

Sichere Programmierung

Projekt 2

Julian Sobott
(76511)
David Sugar
(76050)

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.

Potentiell Problem: Es sollte `"%2u"` verwendet werden, da `d` für die Formatierung von signed Integer 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,

```

Praktikum2|master ⚡ ⇒ gcc gdb-uebung-1.c -o gdb-uebung-1
Praktikum2|master ⚡ ⇒ ./gdb-uebung-1
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
i: 19

```

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

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.

```

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 d)

In dieser Aufgabe geht es nun darum das Programm `gdb-uebung-1.c` in `gdb` auszuführen. Im folgenden wird der Ablauf durch Screenshots und entsprechende Erklärungen beschrieben.

```

0x40110e <__do_global_ctors_aux+46> add    bl, bpl
0x401111 <frame_dummy+1>    mov     ss, WORD PTR [rbp+0x48]
0x401114 <main+2>          mov     ebp, esp
→ 0x401116 <main+4>        sub     rsp, 0x10
0x40111a <main+8>          mov     DWORD PTR [rbp-0x4], 0x0
0x401121 <main+15>         jmp     0x40113b <main+41>
0x401123 <main+17>         mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>         mov     esi, eax
0x401128 <main+22>         mov     edi, 0x402004

```

Abbildung 2: 1. Ausgabe von gdb-uebung-1.c

1. Hier beginnt die `main` Funktion. Als erstes wird der `rsp` Zeiger, welcher auf den Stack zeigt, um `0x10` verschoben, um entprechen Platz auf den Stack zu allozieren.

```

0x401111 <frame_dummy+1>    mov     ss, WORD PTR [rbp+0x48]
0x401114 <main+2>          mov     ebp, esp
0x401116 <main+4>          sub     rsp, 0x10
→ 0x40111a <main+8>        mov     DWORD PTR [rbp-0x4], 0x0
0x401121 <main+15>         jmp     0x40113b <main+41>
0x401123 <main+17>         mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>         mov     esi, eax
0x401128 <main+22>         mov     edi, 0x402004
0x40112d <main+27>         mov     eax, 0x0

```

Abbildung 3: 2. Ausgabe von gdb-uebung-1.c

2. Initialisieren der Variable `i` mit `0x0`.

```

0x401113 <main+1>          mov     rbp, rsp
0x401116 <main+4>          sub     rsp, 0x10
0x40111a <main+8>          mov     DWORD PTR [rbp-0x4], 0x0
→ 0x401121 <main+15>        jmp     0x40113b <main+41>
0x401123 <main+17>         mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>         mov     esi, eax
0x401128 <main+22>         mov     edi, 0x402004
0x40112d <main+27>         mov     eax, 0x0
0x401132 <main+32>         call    0x401030 <printf@plt>

```

Abbildung 4: 3. Ausgabe von gdb-uebung-1.c

3. Unbedingter Sprung in Zeile `<main+41>`.

```
gef> x/d $rbp-0x4
0x7fffffffcdc9c: 0
```

Abbildung 5: 4. Ausgabe von gdb-uebung-1.c

4. Ausgabe von i. (Adresse: Wert). Der Wert wird in Dezimal ausgegeben.

Schritte bis zur nächsten Zeile wurden übersprungen, da sie in Aufgabe 1 b) ausführlich erklärt wurden.

```
→ 0x40113f <main+45>      jbe    0x401123 <main+17>      TAKEN [Reason: C || Z]
↳ 0x401123 <main+17>      mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>      mov     esi, eax
0x401128 <main+22>      mov     edi, 0x402004
0x40112d <main+27>      mov     eax, 0x0
0x401132 <main+32>      call   0x401030 <printf@plt>
0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
```

Abbildung 6: 5. Ausgabe von gdb-uebung-1.c

5. Der bedingte Sprung jbe (jump below or equal) wird genommen, da $0x0 \leq 0x13$. Das heißt, das Programm springt zu <main+17>.

```
→ 0x401123 <main+17>      mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>      mov     esi, eax
0x401128 <main+22>      mov     edi, 0x402004
0x40112d <main+27>      mov     eax, 0x0
0x401132 <main+32>      call   0x401030 <printf@plt>
```

Abbildung 7: 6. Ausgabe von gdb-uebung-1.c

6. Schreiben des Wertes von i in eax und eax dann in esi, um i als Parameter an die printf Funktion zu übergeben.

