Sichere Programmierung
Projekt 2

Julian Sobott
(76511)
David Sugar
(76050)

Inhaltsverzeichnis

1 Zu Aufgabe 1

1.1 a)

Zu beginn der `main()` Funktion wird eine `unsigned int` Variable, i, deklariert, jedoch nicht initialisiert, d.h. bis auf wenige Ausnahmen $i \in \{0..2^{32}-1\}$.

Danach wird die Variable im Kopf der darauf folgenden For-Schleife mit 0 initialisiert. Die Schleife inkrementiert die Variable i am Ende jedes Schleifendurchlaufs und tritt erneut in die Schleife ein, solange i kleiner 20 ist. Innerhalb der Schleife wird der Wert von i, zum jeweiligen Zeitpunkt, formatiert mithilfe von `printf()` in der Standardausgabe ausgegeben. Dabei werden immer 2 Stellen ausgegeben, dies wird über `"%2d"` realisiert.

Potentielles Problem: Es sollte `"%2u"` verwendet werden, da `d` für die Formatierung von signed Integern verwendet wird. In diesem Fall spielt die Formatierung aber keine Rolle.

1.2 b)

Bild 1 zeigt die Ausgabe des Programms.

1.3 c)

```
1     <+8>:   mov    DWORD PTR [rbp-0x4],0x0
2     <+15>:  jmp    0x40113b <main+41>
3     <+17>:  mov    eax,DWORD PTR [rbp-0x4]
4     <+20>:  mov    esi,eax
5     <+22>:  mov    edi,0x402004
6     <+27>:  mov    eax,0x0
7     <+32>:  call   0x401030 <printf@plt>
8     <+37>:  add    DWORD PTR [rbp-0x4],0x1
9     <+41>:  cmp    DWORD PTR [rbp-0x4],0x13
10    <+45>:  jbe    0x401123 <main+17>
```

Für die Variable i wird Speicher auf dem Stack alloziert, die Anfangsadresse ist dabei `rbp-0x4`.

In Zeile <+8> wird i mit 0x0 initialisiert. Danach springt das Programm unbedingt in Zeile <+41>. Hier befindet sich nun die Überprüfung, ob die Schleife verlassen wird, d.h. $i \geq 0x14$, oder ein weiterer Schleifendurchlauf gestartet wird. Dazu wird in Zeile <+41> i mit 0x13 verglichen. Ist der Wert kleiner oder gleich 0x13 wird in Zeile <+17> gesprungen und damit ein weiterer Schleifendurchlauf gestartet. Andernfalls wird die nächste Instruktion ausgeführt und damit die Schleife verlassen.

In Zeile <+17> und <+20> wird der Wert von i, vom Speicher in das `esi` Register geladen. In der darauf folgenden Zeile wird die Adresse des Formatierungsstrings (”i: %2d n”) (0x402004) in `edi` geladen.

Abbildung 1: Ausgabe von gdb-uebung-1.c

```
1
2 gef  x/s 0x402004
3 0x402004:   "i: %2d\n"
```

Weiterhin wird `eax` wieder auf 0x0 zurückgesetzt. Danach wird `printf()` mit den in `edi` und `esi` geladenen Parametern aufgerufen. Schlussendlich wird i inkrementiert und daraufhin wieder verglichen (<+41>).

1.4 •

2 Zu Aufgabe 3

2.1 a) Analysieren Sie den in der Datei enthaltenen Source Code

Das bereitgestellte Quelldatei `gdb-uebung-3.c` enthält die rekursive Implementierung eines **Factorial-Algorithmus**, damit ist $f(n) = n!$.

