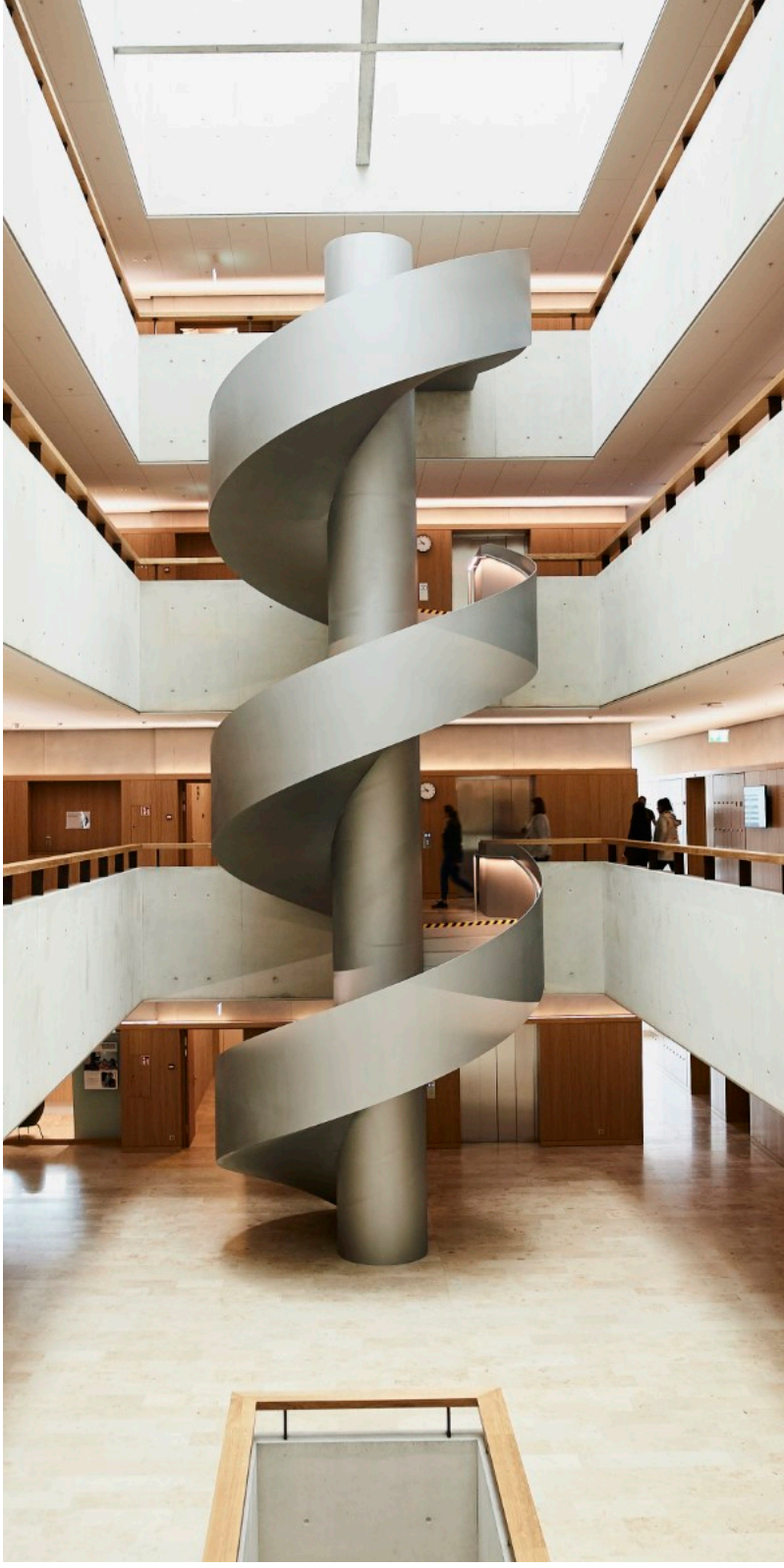
Hochschule Luzern
Modul I.BA_OSSEC



Laborübung

Buffer Overflow Teil 1

HSLU Hochschule Luzern

Dokument Version: #03ead27
Template Version: #703559b
Letzte Änderung: 2024-01-29
S. Miescher, J. Zavrlan, M. Useini

I. Allgemeine Informationen

Name:

Gruppe:

Bemerkungen:

Liste der Verfasser

S. Miescher	Creator
J. Zavrlan	Umwandlung in Latex, Korrektur, Feinschliff
M. Useini	Prüflesen, Testen

Copyright Informationen

II. Inhaltsverzeichnis

III. Vorwort

Feedback

Mit Ihrer Mithilfe kann die Qualität des Versuches laufend den Bedürfnissen angepasst und verbessert werden.

Falls in diesem Versuchsablauf etwas nicht so funktioniert wie es beschrieben ist, melden Sie dies bitte direkt dem Laborpersonal oder erwähnen Sie es in Ihrem Laborbericht oder Protokoll. Behandeln Sie die zur Verfügung gestellten Geräte mit der entsprechenden Umsicht.

Bei Problemen wenden Sie sich bitte ebenfalls an das Laborpersonal.

Legende

In den Versuchen gibt es Passagen, die mit den folgenden Boxen markiert sind. Diese sind wie folgt zu verstehen:

Wichtig

Dringend beachten. Was hier steht, unbedingt merken oder ausführen.

Aufgabe III.1

Beantworten und dokumentieren Sie die Antworten im Laborprotokoll.

Hinweis

Ergänzender Hinweis / Notiz / Hilfestellung.

Information

Weiterführende Informationen. Dies sind Informationen, die nicht zur Ausführung der Versuche benötigt werden, aber bekannt sein sollten.

Story

Hierbei wird die Geschichte vermittelt, die in den Versuch einleitet oder den Zweck des Versuches vorstellt.

