Umgang mit Buffern

— Seminararbeit —

Seminar 'Effiziente Programmierung in C'

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Stefan Bruhns
E-Mail-Adresse:	2bruhns@informatik.uni-hamburg.de
Matrikelnummer:	6453116
Studiengang:	Bachelor of Science Informatik

Betreuer: Michael Kuhn

Hamburg, den 29.12.2013

Abstract

Diese Arbeit beschäftigt sich mit den Umgang mit Buffern in der Programmiersprache C. In ihr werden verschiedene Möglichkeiten der Allozierung von Bufferspeicher vorgestellt, welche in Bezug auf Performance und Verwendung verglichen werden.
Des Weiteren werden durch den falschen Umgang mit Buffern verursachte Sicherheitslücken vorgestellt und mögliche Maßnahmen zur Vermeidung solcher Fehler aufgezeigt.

Inhaltsverzeichnis

1 Einleitung

Buffer im Kontext der C-Programmierung.

Der Begriff Buffer wird in vielen Bereichen verwendet. In der Chemie bezeichnet beispielsweise der Begriff Buffer (Puffer) ein Stoffgemisch, bei dem sich der ph-Wert durch Zugabe einer Säure oder Base deutlich schwächer verändert als bei einer normalen Lösung. Aus diesem Grund ist es im Folgenden wichtig, den Begriff näher zu definieren.

1.1 Buffer in der C-Programmierung

In der Informatik wird unter einem Buffer ein Speicher zur Zwischenspeicherung von Daten verstanden. Wie dieser aufgebaut und auf welchen physikalischen Medien (Festplatte, Hauptspeicher, u.s.w.) er sich befindet, hängt dabei wiederum stark vom Kontext ab. Im Kontext der C-Programmierung und somit auch in dieser Arbeit, wird darunter ein reservierter Speicherbereich im Hauptspeicher verstanden, der zur Zwischenspeicherung Daten gleichen Typs dient.

1.2 Verwendung von Buffern

Buffer werden verwendet um Geschwindigkeitsunterschiede in der Verarbeitung auszugleichen bzw. zu dämpfen. Dies ist meist beim Zugriff auf externe Ressourcen der Fall. So spielen Buffer bei fast allen File I/O-, Netzwerk- und Hardware-Zugriffen eine entscheidende Rolle. Bei der Arbeit mit Zeichenketten und der Nutzung, der von der Standartbibliothek zur Verfügung gestellten String-Funktonen, werden ebenfalls Buffer verwendet. (vergleiche [Pro13])

2 Speichersegmente und Buffer

Buffer in den verschieden Speichersegmenten

Buffer-Speicher kann in den unterschiedlichen Speicherbereichen eines C-Programms reserviert und verwaltet werden. Buffer können zum Beispiel statisch oder dynamisch alloziert werden. Innerhalb dieser Kategorien können weitere Unterscheidungen vorgenommen werden. Um die daraus entstehenden Konsequenzen zu verstehen und die richtige Art von Buffer für einen konkreten Anwendungsfall auszuwählen, sind Kenntnisse über den Aufbau eines C-Programms im Speicher notwendig.

2.1 Speicheraufbau eines C-Programms

Die Repräsentation eines C-Programms im Speicher lässt sich typischerweise in fünf Segmente unterteilen (siehe Abbildung 2.1). Der konkrete Speicher-Aufbau kann je nach Rechnerarchitektur und konkreten Programm variieren. Deswegen soll hier ein typisches Beispiel vorgestellt werden.

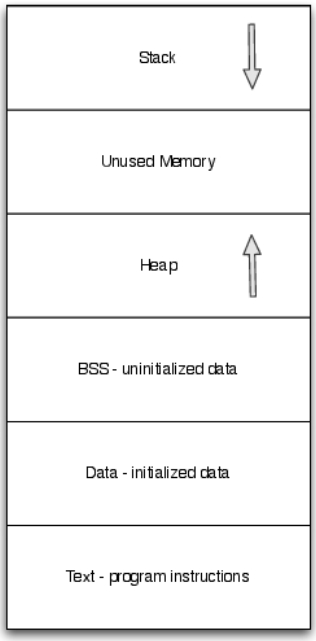

Abbildung 2.1: Speichlayout Quelle: [WRS05]

Text-Segment: Das Textsegment befindet sich an den niedrigsten Speicheradressen. In diesem Bereich wird der ausführbare Programm-Code abgelegt und der aktuelle Befehl über den Instruction-Pointer referenziert. Dieses Segment ist für die Reservierung von Buffer-Speicher irrelevant und soll nur wegen der Vollständigkeit erwähnt werden.