```
→ 0x401132 <main+32>      call   0x401030 <printf@plt>
↳ 0x401030 <printf@plt+0>  jmp     QWORD PTR [rip+0x2fe2]      # 0x404018 <printf@got.plt>
0x401036 <printf@plt+6>  push    0x0
0x40103b <printf@plt+11>  jmp     0x401020
0x401040 <_start+0>      xor     ebp, ebp
0x401042 <_start+2>      mov     r9, rdx
0x401045 <_start+5>      pop     rsi
```

Abbildung 8: 7. Ausgabe von gdb-uebung-1.c

```
$rsi : 0x0
$rdi : 0x0000000000402004 → 0x000a643225203a69 ("i: %2d"?)
```

Abbildung 9: 8. Ausgabe von gdb-uebung-1.c

7.+ 8. Aufrufen der `printf` Funktion mit `i=0`. Übergeben wird in `rsi` und `rdi` der Wert von `i` und ein pointer auf den format string. Dieser Aufruf führt zu folgender Ausgabe auf dem Standardoutput:

```
1 i: 0
```

```

0x401128 <main+22>      mov     edi, 0x402004
0x40112d <main+27>      mov     eax, 0x0
0x401132 <main+32>      call    0x401030 <printf@plt>
→ 0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>      cmp     DWORD PTR [rbp-0x4], 0x13
0x40113f <main+45>      jbe     0x401123 <main+17>
0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret

```

Abbildung 10: 9. Ausgabe von `gdb-uebung-1.c`

9. Hier wird der Wert von `i` nun um eins erhöht.

```
gef> x/d $rbp-0x4
0x7fffffffddccc: 1
```

Abbildung 11: 10. Ausgabe von `gdb-uebung-1.c`

10. Nach der ausführung ist der Wert 1.

```

gef> x/d $rbp-0x4
0x7fffffffddccc: 0
gef> b *main+45 if *0x7fffffffddccc == 0x13
Breakpoint 2 at 0x40113f
gef> info break
Num      Type             Disp Enb Address              What
1        breakpoint       keep y  0x0000000000401116 <main+4>
          breakpoint already hit 1 time
2        breakpoint       keep y  0x000000000040113f <main+45>
          stop only if *0x7fffffffddccc == 0x13

```

Abbildung 12: 11. Ausgabe von `gdb-uebung-1.c`

11. Als nächstes wollen wir das Programm bis zum letzten Durchlauf laufen lassen und dort dann einen Breakpoint setzen. Als erstes geben wir uns hierfür die Adresse für `i` aus. Diese wird benötigt, da der Conditional Breakpoint nur stoppen soll, wenn `i` einen bestimmten Wert hat. In der nächsten Zeile setzen wir den Conditional Breakpoint in die Zeile wo der bedingte Sprung ist (`<main+45>`). Als Bedingung geben wir an, dass der Wert von `i` gleich `0x13` sein soll. Wie auch in C müssen Adressen jeweils mit dem `*` dereferenziert werden. Am Ende wird noch kontrolliert ob der breakpoint richtig gesetzt wurde.

```
gef> c
Continuing.
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
```

Abbildung 13: 12. Ausgabe von gdb-uebung-1.c

12. Lassen wir das Programm nun mit continue (c) laufen, sehen wir alle Schleifendurchläufe mit den entsprechenden Ausgaben. Die letzte Ausgabe ist 18 (0x12).

```
0x401132 <main+32>    call    0x401030 <printf@plt>
0x401137 <main+37>    add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>    cmp     DWORD PTR [rbp-0x4], 0x13
→ 0x40113f <main+45>    jbe     0x401123 <main+17>          TAKEN [Reason: C || Z]
↳ 0x401123 <main+17>    mov     eax, DWORD PTR [rbp-0x4]
0x401126 <main+20>    mov     esi, eax
0x401128 <main+22>    mov     edi, 0x402004
0x40112d <main+27>    mov     eax, 0x0
0x401132 <main+32>    call    0x401030 <printf@plt>
0x401137 <main+37>    add     DWORD PTR [rbp-0x4], 0x1

[#0] Id 1, Name: "gdb-uebung-1", stopped, reason: BREAKPOINT

[#0] 0x40113f → main()

Breakpoint 2, 0x000000000040113f in main ()
gef> x/d $rbp-0x4
0x7fffffffdbcc: 19
```

Abbildung 14: 13. Ausgabe von gdb-uebung-1.c

13. Der bedingte Sprung wird ein letztes Mal genommen, da der Wert von i gleich 19 (0x13) ist.