Zielsetzung

Lernziele, die nach dem Bearbeiten des Kapitels erfüllt sein sollten.

Erkenntnis

Wichtige Erkenntnisse, die aus dem Versuch mitgenommen werden sollten.

1. Abkürzungsverzeichnis

Abkürzung	Beschreibung
GCC	GNU Compiler Collection
GDB	GNU Debugger
ASLR	Address Space Layout Randomization
ELF	Executable and Linkable Format
asm	Assembler, Assembly-Sprache
GAS	GNU Assembler
NASM	Netwide Assembler

2. Theorie

Stack Buffer Overflow bezeichnet eine verbreitete Technik zur Ausnutzung von Schwachstellen in Software, die auf fehlender oder falscher Überprüfung von Bereichsgrenzen bei Speicherzugriffen auf dem Stack basiert.

2.1. Übersicht der CPU-Architektur: Virtual Memory, Stack, Register

Moderne Betriebssysteme stellen jedem Prozess seinen eigenen virtuellen Speicherbereich zur Verfügung. Generell ist es nicht möglich, auf den Speicher eines anderen Prozesses zuzugreifen. Nachdem ein ELF-Programm vom Betriebssystem in den Hauptspeicher (RAM) geladen wurde, sieht der Adressbereich auf der x86-Architektur wie folgt aus:

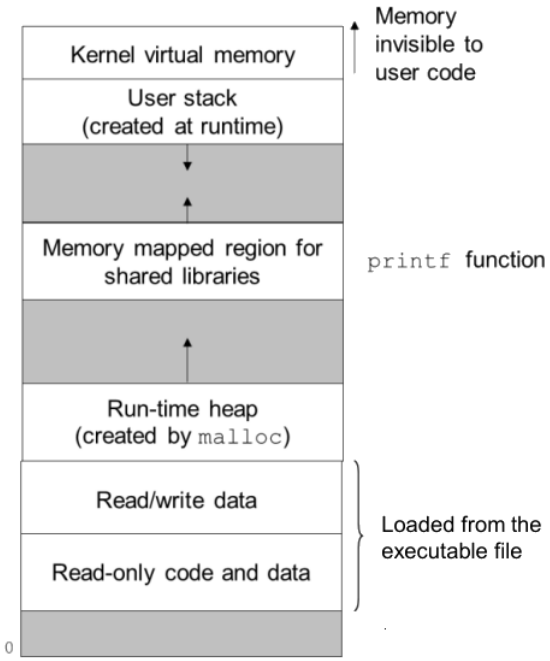


Abbildung 1: Virtueller Adressraum eines Prozesses. Höhere Adressen sind weiter oben.

Der *Stack* wird im Deutschen auch *Kellerspeicher* genannt. Dieser Name kommt vom ursprünglichen Patent, welches die Struktur so beschreibt, dass bei weiterem Platzbedarf jeweils ein weiterer Keller unter dem aktuell tiefsten Keller gegraben wird. Tatsächlich wächst der Stack im Speicher nach unten, also von höheren zu tieferen Adressen.

Auf dem Stack werden lokale Variablen einer Funktion und insbesondere auch Arrays fixer Grösse gespeichert. Zudem finden sich auf dem Stack die Funktionsparameter, welche nicht mehr in ein Register gepasst haben. Der *Stack Pointer* im Register %esp (%rsp auf 64bit-Architektur) gibt das untere Ende an. Beim Aufruf einer Funktion wird der Stack nach unten erweitert, indem der Stack Pointer um die benötigte Anzahl Bytes reduziert wird. Am Ende einer Funktion wird der Speicher freigegeben, indem der alte Stack Pointer wiederhergestellt wird. Dynamisch allozierter Speicher, der nicht an eine Funktion gebunden ist, findet sich auf dem *Heap*.

Die *Register* sind ein Set von Speichern im Prozessor, die als Zwischenspeicher und Operanden für die CPU-Instruktionen dienen. Die Register sind in der CPU-Architektur definiert; bei x86-CPUs sind sie 32bit (4 Byte), auf der *x86_64/amd64-Architektur 64bit (8 Byte) lang. Diese Länge bezeichnet auch die _Wortbreite* – wie lange eine Standard-Speichereinheit ist. Register sind quasi die Variablen, mit denen die CPU rechnet.

2.1.1. Program Counter

Der *Program Counter* (PC, im Deutschen auch «Befehlszähler») oder *Instruction Pointer* («ip»-Teil in Registernamen, selten «Instruktionszeiger» im Deutschen) ist ein besonderes Register. Darin steht die Adresse der **nächsten** auszuführenden Instruktion. Normalerweise wird bei jeder ausgeführten Instruktion auf der CPU der Program Counter um die Länge der aktuellen Instruktion erhöht. Sobald die Instruktion ausgeführt wurde, zeigt der Program Counter also auf den Anfang der nächsten Instruktion, und diese wird dann aus dem Speicher geladen und ausgeführt.

Ganz exakt wird der PC nach dem Laden der Instruktion aus dem Speicher, aber vor dem eigentlichen Ausführen der Instruktion schon erhöht. Dies ist wichtig zu wissen, wenn eine Instruktion den PC irgendwohin speichert.

Einige Instruktionen verändern den Program Counter – Sprünge, Verzweigungen (analog zu if-else-Blöcken in den meisten Programmiersprachen), Funktionsaufrufe und Rücksprung am Funktionsende. Auf Englisch: Jump, Branch, Call, Return. Unter amd64 ist der **Program Counter im Register %rip**.

2.2. amd64 Assembly-Syntax

CPUs moderner Heimcomputer haben meistens eine 64-bit x86-CPU, der Befehlssatz heisst *amd64* oder synonym *x86_64*. In dieser Übung müssen Sie Assembler-Instruktionen lesen können. Diese kommen in zwei Schreibweisen, *AT&T* Syntax und *Intel* Syntax. In dieser Übung verwenden wir AT&T, da dies die Standardeinstellung aller verwendeten Tools ist und deren Source-Destination-Reihenfolge intuitiver.