BSS- und Daten-Segment: Hinter dem Text-Segment mit zunehmenden Speicher-Adressen folgen das Daten- und BSS-Segment. In beiden Segmenten werden alle globalen, als static deklarierten Variablen angelegt. Initialisierte Variablen werden im Daten und nicht initialisierte im BSS Segment alloziert.

Heap: Das Heap-Segment wird für die dynamisch allozierten Daten verwendet. Die Verwaltung des Heaps wird durch das Betriebssystem übernommen und kann über malloc(), calloc() und deren Derivate genutzt werden. Das Heapsegment wächst traditionell zu höheren Speicheradressen dem Stack entgegen. Ein Heap-Speicher steht theoretisch unbegrenzt zur Verfügung; praktisch ist jedoch kein physikalischer Speicher mehr vorhanden.

Stack: Der Stack ist ein LIFO-Speicher, der an der höchsten Speicheradressen beginnt und dem Heap entgegenwächst. In diesem Segment befinden sich die Rücksprungadressen der Funktionsaufrufe, die Funktionparameter und lokale Variablen (Stackframe). Bei jedem Funktionsaufruf wird ein entsprechender Stackframe angelegt und nach Verlassen der Funktion wieder abgebaut(siehe Abildung 2.2). Adressiert wird der Stack über den Stackpointer, der immer auf das zuletzt hinzugefügte Datum zeigt.

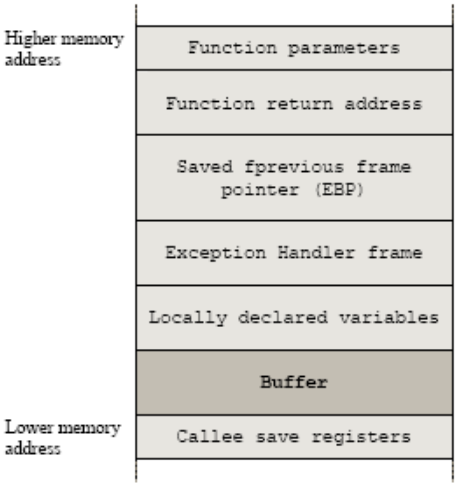

Abbildung 2.2: Aufbau Stackframe Quelle: [STA14]

2.2 Statische Buffer

Statische Buffer können durch eine Deklaration im globalen Scope oder durch das Schlüsselwort static angelegt werden (siehe Listing 2.1). Der allozierte Speicher wird im BSS oder Datensegment des Programms erzeugt. Im Gegensatz zu dynamischen Buffern muss die Größe des benötigten Speicherbereichs bereits zur Compile-Zeit feststehen. Der Vorteil auf diese Weise einen Buffer zu allozieren ist, dass die Laufzeit des Programms nicht beeinflusst wird, da dieser Speicher bereits beim Laden des Programms angelegt wird. Der Gültigkeitsbereich eines solchen Buffers ist global und der reservierte Speicher wird erst mit Beenden des Programms durch das Betriebssystem wieder frei gegeben.

Listing 2.1: Reservieren von statischem Buffer-Speicher

```
1
2  #define BUFFER_SIZE 100
3
4  // uninitalize -> BSS-Segment
5  char static_buffer_first[BUFFER_SIZE];
6
7  // initalize -> DATA-Segment
8  char static_buffer_init[BUFFER_SIZE] = { 1 }
9
10 int main (int argc, char *argv[])
11 {
12   // attantion: scope is also global
13   static char static_buffer_sec[BUFFER_SIZE];
14
15   ...
16 }
```

2.3 Dynamische Buffer: Heap vs.Stack

Dynamische Buffer sind für die folgenden Betrachtungen deutlich interessanter. Hier wird der Buffer-Speicher zur Laufzeit reserviert und freigegeben. Es sollen zwei Arten von dynamischen Speichern betrachtet werden. Auf der einen Seite der Buffer-Speicher, der im Heap-Segment des Programmspeichers angelegt wird. Auf der anderen der, der sich im Stack-Segment des Programms befindet. Diese beiden Arten von Buffern unterscheiden sich auf Grund des Speicherbereichs, in dem sie sich befinden, in ihren Eigenschaften (z.B. in der maximalen Größe, in der Performance und im Gültigkeitsbereich). Auf Grundlage dieser Unterschiede sollte die Wahl des Buffertyps dem jeweiligen Problem angepasst werden. Im Folgenden werden die Eigenschaften von Heap- und Stack-Buffern etwas eingehender vorgestellt und verglichen.

