Umgang mit Buffern

Stefan Bruhns

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

2013-11-14

Gliederung (Agenda)

Code-Beispiele

- Quellcode
  - Download: ...
  - C99 Standard
  - inklusive Makefile
- Umgebung
  - Betriebssystem: Linux-x86_64
  - Compiler: gcc version 4.7.2

Code-Beispiele

- Quellcode
  - Download: ...
  - C99 Standard
  - inklusive Makefile
- Umgebung
  - Betriebssystem: Linux-x86_64
  - Compiler: gcc version 4.7.2
- Laufzeit-Tests
  - auf dem WR-Cluster (AMD-Knoten)
  - GNU gprof GNU Binutils for Ubuntu 2.22

Rückfragen

Fragen

- Verständnisfragen können gerne während des Vortrags gestellt werden!
- Kommentare und Diskussionen am Ende des Vortrages!

Was sind Buffer?

Definition:

A buffer, also called buffer memory, is a portion of a computer's memory that is set aside as a temporary holding place for data that is being sent to or received from an external device, such as a hard disk drive (HDD), keyboard or printer. [Pro13]

Buffer in C

- Allozierter Speicherbereich (Hauptspeicher)
- zeitweise Zwischenlagerung von Daten
- für Daten des gleichen Typs
- Einsatz:
  - zwischen unterschiedlichen Devices mit Geschwindigkeitsunterschieden
  - String-Verarbeitung
  - I/O

Buffer-Beispiel

Beispiel (simple_buffer.c):

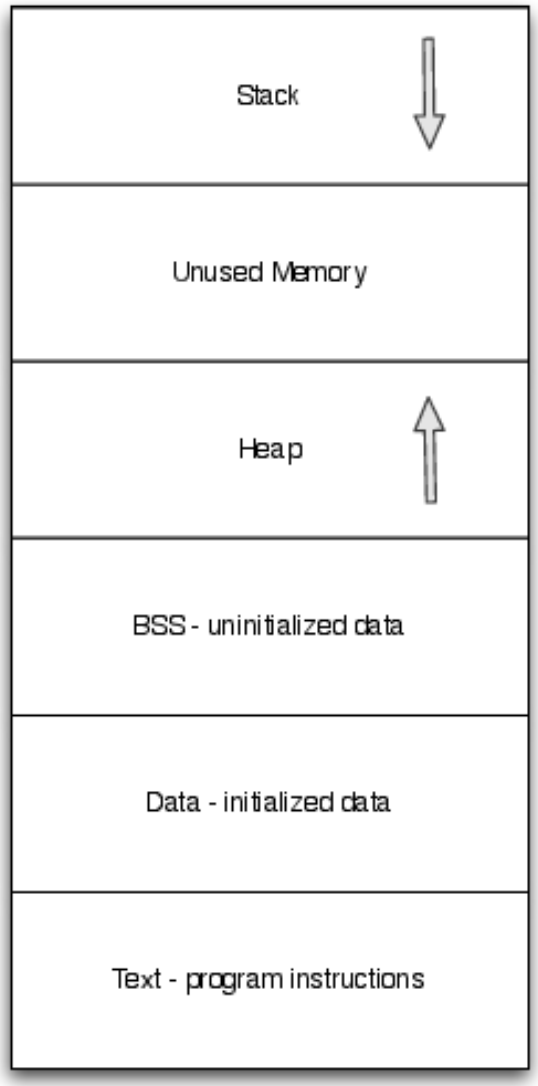
```
#include <stdio.h>

void simple_buffer()
{
        char *buffer = "hello world";
        printf("content: %s\n",buffer);
        printf("bufer: %p\n",buffer);
        printf("&buffer: %p\n",&buffer);
}

int main(int argc, char *argv[])
{
        simple_buffer();
}
```

Buffer-Beispiel

Ausgabe (simple_buffer.c):

```
content: hello world
bufer: 4195828
&buffer: 3246560984
```

Speicheraufbau eines C Programms



Heap- und Stack-Segment

- Stack
  - beginnt an der höchsten Speicheradresse
  - LIFO
  - wächst gegen den Heap (von höchster zu niedrigster Adresse)
  - enthält lokale Variable, Rücksprungadressen u.a.
  - Zugriff über Stack-Pointer

Heap- und Stack-Segment

- Stack
  - beginnt an der höchsten Speicheradresse
  - LIFO
  - wächst gegen den Heap (von höchster zu niedrigster Adresse)
  - enthält lokale Variable, Rücksprungadressen u.a.
  - Zugriff über Stack-Pointer
- Heap
  - beginnt nach dem .bss Segment
  - wächst zu größeren Speicheradressen (gegen den Stack)
  - dynamische Speicher-Allokation (mit malloc, ...)