Eine Instruktion sieht folgendermassen aus:

```
1 leal -8(%ebp,%eax,2), %eax
```

Als erstes kommt der Instruktionsaufruf, hier leal. Dieser setzt sich aus dem eigentlichen **Instruktionsnamen**, lea, und einem **Längensuffix**, l, zusammen. LEA bedeutet *load effective address* und ist eine Instruktion, die eine Speicheradresse berechnet, aber nicht auf den Speicher zugreift. Dahinter folgt der Source-Operand, hier -8(%ebp,%eax,2) und schlussendlich der Destination-Operand, hier %eax. **Destination** ist bei dieser Instruktion das Zielregister, in welches das Ergebnis gespeichert wird.

Source ist hier eine Berechnung in LEA-Syntax. Die Notation $imm(%a,%b,$mul) bedeutet
%a + %b * $mul + $imm und macht in unserem Beispiel diese Berechnung: Wert aus Register %ebp plus Wert aus Register %eax mal 2 minus 8. Das Ergebnis wird wie erwähnt in %eax gespeichert. Die LEA-Syntax kann für $mul nur die Werte 1, 2, 4 und 8 annehmen, für 1 wird dieser auch weggelassen. Der Immediate-Wert $imm und eins der beiden Register können ebenfalls weggelassen werden, dann wird stattdessen implizit 0 genommen. Die LEA-Syntax wird auch in anderen Instruktionen verwendet und nennt sich auch *indirect addressing mode*.

Valide **Suffixe** am Instruktionsnamen sind **b** (byte = 8bit), **w** (word = 16bit), **l** (long = 32bit), und **q** (quad = 64bit). Beachten Sie, dass hier ein Wort 16bit bedeutet, was **nicht** der *amd64*-Wortbreite von 64bit entspricht.

Register beginnen mit einem % und haben unterschiedliche Namen für verschiedene Längen. Bei 64bit-Prozessoren sind alle Register 64bit lang, können aber auch für Instruktionen mit kürzeren Operanden verwendet werden. Dann hat das gleiche Register einen anderen Namen: Das Register %rax heisst im Beispiel oben %eax, weil die 32bit-Instruktion (ersichtlich aus dem l-Suffix) nur auf die unteren 32bit des Registers zugreift. Für 16bit-Instruktionen heisst das Register %ax, für 8bit-Instruktionen %al (das *l* hier für *lower*).

Weiter gibt es sogenannte **Immediate**-Werte, die je nach Instruktion anstelle eines Registers stehen können. Notiert werden diese mit einem $ davor und meistens als Hex-Wert mit 0x-Präfix, also z.B. $0x4. Dies bedeutet, dass statt des

Inhalts eines Registers einfach dieser konstante Wert genommen wird.

Wichtig beim Suchen nach **Dokumentation online** ist, dass diese möglicherweise Intel-Syntax nutzt oder Dinge abkürzt. Der Unterschied ist anhand der folgenden Instruktionsschreibweisen erkennbar:

```
1 addl $1, (%eax) # memory[%eax] := memory[%eax]+1 (32bit mode) **AT&T**
2 add DWORD PTR[eax], 1 # memory[%eax] := memory[%eax]+1 (32bit mode) **Intel
     **
```

Die Breitenangabe ist hier wichtig, da im Speicher ein 32bit-Wert und nicht bloss ein Byte um 1 erhöht wird. Sowohl AT&T-Syntax als auch Intel-Syntax lassen die Breitenangabe weg, wenn dies z.B. aus den Registernamen eindeutig ist:

```
1 mov %rsp, 4(%rax) # memory[%rax+4] := %rsp (64bit mode) **AT&T**
2 mov [rax+4], rsp # memory[%rax+4] := %rsp (64bit mode) **Intel**
```

Wie Sie sehen, fehlen in der **Intel-Syntax** die sogenannten *Sigils* $ und % sowie das Instruktionsbreitensuffix, die Memory-Schreibweise ist anders, und ***Source* und *Destination* sind vertauscht**. Intel-Syntax nutzt eine Breitenangabe als Teil des Operands bei Speicherzugriffen, wie *DWORD PTR* (double word pointer = 32bit). Ausserdem sollten Sie nach dem generellen Instruktionsnamen ohne Suffix suchen, also *mov* statt *movl*.

2.3. Hexadezimalwerte

Hexadezimalwerte sind **Zahlen zur Basis 16** statt zur üblichen Basis 10. Nebst den Ziffern von *0* bis *9* braucht es noch die Buchstaben von *a* bis *f*, um 16 verschiedene Werte darzustellen. Aus dem Sekundarschul-Unterricht sollte noch bekannt sein, wie man Zahlen Programmiersprachen zu anderer Basis interpretiert: Die Hexadezimalzahl 10_{16} entspricht der Dezimalzahl16_{10}. Vorteil von Hex-Zahlen ist, dass man mit genau zwei Stellen, also 00 bis FF, genau den Wertebereich eines Bytes (0 bis 255) abbilden kann. Ob Grossbuchstaben oder Kleinbuchstaben genommen werden, ist übrigens egal.

Übliche Schreibweise in Programmiersprachen und dergleichen für Hexwerte ist das Voranstellen von 0x vor einen Hexwert. Auch in diesem Text wird teilweise z.B. eine Speicheradresse als 0x2004 geschrieben, auch wenn in der Bildschirm-Ausgabe das 0x-Präfix fehlt. Damit ist eindeutig, dass es sich um eine Hex-Zahl handelt. Der entsprechende Dezimalwert 8196 wäre hier wenig hilfreich, da die verwendeten Tools Adressen als Hexwert darstellen.