2.3.1 Buffer im Stack-Segment

Wie in dem Abschnitt 2.1 bereits angesprochen, werden die lokalen Variablen einer Funktion im Stack-Segment angelegt. Daraus ergibt sich, dass es für die Allokation von Stack-Speicher ausreicht, eine lokale Variable zu deklarieren (siehe Listing 2.2). Da der Stackframe für eine Funktion bei ihrem Aufruf angelegt und nach Verlassen der Funktion wieder abgebaut wird, ist der Gültigkeitsbereich (Scope) nur lokal innerhalb der Funktion gegeben. Dies bedeutet gleichzeitig, dass beim Verlassen der Funktion der reservierte Speicher automatisch wieder frei gegeben wird und sich der Programmierer nicht selbst um die Freigabe des Speichers kümmern muss, wodurch keine Speicherleaks auftreten können. Da zur Reservierung von Stack-Speicher nichts weiter nötig ist, als eine Subtraktion des Stack-Pointers, ist diese Art der Allokation von Speicher sehr effizient. Der Nachteil beim Anlegen von Buffern auf dem Stack liegt darin, dass das Stack-Segment eine begrenzte Größe hat. Die Größe des verfügbaren Stack-Speichers ist vom verwendeten Betriebssystem und Compiler abhängig und liegt in der Regel standardmäßig zwischen 1-4 MB. Bei einigen Compilern ist es möglich, mittels bestimmter Flags die Größe des Stack-Segments zu beeinflussen oder unter Linux im Programm selbst mittels der Funktion setrlimit() aus sys/resource.h (siehe *man-page setrlimit*).
Auf Grund der vorgestellten Eigenschafen, wie dem lokalen Gültigkeitsbereich, der einfachen Nutzung, der guten Performance und der maximalen Größenbegrenzung des Stack-Segments, eignet sich diese Form von Buffer-Speicher besonders gut für Anwendungsfälle, in denen häufig kleine und kurzlebige Zwischenspeicher benötigt werden.

Listing 2.2: Reservieren von Buffer-Speicher auf dem Stack

```
1
2  #define BUFFER_SIZE 100
3
4  void buffer()
5  {
6    // buffer placed on stack only avalible in this function
7    char stack_buffer[BUFFER_SIZE];
8  }
9
10 int main (int argc, char *argv[])
11 {
12   // buffer placed on stack segment scope-> locale(main)
13   char stack_buffer[BUFFER_SIZE];
14
15   ...
16 }
```

2.3.2 Buffer im Heap-Segment

Um Buffer-Speicher auf dem Heap zu allozieren werden die, von der C-Bibliothek zur Verfügung gestellten, Funktionen malloc() und deren Derivate(calloc ...) genutzt. Der so reservierte Speicher muss durch den Programmierer selbst wieder frei gegeben werden und ist damit von der Erzeugung bis zur Freigabe durch den Aufruf der free-Funktion gültig (siehe Listing 2.3).
Der Heap-Speicher wird durch das Betriebssystem verwaltet und ist theoretisch unbegrenzt (außer es ist kein physischer Speicher mehr vorhanden). Da der Aufruf von malloc auch einem Systemcall darstellt, ist diese Art der Speicherreservierung deutlich langsamer, als bei der im vorherigem Kapitel vorgestellten Allokation von Stack-Speicher. Durch die Möglichkeit, beliebig viel Speicher zu reservieren und einer selbst festgelegen Gültigkeitsdauer, eignen sich Heap-Buffer für große Datenmengen und für eine Verwendung über Funktionsgrenzen hinaus.