Def: $n! = n * (n-1) * (n-2) * ... * 1 = \prod_{i=1}^{n} i,\ n \in \mathbb{N}$

2.1.1 Implementierung

```
1 unsigned int f(unsigned int i) {
2   if (i>1) {
3     return i * f(i-1);
4   } else {
5     return 1;
6   }
7 }
```

Abbildung 2: factorial function

Die Funktion, abgebildet in 2.1.1 nimmt einen vorzeichenlosen Integer als Argument und gibt als Ergebnis ebenfalls einen vorzeichenlosen Integer zurück. Dabei ist **int** jedoch betriebssystemabhängig definiert. Für die meisten Systeme kann jedoch angenommen werden, dass es sich dabei um ein **4 Byte** großes Wort handelt, d.h für den Rückgabewert kommen Werte innerhalb des Wertebereichs $W = [0, 2^{32} - 1]$ in Frage. Aufgrund des extrem schnellen Wachstums von $n!$ ist dies ein sehr beschränkender Faktor, der bei der Nutzung unbedingt mit zu berücksichtigen ist, da es schnell zu einem Überlauf und damit zu einer Verfälschung des Ergebnisses kommen kann. In Zeile 2 wird danach geprüft, ob der übergebene Wert größer 1 ist. Sollte dies der Fall sein, wird das Produkt von i und dem Ergebnis des Rekursiven Aufrufs von $f(i-1)$ zurückgegeben. Andernfalls wird die Konstante 1 als Rückgabewert der Funktion genutzt, siehe Zeile 5.

2.1.2 Aufruf

```
1 int main() {
2   unsigned int i=5, r=0;
3
4   r = f(i);
5
6   printf("i = %d, f(i) = %d\n", i, r);
7 }
```

Abbildung 3: invocation of f()

In der **main()** Funktion wird die in 2.1.1 beschriebene Funktion **f(unsigned int)** aufgerufen und dabei **i = 5** als Argument übergeben. Das Ergebnis des Aufrufs wird der Variable **int r** zugewiesen, siehe Zeile 4. Danach wird **i** und das Ergebnis **r** mittels **printf()** auf der Kommandozeile ausgegeben.

2.2 b) Kompilieren Sie den C Code und führen Sie das Programm aus

2.2.1 Kompilieren

Das Kompilieren des Quellkodes innerhalb von `gdb-uebung-3.c` kann mittels des folgenden Befehls auf der Kommandozeile ausgeführt werden.

```
1 $ gcc gdb-uebung-3.c -o gdb-uebung-3
```

Der in diesem Fall genutzte Compiler heißt **gcc** (GNU compiler collection). Dabei ist gdb-uebung-3.c der Name der Quelldatei. Mittels **-o gdb-uebung-3** wird der gewünschte Name, der zu erstellenden Programmdatei, angegeben.

2.2.2 Ausführen

Das Programm kann nun auf der Kommandozeile ausgeführt werden.

```
1 $ ./gdb-uebung-3
2 i = 5, f(i) = 120
```

Zum verifizieren des Ergebnisses kann dieses auch noch einmal Händisch berechnet werden, $\prod_{i=1}^{5} i = 1 * 2 * 3 * 4 * 5 = 2 * 3 * 20 = 2 * 60 = 120$. Das Ergebnis stimmt, die Funktionen scheint auf den ersten Blick also richtig implementiert. Um nachzuvollziehen, wie die das Ergebnis zustande kommt, kann der folgende Graph betrachtet werden.