2.4. C Standard Library

Die Programmiersprache C definiert eine Standardbibliothek, welche nützliche Funktionen bereitstellt. Diese wird C Standard Library genannt, oder kurz libc. Beim Kompilieren werden Verweise auf diese externen Funktionen ins Binary eingefügt. Im Disassembly werden Funktionsaufrufe zu Funktionen aus der libc folgendermassen dargestellt:

```
1 callq 1040 <fgets@plt>
```

Hier beispielhaft ein Aufruf zur Funktion fgets() aus der C Standard Library. Die Assembly-Instruktion callq ruft eine Funktion auf, danach folgt als Argument die Adresse in Hexadezimal-Schreibweise (ohne 0x-Präfix), und schliesslich folgt ein Adresshinweis: Adresse 1040 zeigt auf die Funktion fgets() in der PLT-Sektion.

Genauso wie für Unix-Befehle auf der Kommandozeile haben auch für Funktionen aus der libc jeweils eine Manpage zur Dokumentation. Diese befinden sich normalerweise in Sektion 3, für fgets() kann diese also mit folgendem Befehl angezeigt werden:

```
1 man 3 fgets
```

2.5. C-Strings und ASCII

In der Programmiersprache C sind Zeichenketten (Strings) äquivalent zu Byte-Arrays. Am Ende des Strings wird ein Null-Byte eingefügt, dieses wird auch *terminating zero* genannt. Die Länge ergibt sich implizit aus der Anzahl Bytes vor dem ersten Null-Byte (Zeichenketten dürfen kein Null-Byte enthalten), das Null-Byte selbst wird nicht mitgezählt.

Es wird nur die Speicheradresse des ersten Bytes im Programm referenziert (als char pointer oder in C-Syntax char *), aber weder die End-Adresse noch die Länge werden explizit gespeichert. Dies hat den Vorteil, dass mit nur einem zusätzlichen Byte (dem Null-Byte) ein String beliebiger Länge gespeichert werden kann. Häufiger Anfängerfehler beim Programmieren in C ist zu vergessen, ein zusätzliches Byte an Speicher für das Null-Byte zu reservieren.

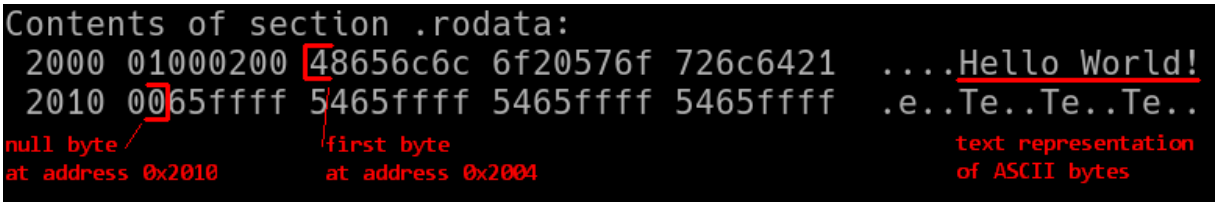


Abbildung 2: C-String innerhalb eines Hexdumps

In Abbildung 2 ist ab Adresse 0x2004 der String ”Hello World!” zu finden. Die Länge beträgt 12 Zeichen, danach folgt das Null-Byte an Adresse 0x2010.

Die Übersetzung zwischen Hexadezimalwert und Zeichen (character) folgt dem ASCII-Standard, die Werte lassen sich in einer ASCII-Tabelle nachschlagen, unter Anderem zu finden in der Manpage zu ascii(7) oder in Abbildung 5 im Anhang. Die Werte von 128 bis 255 sind übrigens nicht belegt, deshalb spricht man auch von *7-bit ASCII*, weil das oberste (achte) Bit immer 0 ist.

Unterschieden wird zwischen Zeichen, die sich am Bildschirm darstellen lassen, sogenannte *druckbare Zeichen* (printable characters), und non-printable characters. Der Bereich 0x20 bis 0x7E enthält die printable characters.

2.6. Executable and Linkable Format

Während ein ausführbares Programm direkt mit fertigen binären CPU-Instruktionen geschrieben werden *könnte*, ohne sich an Vorgaben für den Aufbau der Binärdatei zu halten, geben Betriebssysteme ein Format vor, an welches sich Programme halten müssen. Dabei werden Metadatenformat, Aufbau, Speicherstruktur, Zielinstruktionssatz, Startinstruktion und Position von Daten und Code definiert.

Für Unix-basierte Betriebssysteme hat sich das *Executable and Linkable Format* (ELF) durchgesetzt. Dessen grundlegende Struktur ist in Abbildung 3 veranschaulicht: Das Binary wird in *Sektionen* (sections) aufgeteilt, diese sind im Section Header am Ende des Binary verlinkt. Sections haben einen Namen, der mit einem Punkt beginnt, beispielsweise .dynamic oder .text. Die Sections enthalten verschiedene Teile eines Binaries. Wichtig sind .text, wo der eigentliche Maschinencode drinsteht, und .rodata (read-only data), wo Konstanten drin stehen, beispielsweise Zeichenketten (Strings).

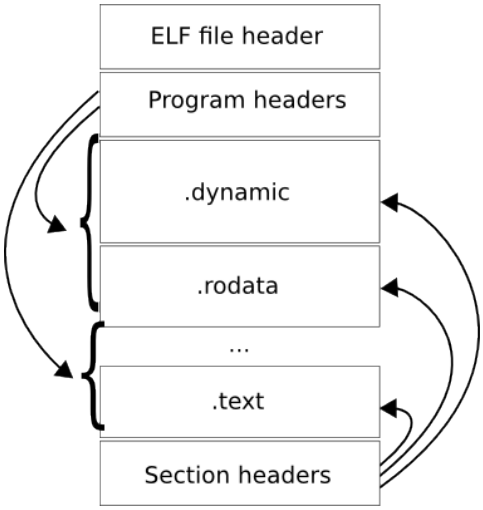


Abbildung 3: Sections im ELF-Binary

Ein Beispiel eines minimalen x86-Binaries ist in Abbildung 4 zu sehen. Dieses kommt komplett ohne Sections aus. Auf der linken Seite ist der tatsächliche Inhalt des Binaries als Hexdump; rechts werden die Bestandteile erklärt. Beispielsweise zeigt der Wert von e_entry im ELF-Header auf die Adresse der ersten CPU-Instruktion. Im schwarzen Fenster sieht man, was beim Ausführen in der Konsole passiert: Da das Programm selber keine Ausgabe produziert, wird der Rückgabewert im zweiten Befehl mit echo $? ausgegeben. Die Shell-Variable $? enthält jeweils den Rückgabewert des vorherigen Befehls.