Listing 2.3: Reservieren von Buffer-Speicher auf dem Heap

```
1
2  include <stdlib.h>
3
4  #define BUFFER_SIZE 100
5
6  char *create_buffer()
7  {
8    // buffer placed on heap and return a reference to it
9    char *buffer = malloc(sizeof(char) * BUFFER_SIZE);
10   return buffer;
11 }
12
13 void destroy_buffer(char *)
14 {
15   free(buffer);
16 }
17
18 int main (int argc, char *argv[])
19 {
20   char *buffer = create_buffer();
21   ...
22   // do something
23   ...
24   destroy_buffer(buffer);
25 }
```

2.3.3 Performance

Wie bereits zuvor angesprochen, gibt es bei der Allokation von Buffer-Speicher Performanceunterschiede zwischen der Allokation von Heap- beziehungsweise Stack-Speicher. Im Rahmen dieser Arbeit wurde hierzu eine Messung durchgeführt, welche die bereits getroffenen Aussagen belegen soll. Durchgeführt wurden diese Messungen auf den AMD-Knoten des Lehre-Clusters des Fachbereichs Arbeitsbereich Wissenschaftliches Rechnen. Der für die Messung verwendete Code steht auf der Kurswebsite[1] des Arbeitsbereichs Wissenschaftliches Rechnen zur Verfügung.

Ergebnisse der Messung:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 36.30      4.25     4.25 1000000000     4.25     4.25  heap_buffer
 26.59      7.37     3.11                             main
 25.47     10.35     2.98 1000000000     2.98     2.98  stack_buffer
```

Die Ergebnisse zeigen, dass in diesem Beispiel die Reservierung von Speicher auf dem Heap 1,5 mal so lange dauert wie die Reservierung von Stack-Speicher. Da bei diesem Versuch 1000000000 mal die gleiche Buffergröße reserviert und dann wieder freigegeben worden ist, kann davon ausgegangen werden, dass die Reservierung von Heap-Speicher in realen Anwendungen noch deutlich langsamer sein wird (da hier davon auszugehen ist, dass immer wieder das gleiche Stück Speicher zurückgeben wird). Alles in allem untermauern die Ergebnisse dieses Laufzeittests die Behauptung, dass Heap-Allokation deutlich mehr Zeit kosten als ihr jeweiliges Stack-Gegenstück.

[1] Link zum Code: http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2013_2014/epc-1314-bruhns-buffer-materialien.tar.gz

3 Sicherheit

Sicherheitsaspekte, die beim Umgang mit Buffern beachtet werden sollten

Die meisten Sicherheitslücken und Fehler in C-Programmen sind auf einen falschen bzw. unvorsichtigen Umgang mit Buffern zurückzuführen. In diesem Kapitel sollen einige typische Fehler und auch Möglichkeiten, diese zu vermeiden und rechtzeitig zu erkennen, vorgestellt werden.

3.1 Memory Leaks

Memory Leaks können bei der dynamischen Allokation von Speicher im Heap-Segment mittels der Funktionen malloc entstehen. Dies geschieht, wenn der so reservierte Speicher nicht korrekt wieder freigegeben wird.
Bei kleinen Programmen, die nur einmal zum Beispiel in der main Funktion Speicher reservieren, führt dies meist nicht zu Problemen, da beim Verlassen des Programms der gesamte reservierte Speicher an das Betriebssystem zurück gegeben wird. Wird allerdings in einem Programm beispielsweise Speicher innerhalb einer Schleife reserviert und nicht wieder korrekt freigeben, führt dies zu dazu, dass das Programm immer mehr Speicher beansprucht. Dies kann soweit gehen, bis kein physischer Speicher mehr zur Verfügung steht.

Listing 3.1: Memory Leak

```
1  #include <stdlib.h>
2
3  #define BUFFER_SIZE 10000
4
5  int main(void) {
6     char *buffer;
7     int i = 0;
8     while(i < 1000) {
9        buffer = malloc(sizeof(char) * BUFFER_SIZE);
10       // do something with the buffer
11       i++;
12    }
13    free(buffer);
14    return EXIT_SUCCESS;
15 }
```

Im Listing 3.1 wird dieses Problem deutlich. Innerhalb der Schleife wird bei jedem Schleifendurchlauf Speicher reserviert, der allerdings nicht innerhalb der Schleife wieder freigegeben wird. Der free()-Aufruf außerhalb der Schleife gibt nur den, im letzten Schleifendurchlauf reservierten, Speicher wieder frei.
Solche Fehler in einem fertigen Programm zu finden, stellt sich oft als nicht triviale Aufgabe dar. Zum einen gäbe es die Möglichkeit, dies über diverse Makros zu versuchen, bei denen bei jedem malloc-Aufruf ein Zähler hoch gezählt wird und ein Logeintrag erstellt wird. Bei diesem Verfahren würde man bei jedem free-Aufruf den Zähler vermindern und ebenfalls einem Logeintrag erzeugen. Dieses Verfahren wird von Jürgen Wolf in seinem Buch *C von A bis Z* vorgestellt (vergleiche [Wol09]).

Listing 3.2: Debug Makros aus C von A bis Z (siehe [Wol09])