C Calling Conventions

- Was passiert beim Funktionsaufruf?
  1. Sicherung der Register
  2. Argumente auf den Stack pushen (in umgekehrter Reihenfolge)
  3. Aufruf der Funktion (Rücksprungadresse auf Stack)
     1. Sicherung des Base-Pointers auf den Stack
     2. Base-Pointer := Stack-Pointer
     3. lokale Variablen auf dem Stack
     4. return value in %eax bzw. %qax
     5. Wiederherstellung des Stack- und Base-Pointer
     6. return (ret)
  4. Argumente von Stack entfernen

Funktionsaufruf-Beispiel in C:

```
int add(int arg1, int arg2)
{
        int result = 0;
        result = arg1 + arg2;
        return result;
}

int main(int argc, char *argv[])
{
        int result = add(1,2);
        return result;
}
```

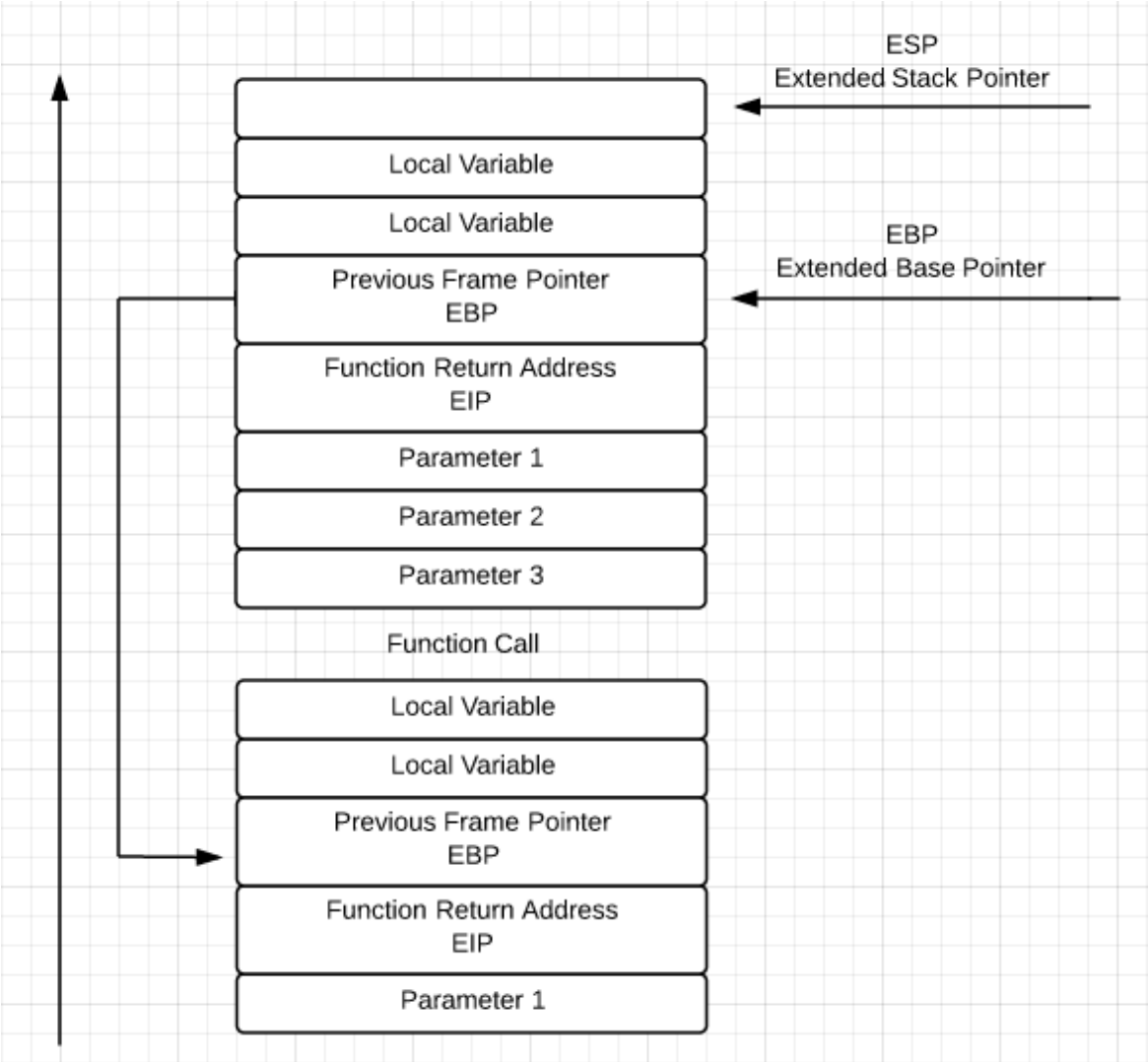
Funktionsaufruf-Beispiel in Assembler(add):

```
add:
        pushl   %ebp ;save base-pointer
        movl    %esp, %ebp
        subl    $16, %esp ;alloc stack memory
        movl    $0, -4(%ebp)
        movl    12(%ebp), %eax
        movl    8(%ebp), %edx
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
```