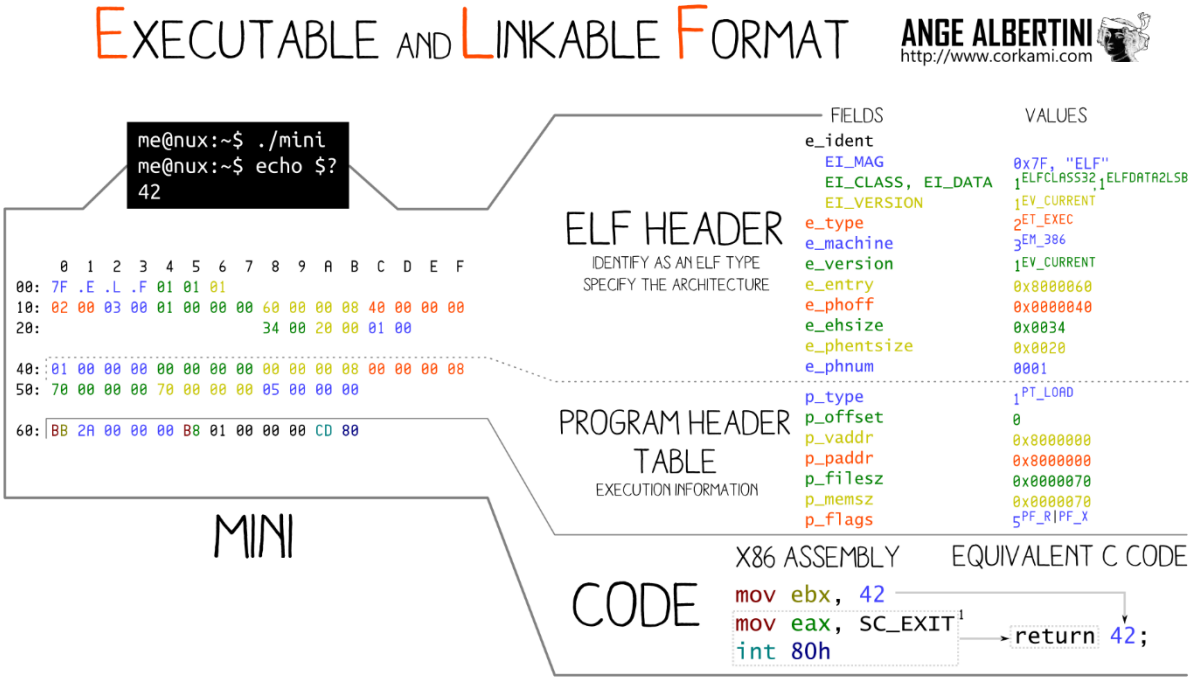


Abbildung 4: Minimales x86-Binary ohne Sections. CC-BY corkami.com

2.7. Calling Convention

In einem CPU-Instruktionssatz gibt es keine Funktionen wie in normalen Programmiersprachen. Stattdessen wird durch Änderung des Instruction Pointers (siehe Abschnitt 1.1.1) an eine bestimmte Stelle im Code gesprungen. Damit

Funktionsaufrufe abgebildet werden können, sind in der sogenannten *Calling Convention* festgelegt, welche Register eine aufgerufene Funktion verändern darf, wo die Funktionsparameter stehen, und wie der Stack aufgebaut ist.

Wichtig für den Laborversuch ist das *System V amd64 Application Binary Interface*, Standard unter Unix-ähnlichen Betriebssystemen: Das **erste Argument** einer Funktion befindet sich im Register %rdi, der **Rückgabewert** in %rax. Alle übrigen Details sind für diese Laboraufgabe nicht relevant.

2.8. Stack-Aufbau

Hier sehen Sie einen beispielhaften Aufbau eines Stacks zur Laufzeit eines Programms mit den Stack-Bereichen («frames») zweier Funktionen. Immer wichtig: Der Stack wächst nach unten, Adressen wachsen nach oben. Blau ist die aufgerufene Funktion.

Adresse	**Wert**	**Bedeutung**	**Pointer**
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	Voriger %rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	Voriger %rsp
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp	Aktueller %rbp
0x7f18	0x0000000000007f40	Gespeicherter Parameter %rdi	Zeiger auf String
0x7f10	0x0000000000000000	Lokale Variable aktuelle Funktion	Aktueller %rsp

In den folgenden Abschnitten ist Schritt für Schritt erklärt, was auf dem Stack passiert, wenn aus einer Funktion eine zweite Funktion aufgerufen wird.

2.8.1. Vor Funktionsaufruf

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	%rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	%rsp

Der *Base Pointer* zeigt jeweils auf das obere Ende vom Stackbereich der aktuellen Funktion, der *Stack Pointer* zeigt auf das untere Ende.

2.8.2. Funktionsaufruf

Die call-Instruktion macht folgendes:

1. Instruction Pointer (zeigt schon auf die nächste Instruktion) auf den Stack schreiben (push)
2. Ändere Instruction Pointer zu Startadresse der neuen Funktion

Push bedeutet, der Stack Pointer wird zuerst um ein 64bit-Wort (= 8 Bytes) erhöht, der Stack wächst also um 8 Byte nach unten. Dann wird der zu speichernde Wert in den Speicher geschrieben, und zwar an die Adresse, auf welche der Stack Pointer jetzt nach der Erhöhung hinzeigt.