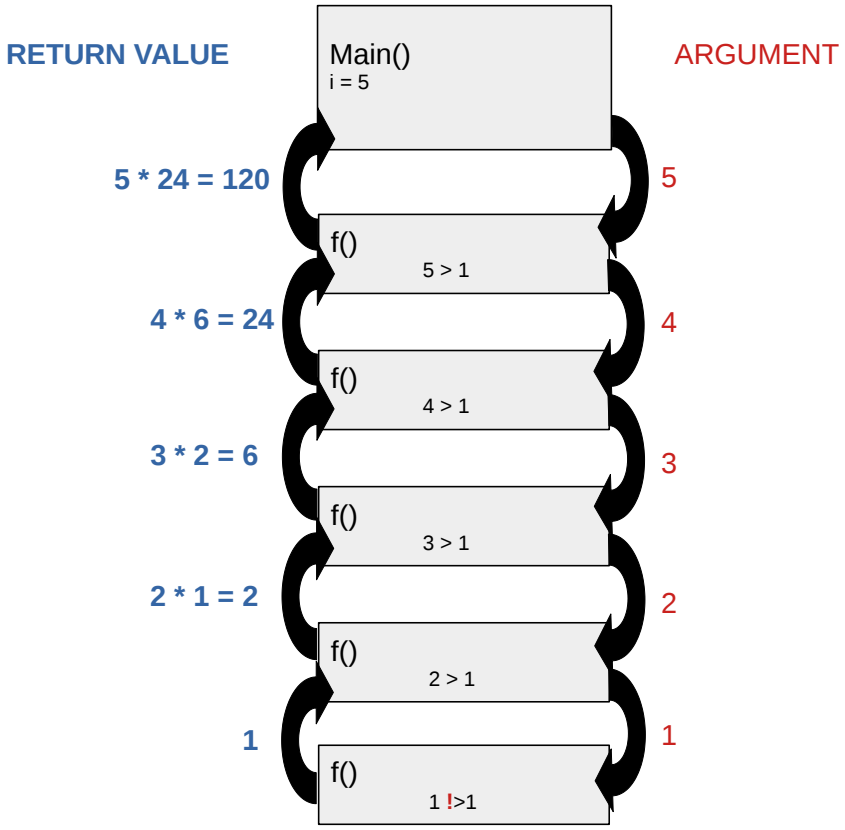

Abbildung 4: recursive call of f()

Anfangs wird in **main()** die Funktion **f()** mit **5 als Argument** aufgerufen. Die aufgerufene Funktion f(5) prüft nun, ob der Parameter größer als 1, d.h. $\mathbf{5 > 1}$, ist. Ist dies der Fall, ruft sich die Funktion selbst wieder auf, dieses Mal jedoch mit dem **dekrementierten** Parameter als Argument. Dies wiederholt sich bis 1 bzw. 0 übergeben wird, in diesem Fall wird 1 zurückgegeben und das Ergebnis 'aufsteigend' berechnet.

2.3 c) Führen Sie das Programm im GDB aus

2.3.1 Wie viele Stack Frames werden erzeugt?

2.3.2 Wie ist der Inhalt dieser Stack Frames?

2.3.3 Wie wird die Parameterübergabe in Assembler umgesetzt?

Für die Übergabe von Parametern an Subroutinen muss unter x86 eine Fallunterscheidung gemacht werden. Je nachdem, ob es sich um Programme für ältere 32-Bit Prozessoren handelt oder um Programme für neuere 64-Bit Prozessoren, werden verschiedene sog. **Calling Conventions** (Aufruf Konventionen) verwendet. Diese Konventionen dienen dazu einen

Ablauf zu definieren, sodass unabhängig vom Autor des Codes darauf vertraut werden kann, dass Abläufe wie z.B. ein Unterprogrammaufruf **immer** auf die selbe Weise durchgeführt werden.

32-Bit

Ein Unterprogrammaufruf kann in folgende Schritte untergliedert werden.

1. Zuerst müssen die **Caller Saved Register**, falls nötig, auf dem Stack gespeichert werden, da die **aufgerufene** Funktion für diese Register keine Garantie übernimmt, dass diese nicht überschrieben werden. Die Register sind: ebx, ecx, edx, r10, r11.

2. Danach müssen die Parameter in **umgekehrter Reihenfolge** auf den Stack gepushed werden. Aufgrund der Funktionsweise des Stacks ist dann der Erste Parameter der Subroutine direkt angrenzend and die gespeicherte Rücksprungadresse, die im nächsten Schritt auf dem Stack hinterlegt wird.

3. Nun wird mit **call** der Unterprogrammaufruf durchgeführt. Dabei wird die **Rücksprungadresse** (die Adresse des auf call folgenden Befehlswortes) auf den Stack gepushed und ein Sprung zum ersten Befehl des Unterprogramms durchgeführt.