```
1  /* mem_check.h */
2  #ifndef MEM_CHECK_H
3  #define MEM_CHECK_H
4  #define DEBUG_FILE "Debug"
5
6  static int count_malloc=0;
7  static int count_free  =0;
8  FILE *f;
9
10 #define malloc(size) \
11         malloc(size);\
12         printf("malloc in Zeile %d der Datei %s (%d Bytes)
               ↪ \n"\
13         ,__LINE__,__FILE__, size);\
14         count_malloc++;
15
16 #define free(x)\
17         free(x); \
18         x=NULL;\
19         printf("free in Zeile %d der Datei %s\n",
20                __LINE__,__FILE__);\
21         count_free++;
22
23 #define return EXIT_SUCCESS; \
24         f=fopen(DEBUG_FILE, "w");\
25         fprintf(f, "Anzahl malloc : %d\n",count_malloc);\
26         fprintf(f, "Anzahl free   : %d\n",count_free);\
27         fclose(f);\
28         printf("Datei : %s erstellt\n", DEBUG_FILE);\
29         return EXIT_SUCCESS;
30
31 #endif
```

Eine weitere Möglichkeit Memory Leaks aufzuspüren, bietet das Tool valgrind welches mit der Option –leak-check auch Speicherlecks erkennen kann. Führt man das Listing 3.1 mit valgrind aus, erhält man Ausgabe 3.3, der entnommen werden kann, dass 999 reservierte Blöcke nicht wieder freigegeben wurden.

Listing 3.3: Valgrind Leak Check