Auf dem Stack liegt jetzt also zusätzlich die sogenannte **Rücksprungadresse**, also die Instruktion, die in der aktuellen Funktion der Call-Instruktion folgt.

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	%rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	%rsp

2.8.3. Funktions-Prolog 1: Alten Base Pointer speichern

Zu Beginn einer Funktion wird als erstes der Base Pointer auf den Stack gespeichert (push):

```
1 pushq %rbp
```

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	%rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	
0x7f20	0x0000000000007f48	Gespeicherter Basepointer	%rsp

2.8.4. Funktions-Prolog 2: Base Pointer auf Stack Pointer setzen

Danach wird der Base Pointer zum Stack-Anfang der neuen Funktion gesetzt, also da, wo aktuell der Stack Pointer ist.

```
1 movq %rsp,%rbp # %rbp := %rsp
```

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	
0x7f20	0x0000000000007f48	Gespeicherter Basepointer	%rbp, %rsp

2.8.5. Funktions-Prolog 3: Stack Pointer reduzieren

Danach wird der Stack-Bereich für die aktuelle Funktion reserviert, indem der Stack Pointer um die benötigte Grösse reduziert wird (Stack wächst nach unten):

```
1 sub $0x10,%rsp
```

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	
0x7f20	0x0000000000007f48	Gespeicherter Basepointer	%rbp
0x7f18	0x0000000000000000	Neu Reservierter Speicher	
0x7f10	0x0000000000000000	Neu Reservierter Speicher	%rsp

2.8.6. Funktions-Epilog 1: Stack Pointer erhöhen

Am Ende einer Funktion wird der Stack-Bereich wieder freigegeben, indem der Stack Pointer auf den Wert des aktuellen Base Pointers gesetzt wird:

```
1 movq %rbp,%rsp # %rsp := %rbp
```

Da der Stack Pointer während der laufenden Funktion weiter verändert werden kann, wird nicht einfach der gleiche Wert wie vorher wieder abgezogen. Der Base Pointer verändert sich nicht, referenziert also immer die korrekte Stelle. Auf diese Art muss in der laufenden Funktion nicht mitgezählt werden, um wie viel sich der Stack Pointer verändert hat.

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	
0x7f20	0x0000000000007f48	Gespeicherter Base Pointer %rbp, %rsp	%rbp, %rsp

2.8.7. Funktions-Epilog 2: Base Pointer wiederherstellen

Der auf dem Stack gespeicherte vorherige Wert des Base Pointers wird mit einer pop-Instruktion wiederhergestellt:

```
1 popq %rbp
```

Pop ist eine umgekehrte Push-Instruktion. Das bedeutet, dass zuerst der Wert auf dem Stack ins Register geladen wird und danach der Stack Pointer um die entsprechende Wortbreite erhöht (wiederum 8 Bytes).

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	%rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip	%rsp

2.8.8. Leave-Instruktion

Die beiden Schritte «mov, pop» des Funktionsepilogs werden üblicherweise in der leave-Instruktion zusammengefasst. Diese macht genau das gleiche wie die beiden Schritte des Epilogs in einer einzigen Instruktion:

```
1 leaveq
```

2.8.9. Funktionsrücksprung (return)

Zum Schluss wird der Instruction Pointer wiederhergestellt: Die ebenfalls auf dem Stack gespeicherte Rücksprungadresse vom Stack nach %rip gelesen und der Stack Pointer um eine Wortbreite erhöht, wie bei einer pop-Instruktion:

```
1 retq
```

Adresse	Wert	Bedeutung	Pointer
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion	%rbp
0x7f40	0x0000000000636261	String «abc»	
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion	
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion	%rsp

Somit sieht der Stack wieder so aus, wie er vor Funktionsaufruf war.

Falls die Funktion einen Rückgabewert hat, steht dieser nun im Register %rax, wie in Abschnitt 1.7 erwähnt.

3. Vorbereitung

3.1. Tasks

Zur Vorbereitung sind folgende Tasks im vorab zu erledigen.

- Lesen Sie den Theorieteil des vorherigen Kapitels

3.2. Benötigte Mittel

Für diese Vorbereitungsaufgabe benötigen Sie einen SSH-Zugang zur Laborübungs-VM.

Die VM verfügt nicht über eine grafische Oberfläche, daher lassen sich sämtliche Aktionen über die Kommandozeile durchführen.

4. Theorieverständnis

Nachdem Sie den Theorie-Teil gelesen haben, sollten Sie diese Fragen beantworten können.

- Beantworten Sie die Theoriefragen

Ein Hexdump enthält den folgenden Block: 61626300*. Handelt es sich dabei um einen gültigen C-String? Wenn ja, geben Sie Länge und Inhalt an, wenn nein, begründen Sie.*

Eine Funktion hat den Stackbereich 0x7f50 *bis* 0x7f80*. An welche Adresse zeigt der Base Pointer* %rbp*, an welche Adresse zeigt der Stack Pointer* %rsp*?*

Wenn diese Funktion eine zweite Funktion aufruft, an welcher Adresse zeigt dann der Base Pointer? An welcher Adresse ist die Rücksprungadresse auf dem Stack gespeichert?

5. Stack Buffer Overflow im Quellcode

Gegeben ist ein Programm sowohl in Quellcode (geschrieben in C) als auch kompiliert. Das kompilierte Binary gehört dem *root*-User und hat das *setuid*-Bit gesetzt. Dies bedeutet, dass das Programm von einem normalen Benutzer ausgeführt werden kann, dann aber root-Rechte besitzt.