```

0x401132 <main+32>      call    0x401030 <printf@plt>
0x401137 <main+37>      add     DWORD PTR [rbp-0x4], 0x1
0x40113b <main+41>      cmp     DWORD PTR [rbp-0x4], 0x13
→ 0x40113f <main+45>      jbe     0x401123 <main+17>      NOT taken [Reason: !(C || Z)]
0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret
0x401148               nop     DWORD PTR [rax+rax*1+0x0]
0x401150 <__libc_csu_init+0> push    r15

```

```

[#0] Id 1, Name: "gdb-uebung-1", stopped, reason: SINGLE STEP

```

```

[#0] 0x40113f → main()

```

```

0x000000000040113f in main ()
gef> x/d $rbp-0x4
0x7fffffffddccc: 20
gef> x/h $rbp-0x4
0x7fffffffddccc: 20
gef> x/x $rbp-0x4
0x7fffffffddccc: 0x0014

```

Abbildung 15: 14. Ausgabe von gdb-uebung-1.c

14. Die Schleife ist eine letztes Mal durchgelaufen wie erwartet und hat noch die 19 ausgegeben. Wenn wir nun aber an dem bedingten Sprung angekommen wird hier nicht mehr genommen, da die Bedingung nicht mehr zutrifft ($i = 0x14$ und somit gilt nicht mehr $i \leq 0x13$).

```

0x401136 <main+36>      inc     DWORD PTR [rbx-0x7cfe03bb]
0x40113c <main+42>      jge     0x40113a <main+40>
0x40113e <main+44>      adc     esi, DWORD PTR [rsi-0x1e]
→ 0x401141 <main+47>      mov     eax, 0x0
0x401146 <main+52>      leave
0x401147 <main+53>      ret
0x401148               nop     DWORD PTR [rax+rax*1+0x0]
0x401150 <__libc_csu_init+0> push    r15
0x401152 <__libc_csu_init+2> lea     r15, [rip+0x2ca7]      # 0x403e00

```

Abbildung 16: 15. Ausgabe von gdb-uebung-1.c

15. Anstatt an `<main+17>` zu springen, wurde zur nächsten Anweisung gesprungen `<main+47>`. Somit wurde die Schleife verlassen. Hier wird noch die 0 als Rückgabewert gespeichert.

```

gef> c
Continuing.
[Inferior 1 (process 22361) exited normally]
gef> 

```

Abbildung 17: 16. Ausgabe von gdb-uebung-1.c

16. Mit `continue (c)` lassen wir das Programm zuende durchlaufen und es wird normal beendet.

2 Zu Aufgabe 1

2.1 a)

Analyse des Programs *gdb-uebung-2.c*

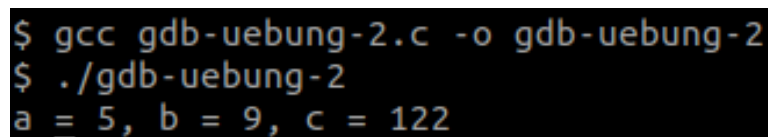
```
1 #include <stdio.h>
2
3 int f(int a, int b) {
4     return 3*a + 7*b;
5 }
6
7 int g(int a, int b) {
8     return 10*a*a - 3*b;
9 }
10
11 int h(int a, int b) {
12     return a + b + 300;
13 }
14
15 int main() {
16     int a = 5, b=9, c=0;
17
18     c = f(g(a,h(a,b)),h(b,a));
19
20     printf("a = %d, b = %d, c = %d\n", a, b, c);
21 }
```

Das Programm besteht aus drei Hilfsfunktionen *f*, *g*, *h*, die jeweils 2 int's als Eingabe bekommen und mit Grundrechenarten ein Ergebnis berechnen und zurück geben. Die Berechnungen scheinen willkürlich sein.

In der *main* Funktion, werden zuerst drei int's *a*, *b*, *c* initialisiert werden. Daraufhin wird in einem geschachtelten Funktionsaufruf der Wert von *c* berechnet. In den Funktionsaufrufen, werden die Rückgabewerte einer Hilfsfunktion immer direkt an die nächste Funktion, als Parameter, übergeben. Am Ende wird durch ein Aufruf der *printf* Funktion ein formatierter String mit den Werten von den Variablen ausgegeben.

2.2 b)

Bild 18 zeigt die Ausgabe des Programms.



```
$ gcc gdb-uebung-2.c -o gdb-uebung-2
$ ./gdb-uebung-2
a = 5, b = 9, c = 122
```

Abbildung 18: Ausgabe von *gdb-uebung-2.c*

Wie erwartet sind die Werte von **a**, **b** unverändert zur ursprünglichen Initialisierung. Nur **c** hat den neuen Wert, den es zugewiesen bekommen hat.

2.3 c)

2.3.1 Reihenfolge der Funktionsaufrufe

Die Reihenfolge wurde herausgefunden, indem in gdb mit **s** immer der nächste Schritt ausgeführt wurde. Hierfür wurde das Programm mit Debug Informationen compiliert (Option **-g** bei gcc).

1. **h(9, 5)** (2. **h**)
2. **h(5, 9)** (1. **h**)
3. **g(5, 314)**
4. **f(-692, 314)**

Interessant bei der Ausführung ist, dass bei dem Aufruf von **f** zuerst der 2. Parameter (**h**) ausgewertet wird und dann erst der 1. (**g**).