Funktionsaufruf-Beispiel in Assembler(main):

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp ;alloc stack memory
        movl    $2, 4(%esp) ;push params to stack
        movl    $1, (%esp)
        call    add ;function call
        movl    %eax, -4(%ebp) ;return value in eax
        movl    -4(%ebp), %eax
        leave
        ret
```

Funktionsaufrufe in C (Stack) :



Buffer auf dem Stack

- Reservierung des Speichers durch Änderung des Stack-Pointers
- nach Verlassen der Funktion nicht mehr gültig

stack_buffer.c

```
#include <stdio.h>
#include <string.h>

void buffer()
{
        char buffer[1000];
        strcpy(buffer,"hello world");
        printf("%s",buffer);
}

int main(int argc, char *argv[])
{
        buffer();
}
```

Buffer auf dem Stack

stack_buffer.s

```
buffer:
   pushq %rbp
   movq %rsp, %rbp
   subq $1008, %rsp ;alloc Memory on stack
   leaq -1008(%rbp), %rax
   movabsq $8031924123371070824, %rdx
   movq %rdx, (%rax)      ;write to Buffer
   ...
```

alloca()

```c
#include <alloca.h>
void *alloca(size_t size);
```

- reserviert Speicher auf dem Stack
- kein C-Standard
- kein free() nötig (Freigabe beim Verlassen der Funktion)
- undefiniertes Verhalten bei Stackoverflow

Buffer auf dem Heap

- Reservierung des Speichers durch Allokatoren
- Verwaltung durch OS
- manuelle Freigabe nötig (free())

heap_buffer.c

```
void buffer(){
        char *buffer;
        buffer = (char*)malloc(1000*sizeof(char));
        strcpy(buffer, "hello world");
        printf("%s",buffer);
        free(buffer);
}

int main(int argc, char *argv[]){
        buffer();
}
```

Buffer auf dem Heap

heap_buffer.s

```
buffer:
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movl $1000, %edi ;setup buffer length (malloc param)
  call malloc ;alloc heap space
  movq %rax, -8(%rbp) ;
  movq -8(%rbp), %rax
  movabsq $8031924123371070824, %rdx
  movq %rdx, (%rax)
  ...
```

Performance

Messung Allokation