```
1  ~ $ valgrind --leak-check=yes ./mem_leak
2  ==19331== Memcheck, a memory error detector
3  ==19331== Copyright (C) 2002-2013, and GNU GPL'd, by Julian
     ↪ Seward et al.
4  ==19331== Using Valgrind-3.9.0 and LibVEX; rerun with -h
     ↪ for copyright info
5  ==19331== Command: ./mem_leak
6  ==19331==
7  ==19331==
8  ==19331== HEAP SUMMARY:
9  ==19331==     in use at exit: 9,990,000 bytes in 999 blocks
10 ==19331==   total heap usage: 1,000 allocs, 1 frees,
     ↪ 10,000,000 bytes allocated
11 ==19331==
12 ==19331== 9,990,000 bytes in 999 blocks are definitely lost
     ↪ in loss record 1 of 1
13 ==19331==    at 0x4C28730: malloc (in
     ↪ /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
14 ==19331==    by 0x400557: main (test.c:9)
15 ==19331==
16 ==19331== LEAK SUMMARY:
17 ==19331==    definitely lost: 9,990,000 bytes in 999 blocks
18 ==19331==    indirectly lost: 0 bytes in 0 blocks
19 ==19331==      possibly lost: 0 bytes in 0 blocks
20 ==19331==    still reachable: 0 bytes in 0 blocks
21 ==19331==         suppressed: 0 bytes in 0 blocks
22 ==19331==
23 ==19331== For counts of detected and suppressed errors,
     ↪ rerun with: -v
24 ==19331== ERROR SUMMARY: 1 errors from 1 contexts
     ↪ (suppressed: 1 from 1)
```

3.2 Buffer Overflows

Buffer Overflows sind die wohl am meisten ausgenutzten Sicherheitslücken in C-Programmen. Von einem Buffer Overflow spricht man, wenn versucht wird mehr Daten in einen reservierten Speicherbereich zu schreiben als dieser groß ist. Im besten Fall resultiert dies in

einem Segmentation Fault, da das Programm keinen Zugriff auf den nicht reservierten Speicher hat. Im schlimmsten Fall ist es allerdings möglich, einen Buffer Overflow zu nutzen, um den Programmablauf oder Programmdaten zu manipulieren. Buffer Overflows können sowohl auf dem Heap als auch auf dem Stack auftreten. Die Abbildung 2.1 zeigt, wie ein Buffer Overflow auf dem Stacksegment dazu ausgenutzt werden kann, um die Rücksprungadresse gezielt zu manipulieren und es so möglich wird zu einer beliebigen Stelle im Code zu springen. Dies könnte auch ein, über eine Buffer Overflow Lücke eingeschleuster, Angriffscode sein. In den bereits erwähnten Codebeispielen zu dieser Arbeit befindet sich ein solcher Buffer Overflow, der während des dazugehörigen Vortrags vorgeführt wurde.

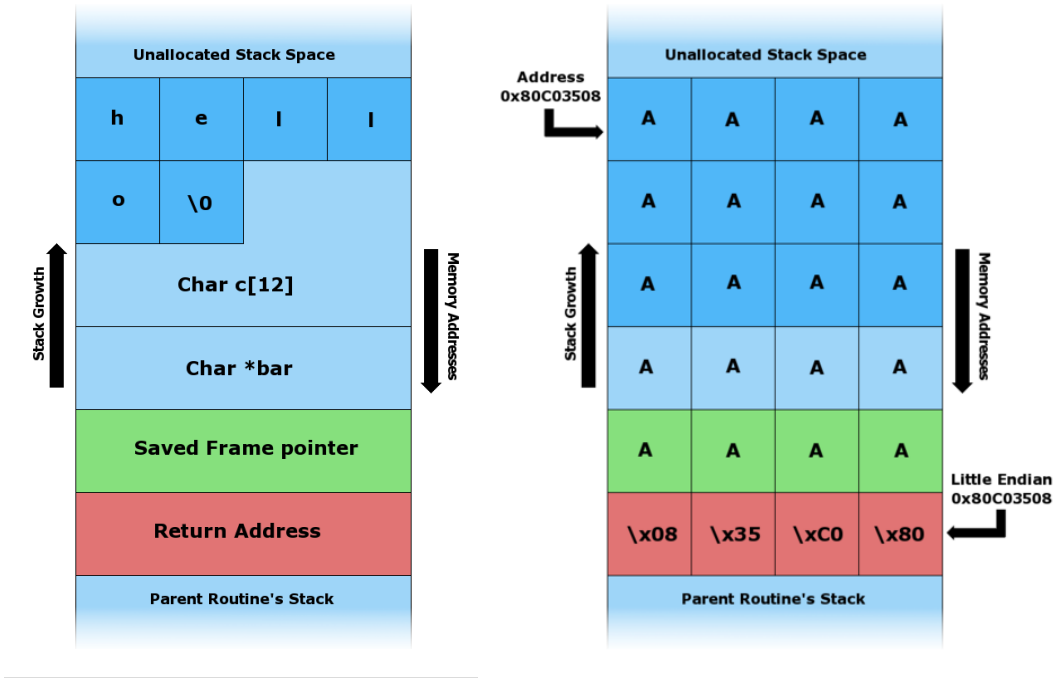

Abbildung 3.1: Stackoverflow Quelle: [WIK14]

Buffer Overflows entstehen, wenn die vorherige Überprüfung der Größe des Buffers und die der zu schreibenden Daten nicht korrekt durchgeführt wird. Oft wird die Größe nicht über Makros definiert, sondern als Zahl, welche bei der Allokation und bei der Prüfung angegeben wird. Ändert man nun die Größe des Buffers, wird oft vergessen die Größenprüfungen anzupassen. Hier sollten am besten Makros und sizeof() verwendet werden (siehe Tabelle 3.1). Auch bei der Arbeit mit Strings wird oft vergessen, dass ein C-String null terminiert ist und so der String 'Hallo' nicht 5 sondern 6 Byte groß ist. Des Weiteren sollte bei der Arbeit mit Strings darauf geachtet werden, dass anstelle der Standard-Stringfunktionen der C-Bibliothek, das jeweilige Gegenstück mit Größenprüfung eingesetzt wird (siehe Tabelle 3.2).

Tabelle 3.1: korrekte Größenprüfungen (vergleiche: [Inc12])

So nicht!	Lieber so!

`char buffer[100];`
 	
`if(size_of_data < 100){`
`  // write data`
`}`
	`#define BUFFER_SIZE 100`
	`char buffer[BUFFER_SIZE];`
	
	`if(size_of_data < BUFFER_SIZE){`
	`  // write data`
	`}`

`char buffer[100];`
 	
`if(size_of_data < 100){`
`  // write data`
`}`
	`char buffer[100];`
	
	`if(size_of_data < sizeof(buffer)){`
	`  // write data`
	`}`

Tabelle 3.2: Stringfunktionen

Unsichere Funktion	Gegenmaßnahme
`scanf("%s",str);`	`scanf("%10s",str);`
`gets(puffer);`	`fgets(puffer, MAX\_PUFFER, stdin);`
`strcpy(buf1, buf2);`	`strncpy(buf1, buf2, SIZE);`
`strcat(buf1 , buf2);`	`strncat(buf1, buf2, SIZE);`
`sprintf(buf, "%s", temp);`	`snprintf(buf, 100, "%s", temp);`

Auch nach der Programmerstellung gibt es Möglichkeiten, die Ausnutzbarkeit von Buffer Overflows zu senken. So bieten fast alle modernen Betriebssysteme die Möglichkeit der Address Space Layout Randomization. Hierbei wird der Adressraum eines Programms zufällig ausgewürfelt und es wird einem Angreifer so erschwert, die Adressen von möglichem Schadcode zu berechnen. Dieses Verfahren kann allerdings auch durch Sprayingtechniken umgangen werden. Auch Compiler bieten Möglichkeiten, Stack basierte Overflows schwerer ausnutzbar zu machen. Hier bietet es sich an, zum Beispiel die Möglichkeit gcc mit den Prameter -fstack-protector* aufzurufen.

3.3 Indexfehler und Integer Overflows

Auch durch die Nutzung falscher Indexe ist es möglich auf nicht reservierte Speicherbereiche zuzugreifen. Besonders gefährlich wird dies, wenn die Indexe oder Buffergrößen durch Nutzereingaben berechnet werden. Stellen wir uns ein zweidimensionales Array als Buffer vor, für das n * m Bytes Speicher reserviert werden sollen. Der Nutzer soll nun die Möglichkeit haben, durch die Eingabe von n und m die Größe des Feldes festzulegen. Ist nun n*m Größer als die maximale Größe eines Integer, wird bei der Allokation nicht die erwartete Menge Speicher reserviert. Ein naiver Ansatz um zu prüfen, ob n*m zu keinem Integer Overflow führt, könnte wie im Listing 3.4 aussehen. Diese Art der Prüfung wirkt auf den ersten Blick plausibel, allerdings kann der C Compiler solcher If-Anweisungen weg optimieren (siehe [Inc12]). Aus diesem Grunde sollte die Prüfung wie in Listing 3.5 durchgeführt werden. Es sollte also direkt auf einen Overflow vor der Berechnung der Größe geprüft werden.

Listing 3.4: Interger Overflow Falsche Prüfung