Lassen Sie sich den Verzeichnisinhalt ausgeben und überprüfen Sie, ob die Datei tatsächlich dem Besitzer *root* und dem Setuid-Bit hat:

```
1 ls -l ~/vulnapp
```

Die korrekte Ausgabe sieht dann so aus:

```
1 -rwsr-xr-x 1 root root 19840 Sep 16 21:42 /home/labadmin/vulnapp
```

Das 's' an vierter Stelle steht für *setuid*.

Ziel der Laborübung ist es, eine Möglichkeit zu finden, mit dem Programm eine *root-Shell* zu bekommen, also eine Kommandozeile mit Superuser-Rechten. Dazu wird eine klassische *Stack Buffer Overflow*-Lücke ausgenützt.

5.1. Verständnis der Programmfunktionalität

Zu Beginn testen Sie aus, was das Programm eigentlich macht:

```
1 ./vulnapp
```

Wie Sie bemerken, verlangt das Programm einen Parameter auf der Kommandozeile (führen Sie dieses also mit einem beliebigen Parameter nochmals aus) und später eine Eingabe zur Laufzeit; beide werden vom Programm wieder ausgegeben.

5.2. Analyse des Quellcodes

Schauen Sie sich den Quellcode des Programms an, z.B. mit:

```
1 cat ~/vulnapp.c
```

In Zeile 9 sehen Sie die Array-Variable buf, die 16 Werte des Typs char (1 Byte) speichern kann. Danach wird eine Eingabe vom Benutzer angefragt und in buf eingelesen. Dies geschieht mittels der libc-Funktion scanf(). Diese ist anfällig für einen Buffer Overflow, Details erfahren Sie im nächsten Unterkapitel.

Der unsichere Code liegt innerhalb einer Funktion, die aus der main()-Funktion aufgerufen wird.

Wie heisst die Funktion, in der der unsichere Code liegt? Wie viele Argumente verlangt diese?

In Zeile 38 befindet sich ein Aufruf setuid(0). Dieser ändert den ausführenden Benutzer zu root, was nur dann tatsächlich funktioniert, wenn das Programm dem root-User gehört und das setuid-Bit besitzt.

Wird der Code in der oben gefundenen unsicheren Funktion vor oder nach dem Aufruf von setuid() *ausgeführt?*

5.3. scanf verstehen

Die libc-Funktion scanf() liest Werte aus der Benutzereingabe in Variablen ein. Dies ist beispielsweise nützlich, wenn Zahlen aus einem Text extrahiert und in numerischen Variablen gespeichert werden sollen.

Sie können die Dokumentation in der zugehörigen Manpage finden:

```
1 man 3 scanf
```

Zur Formatspezifikation %s findet sich folgendes:

Matches a sequence of non-white-space characters; the next pointer must be a pointer to character array that is long enough to hold the input sequence and the terminating null byte ('\0'), which is added automatically. The input string stops at white space or at the maximum field width, whichever occurs first.

Der scanf()-Aufruf im Code sieht so aus:

```
1 scanf("%s",buf);
```

Da keine *Maximum Field Width* angegeben ist (z.B. %15s statt %s), werden also so viele Zeichen eingelesen, bis das erste Whitespace-Zeichen auftaucht. Diese sind in der Manpage zu isspace() aufgelistet:

```
1 man 3 isspace
```

Wie Sie in der Laborübung sehen werden, können durch diese Funktion also mehr als die 16 Bytes in die Variable buf eingelesen werden. Diese Sicherheitslücke wird später in der Laborübung ausgenützt.

6. Stack-Aufbau analysieren

Um den Aufbau des Stacks zu sehen, müssen wir das Programm untersuchen, während dieses läuft. Dazu verwenden wir in der zweiten Laborübung einen Debugger, *gdb*.

Für diesen ersten Teil müssen Sie den Debugger noch nicht verwenden; die Ausgabe des Debuggers ist hier angegeben.

6.1. Stack anzeigen

Um herauszufinden, wie der Stack der main()-Funktion aussieht, benützt man in gdb folgenden Befehl:

```
1 x/7xg $rsp
```

Im Theorieteil der zweiten Laborübung wird die Verwendung von gdb erklärt, der Befehl hier dient nur als Referenz, Sie müssen diesen momentan nicht tatsächlich ausführen. Wichtig hier ist, dass der Befehl uns Quadwords, also 8-Byte-Werte, zurückgibt.

6.2. Quadwords in gdb

Die Ausgabe von gdb wird etwa so aussehen:

Adresse	Wert	Wert
0x7fffffffe450	0x00007fffffffe568	0x0000000200000000
0x7fffffffe460	0x0000000000000000	0x0000000200000000
0x7fffffffe470	0x656d75677241794d	0x000000000000746e
0x7fffffffe480	0x0000555555555200	

Links die Adresse, danach in der Mitte der 8-Byte-Wert (*Quadword*) an dieser Adresse, dann mit etwas Abstand rechts der nächste 8-Byte-Wert. Die **Adresse des rechten Werts ist um 8 höher**, in der ersten Zeile beispielsweise 0x7fffffffe458:

```
0x7fffffffe450: 0x00007fffffffe568      0x0000000200000000
                QWord an Adresse …e450   QWord an Adresse …e458
```

Abbildung 5: Positionen der QWords

Hinweis

Verwirrungsgefahr: GDB zeigt mit dem x-Befehl die Adressen ab der gegebenen Adresse *und höher* an. Da der Stack *nach unten* wächst, sehen Sie die Werte *nach oben*, also «rückwärts» im Stack.
Innerhalb eines 8-Byte-Werts sind die Bytes nochmals rückwärts, die Adresse zeigt also exakt auf das Byte ganz rechts im 8-Byte-Wert, dann folgt das Byte links davon usw. Nochmals erklärt in Abschnitt 5.4.

Hinweis

Warum Quadwords?
Wir arbeiten auf 64bit-Systemen. Adressen und Werte sind grundsätzlich immer 64bit = 8 Bytes = 1 Quadword.

