  POP   EBP
004012C3  RETN
```

Aufgabe 2: Assembler Subroutinen (1 Punkt)

Beim Analysieren einer Binärdatei stoßen Sie auf folgenden Subroutinen-Aufruf im Code des Programms:

Instruktion	Kommentar
`push eax`	
`push ecx`	
`call CAFEBABEh`	
`add esp, 8`	
`...`	

An der Speicherstelle `CAFEBABEh` befinden sich die in der folgenden Tabelle aufgeführten Instruktionen:

Instruktion	Kommentar
`push ebp`	Wert von `ebp` wird auf Stack gesichert.
`mov ebp, esp`	
`sub esp, 4`	
`mov ecx, [ebp+8]`	
`add ecx, [ebp+12]`	
`mov [ebp-4], ecx`	
`dec dword ptr[ebp-4]`	
`mov eax, [ebp-4]`	
`mov esp, ebp`	
`pop ebp`	
`ret 8`	

1. Geben Sie zu jeder Instruktion in dem entsprechenden Kommentarfeld an, welchem Zweck die Instruktion dient.
2. Was macht die Subroutine genau? Wie könnte die entsprechende Subroutine in der Hochsprache C aussehen?
3. Welche Instruktionen in obiger Tabelle zählen zum Prolog, welche zum Epilog?
4. Beim Ausführen bemerken Sie, dass das Programm immer abstürzt. Durch Debugging lässt sich jedoch relativ schnell erkennen, dass die Abstürze erst nach dem Funktionsaufruf auftreten. Der Fehler liegt allerdings trotzdem im oberen Code. Warum stürzt das Programm ab? Warum tritt der Fehler erst später auf?

Aufgabe 3: Fibonacci in Assembler (2 Punkte)

Schreiben Sie ein Konsolenprogramm in Assembler, welches die Fibonacci-Folge (1 1 2 3 5 ...) einer eingegebenen Zahl n (mit $2 \leq n \leq 47$) ausgibt. Bitte nutzen Sie '*fibonacci.asm*' als Grundgerüst und implementieren Sie sowohl eine iterative (`_Fibonacci_Iterative(n)`), als auch eine rekursive Version (`_Fibonacci_Recursive(n)`).

Aufgabe 4: Indirekte Adressierung in x86-Assembler (2 Punkte)

Angenommen, es stehen die folgenden Variablendeklarationen in der Hochsprache C zur Verfügung (`int` steht für eine 32-bit breite Variable, `short` für eine 16-bit breite Variable, `char` für eine 8-bit breite Variable), wobei die entsprechenden Werte oder Adressen zum Ausführungszeitpunkt in die verschiedenen Register geladen wurden, wie in den Kommentaren angegeben:

```
int i;          // Wert in ecx
int x;          // Adresse in ebx
short my[64];   // Adresse in edx
char ch[8];     // Adresse in edi
int arr[256];   // Adresse in esi
```

Übersetzen Sie die folgenden Hochsprachen-Ausdrücke, wie bei den ersten zwei Beispielen bereits geschehen, in jeweils **einen einzigen** x86-Befehl, falls dies möglich ist. Sollten unbedingt mehrere Befehle nötig sein, trennen Sie diese bitte mit einem Semikolon (";"). Jede unnötige Angabe soll vermieden werden. Das Ergebnis soll immer in das Register `eax` geladen werden, welches zuvor jeweils einen unbekannten Wert enthält.

Hochsprachen-Ausdruck	x86-Befehl
`i`	`mov eax, ecx`
`x`	`mov eax, [ebx]`
`arr[10]`	
`my[i]`	
`arr[i + 10]`	
`arr[i] + 10`	
`my[2*i]`	
`my[2*i + 1]`	
`ch[i - 1]`	
`ch[x - 1]`	

Hinweise:

- Achten sie besonders auf die verschiedenen Bitlängen der Variablen!
- Der Arbeitsspeicher wird byteweise adressiert, d.h. jedes Byte hat eine eigene Adresse.
- Wieviel gelesen werden soll, kann man mit `dword ptr [...]`, `word ptr [...]` oder `byte ptr [...]` begrenzen, ansonsten wird das Zielregister komplett gefüllt.
- Der Befehl `movzx` *(MOVe Zero eXtended)* hilft bei Operanden unterschiedlicher Länge, er füllt die nicht vorhandenen höherwertigen Stellen mit Nullen auf, um Reste alter Werte zu beseitigen, und funktioniert ansonsten wie der Befehl `mov`.

Aufgabe 5: Analyse von Programmen I (3 Punkte)

1. Analysieren Sie das Programm *XorMadness.exe*. Laden Sie dazu die Datei in einen Debugger Ihrer Wahl (z.B. OllyDbg) und finden Sie die passende Eingabe, um die Erfolgsmeldung auszugeben. Achten Sie zunächst darauf, welche Eingaben akzeptiert und wie diese weiter verarbeitet werden. Es besteht ebenfalls die Möglichkeit, die Lösung zu bruteforcen - hierzu können Sie ein kleines Programm (in C) schreiben, welches die Arbeit für Sie übernimmt. Geben Sie, falls vorhanden, den Quelltext Ihres Bruteforcers mit ab. (1 Punkt)
2. Jetzt sollen Sie ein Programm erstellen, welches die Lösung von *XorMadness.exe* direkt (ohne Bruteforce!) ausgibt. Dazu sollen Sie den Algorithmus der exe-Datei „umgekehrt" implementieren und somit den richtigen Schlüssel berechnen. Geben Sie ihren C-Quelltext ab! (2 Punkte)

Aufgabe 6: Analyse von Programmen II (3 Punkte)

Analysieren Sie das Programm *what.exe*, bei dem kein Lösungswort existiert. Sie sollen lediglich herausfinden was das Programm macht und das Verhalten in 1-2 kurzen Sätzen erläutern. Beschreiben Sie auch hier, wie Sie vorgegangen sind.

Abgabe der Übungen

Die Hausaufgaben sind bis zur übernächsten Woche in Moodle einzureichen. Sie erreichen die Moodle-Seite per `http://moodle.rub.de`: Loggen Sie sich in das System ein und wählen Sie unter *Meine Kurse* die Veranstaltung *Programmanalyse WS 11/12* aus.