```
amd1
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 52.69      2.15     2.15 100000000    21.50    21.50  alloca_buffer
 20.41      2.98     0.83 100000000     8.33     8.33  heap_buffer
 19.05      3.76     0.78                             main
  6.80      4.04     0.28 100000000     2.78     2.78  stack_buffer
  1.98      4.12     0.08                             frame_dummy
```

- Zeit für Funktionsaufruf ca. 0.1 sec
- alloca unerwartet langsam
- normale Stack-Allokation schneller als Heap-Allokation
- Grund: Stack-Allokation nur Subtraktion des Stack-Pointers

Vor und Nachteile

Unsichere Heap	Stack
+ "unbegrenzt"	- begrenzt
- langsame Allokation	+ schnelle Allokation
+ Fehler schwerer auszunutzen	-Manipulation der Rücksprungadresse

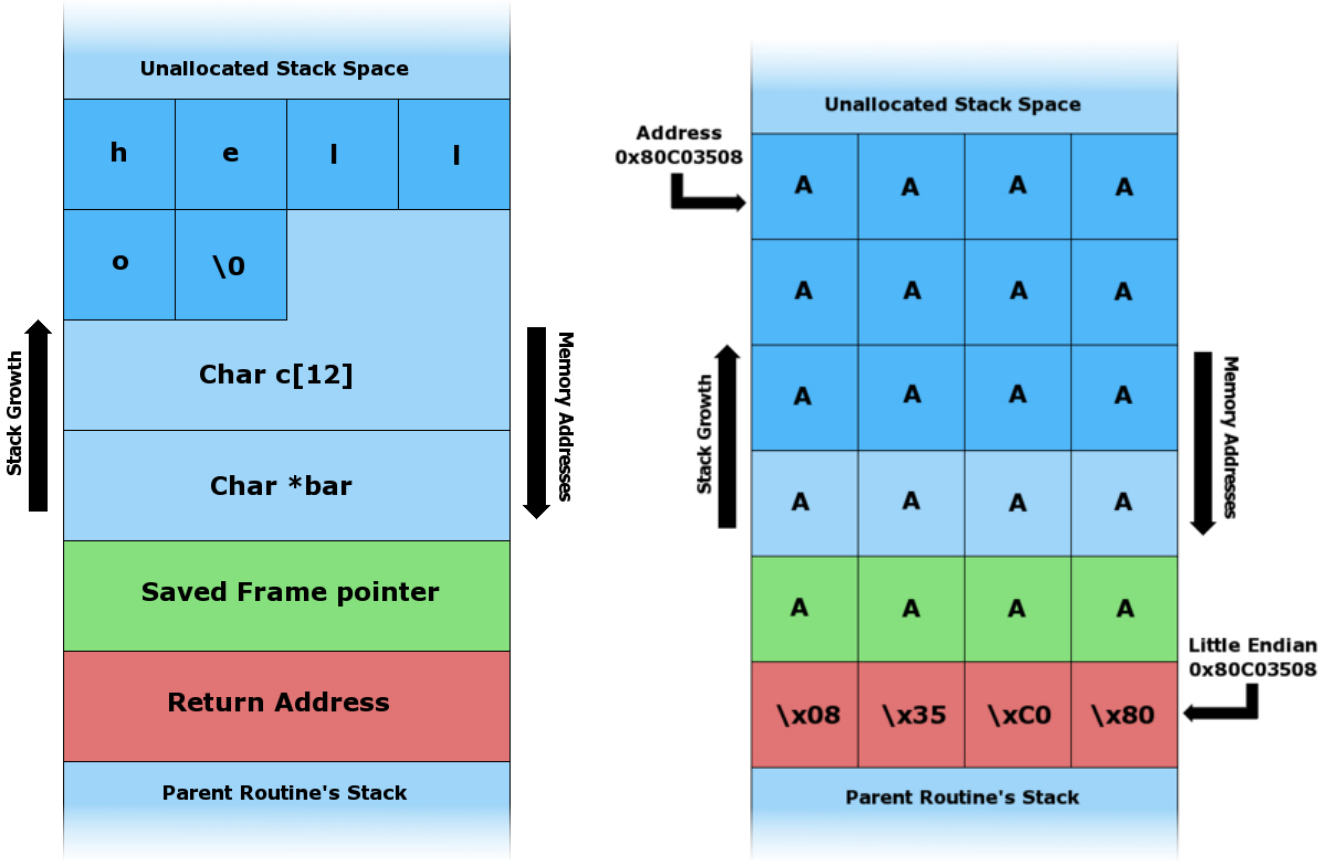
- Wann was?
  - große dynamische Buffer besser auf dem Heap
  - kleine Buffer auf dem Stack

Bufferoverflow

- Es wird über den reservierten Speicherbereich hinweg geschrieben
- Ursachen:
  - keine Prüfung der Länge der Daten
  - Verwendung von unsicheren Funktionen (strcpy,..)
  - Indexfehler
  - Integer-Overflows
- Folgen:
  - Sicherheitslücken
  - Einschleusen von Schadcode möglich
  - Manipulation vom Rücksprungadressen
  - ...

kleine Demo (buffer_overflow.c)

Bufferoverflow Stack



Gegenmaßnahmen

- Programmerstellung:
  - Vor Schreibzugriff Größenprüfung
    - sizeof(buffer) nutzen
    - Makros nutzen
    - Integerüberläufe vermeiden
  - unsicheres Einlesen von Eingabestreams vermeiden
  - unsichere Funktionen zur Stringbearbeitung vermeiden

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);

Gegenmaßnahmen

- Compiler
  - Stack-smashing protection
  - gcc -fstack-protector*
- Betriebssystem
  - Address Space Layout Randomization
  - ASLR kann durch Spraying umgangen werden

Literatur


Jonathan Bartlett.
Programming from the ground up, 2003.


Apple Inc.
Secure coding guide, Juni 2012.


The Linux Information Project.
Buffer definition, Oktober 2013.


Jürgen Wolf.
C von a bis z, 2009.

Was sollte man mitnehmen

- Was sind Buffer?
- Speicheraufbau eines C-Programms
- Was passiert beim Funktionsaufruf?
- Unterschied Heap und Stack
- Was für Fehler können beim Umgang mit Buffern auftreten?
- Was sind Bufferoverflows?
- Warum führen diese zu Sicherheitslücken?
- Wie kann man diese vermeiden?