6.3. Adress-Reihenfolge Stack vs. gdb

Werte im Stack-Diagramm schreiben wir von der höchsten zur tiefsten Adresse. In gdb bekommen wir die Werte von der tiefsten zur höchsten Adresse ausgegeben.

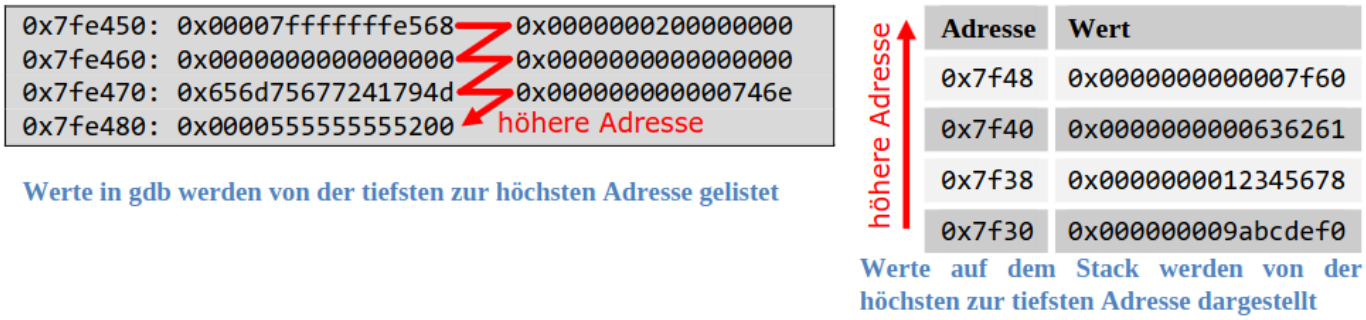


Abbildung 6: Reichenfolge der Adressen

Werte auf dem Stack werden von der höchsten zur tiefsten Adresse dargestellt

6.4. Bytes innerhalb von Quadwords

Die Ausgabe von gdb zeigt uns Quadword-Werte im Speicher an. Adressen zeigen aber immer auf ein einzelnes Byte: Das erste Byte in einem Wert.

Abbildung 7: Adresse des ersten Bytes im ersten Quadword

Wo ist nun das erste Byte in einem Quadword? Dazu müssen wir wissen, dass Prozessoren aus der x86-Familie (also auch amd64) eine Little-Endian-Architektur haben.

6.4.1. Little Endian

Als *Endianness* bezeichnet man die Reihenfolge von Bytes innerhalb eines Wortes auf einer Prozessor-Architektur. Die gängigen Prozessoren in PCs von Intel und AMD basieren auf x86, einer Little-Endian-Architektur.

Little Endian bedeutet «Begins with Little End». Merke: **Little End First**. Das **erste** Byte in einem Quadword ist also das **kleinste** Byte. Wie in einer Dezimalzahl steht auch in einer Hexadezimalzahl die kleinste Stelle ganz rechts. Zur Wiederholung: Ein Byte hat genau **zwei** Hexadezimal-Stellen.

```
0x7fffffffe450: 0x00007fffffffe568      0x0000000200000000
                erstes Byte = Byte an der Adresse 0x7fffffffe450
```

Abbildung 8: Erstes Byte

Das erste Byte hier hat also den Wert 0x68.

Das zweite Byte ist an der nächsthöheren Adresse, also 0x7fffffffe451:

```
0x7fffffffe450: 0x00007fffffffe568      0x0000000200000000
                zweites Byte = Byte an der Adresse 0x7fffffffe451
```

Abbildung 9: Zweites Byte

Die Bytes innerhalb eines Quadwords sind also tatsächlich «rückwärts» im Speicher. Wenn Sie gdb die Bytes einzeln ausgeben lassen statt als Quadwords, sehen Sie die Reihenfolge andersrum. Acht einzelne Bytes ab der Adresse 0x7fffffffe450, also bis Byte 0x7fffffffe457:

```
0x7fffffffe450: 0x68 0xe5 0xff 0xff 0xff 0x7f 0x00 0x00
```

Abbildung 10: Bytes-Reihenfolge

6.5. Verständnisaufgabe

Hier ist nochmals die Ausgabe von gdb aus Abschnitt 5.2:

Adresse	Wert	Wert
0x7fffffffe450	0x00007fffffffe568	0x0000000200000000
0x7fffffffe460	0x0000000000000000	0x0000000200000000
0x7fffffffe470	0x656d75677241794d	0x000000000000746e
0x7fffffffe480	0x0000555555555200	

Als Verständnisaufgabe sollen Sie in der Beispiel-Ausgabe oben die Werte dreier **einzelner Bytes** finden; also nicht ganze 64bit-Quadwords.

Welcher hexadezimale Wert steht in **Byte*** 0x7fffffffe470?Welcher in Byte* 0x7fffffffe471*? Welcher in Byte* 0x7fffffffe45c*?

6.6. Aufbau des Stacks in Tabelle

Wie erwähnt, ist die Ausgabe von gdb «falsch herum» und in zwei Spalten. Zur Veranschaulichung des Stack-Aufbaus übertragen Sie die Werte auf dem Stack in eine Tabelle. Beachten Sie die Adressen in der linken Spalte!

Übertragen Sie die Werte auf dem Stack aus der Beispiel-Ausgabe von gdb in die Tabelle. Vorsicht: In der Tabelle ist der Stack «richtig herum» von oben nach unten aufgebaut.

Address	Wert	Pointer
0x7fffffffe480		%rbp
0x7fffffffe470		
0x7fffffffe468		
0x7fffffffe460		
0x7fffffffe458		
0x7fffffffe450		%rsp

7. Anhang

7.1. ASCII-Tabelle

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Abbildung 11: ASCII-Tabelle (CC-0)

Notizen