```
1 int buffer_size = n*m;
2
3 if(n > buffer_size || m > buffer_size) {
4   ...
5 }
```

Listing 3.5: Interger Overflow Prüfung

```
1 if (n > 0 && m > 0 && INT_MAX/n >= m) {
2
3     int bytes = n * m;
4     ...
5
6 }
```

4 Fazit

Alles in allem lässt sich sagen, dass man beim Einsatz von Buffern vorher überlegen sollte, welche Art von Buffer für den jeweiligen Anwendungsfall angebracht ist. Die Allokation von Stack-Speicher ist deutlich schneller als die von Heap-Speicher. Im Gegensatz dazu stehen die begrenzte Größe des Stack-Segments und der Fakt, dass Fehler die zu Overflows führen können auf dem Stack meistens deutlich einfacher für Angreifer auszunutzen sind. Des Weiteren sollte man immer Hinterkopf haben, besonders bei Eingaben die in einen Buffer geschrieben werden sollen, durch korrekte Größenprüfung sicher zu stellen, dass es nicht zu Overflows kommt. Hierbei können Makros für die Größe des Buffers helfen. Auch sollte man bei der Berechnung von Buffergrößen oder Indexes immer beachten, dass diese möglicherweise zu Integer Overflows führen können. Was wiederum Buffer Overflows oder Indexfehler nach sich ziehen kann. Das fertige Programm mit Tools wie valgrind auf Speicherleaks und ähnliches zu testen ist generell sinnvoll und nützlich.

Literaturverzeichnis

[Inc12] Apple Inc. Secure coding guide, Juni 2012.

[Pro13] The Linux Information Project. Buffer definition, Oktober 2013.

[STA14] www.tenouk.com, 2014.

[WIK14] en.wikipedia.org buffer overflow, 2014.

[Wol09] Jürgen Wolf. C von a bis z, 2009.

[WRS05] Stephen A. Rago W. Richard Stevens. Advanced programming in the unix® environment: Second edition, 2005.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis