Sicherheitskritische Programmierfehler

Sebastian Plotz
sebastian.plotz@fh-stralsund.de

28. September 2016

fachhochschule stralsund

Einführung

Wie kommt es zu (sicherheitskritischen) Programmierfehlern in Computerprogrammen?

- Zeitdruck bei der Programmerstellung
- hohe Komplexität und großer Umfang (viele Codezeilen) der Programme
- Unwissenheit des Programmierers
- unzureichende Überprüfung der Eingabedaten vom Benutzer
- absichtliches Hinzufügen von Backdoors ;-)
- ...

Wie sieht ein Computerprogramm aus?

Quellcode eines C-Programms, das „Hallo Welt!“ ausgibt

```
1  #include <stdio.h>
2
3  int main(void) {
4    puts("Hallo Welt!");
5  }
```

- vom Programmierer erstellter, menschenlesbarer **Quellcode**
- architekturunabhängig
- so noch nicht auf der CPU ausführbar
- nächster Schritt: Quellcode *kompilieren*

Wie sieht ein Computerprogramm aus?

Assemblercode des Hallo-Welt-Programms (x86_64)

```
1 push   %rbp
2 mov    %rsp,%rbp
3 mov    $0x4005e0,%edi
4 callq 400410 <puts@plt>
5 mov    $0x0,%eax
6 pop    %rbp
7 retq
```

- vom Compiler generierter, menschenlesbarer **Assemblercode**
- architekturabhängig und maschinennah
- so immer noch nicht auf der CPU ausführbar
- letzter Schritt: Assemblercode *assemblieren*

Wie sieht ein Computerprogramm aus?

Maschinencode des Hallo-Welt-Programms (x86_64)

```
55 48 89 e5 bf e0 05 40 00 e8
d2 fe ff ff b8 00 00 00 00 5d
c3
```

- vom Assembler generierter Maschinencode (1010101 ...)
- hier in Form einer hexadezimalen Bytefolge dargestellt
- architekturabhängig
- jetzt (endlich) auf der CPU ausführbar :-)

Wo „lebt“ ein Computerprogramm?

Vor der Ausführung:

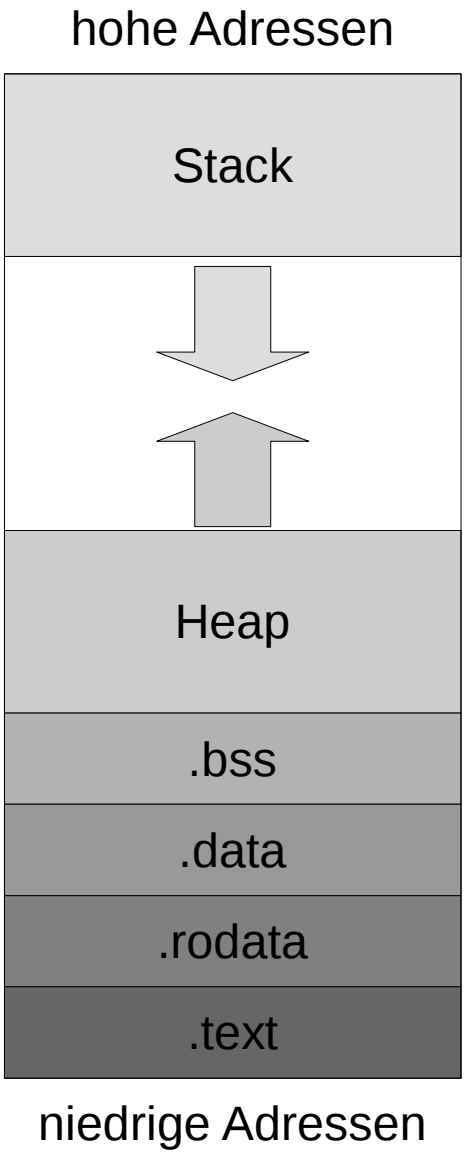
- Programm liegt Form von Maschinencode auf der Festplatte

Während der Ausführung:

- Programm wird beim Start in den **Arbeitsspeicher** geladen (dazu gleich mehr)
- Programm nutzt ebenfalls **CPU-Register** (kleiner Speicher, sehr schneller Zugriff)
- einige Programme greifen zusätzlich zur Laufzeit auf die **Festplatte** zu (sehr großer Speicher, langsamer Zugriff)

Speicherbereiche

- **Stack:** lokale Variablen, Funktionsparameter, gesicherte Registerinhalte
- **Heap:** dynamisch (zur Laufzeit) allokierter Speicher
- **.bss:** nicht initialisierte statische/globale Variablen
- **.data:** initialisierte statische/globale Variablen
- **.rodata:** read-only Daten
- **.text:** ausführbarer Maschinencode



Stack

- Stapelspeicher
- wächst in Richtung niedrigerer Adressen
- Speicherort für lokale Variablen, Funktionsparameter und gesicherte Registerinhalte
- LIFO-Prinzip
- **push:** legt Datum auf dem Stack ab
- **pop:** liest das zuletzt auf dem Stack abgelegte Datum

Buffer Overflows - Grundlagen

Für Buffer Overflows anfälliges C-Programm

```
1  int main(int argc, char *argv[]) {
2    char buf[100]; // lokale Variable --> Stack
3
4    strcpy(buf, argv[1]);
5  }
```

Problem:

Pufferüberlauf, wenn Länge der Benutzerdaten > 100 Byte

- Zeichenkette in C: `Hallo Welt!\0`
- `strcpy()`: kopiert Zeichenkette bis zum ersten Auftreten von `\0`
- `argv[1]`: erster Kommandozeilenparameter (Benutzerdaten)

Buffer Overflows - Beispiel

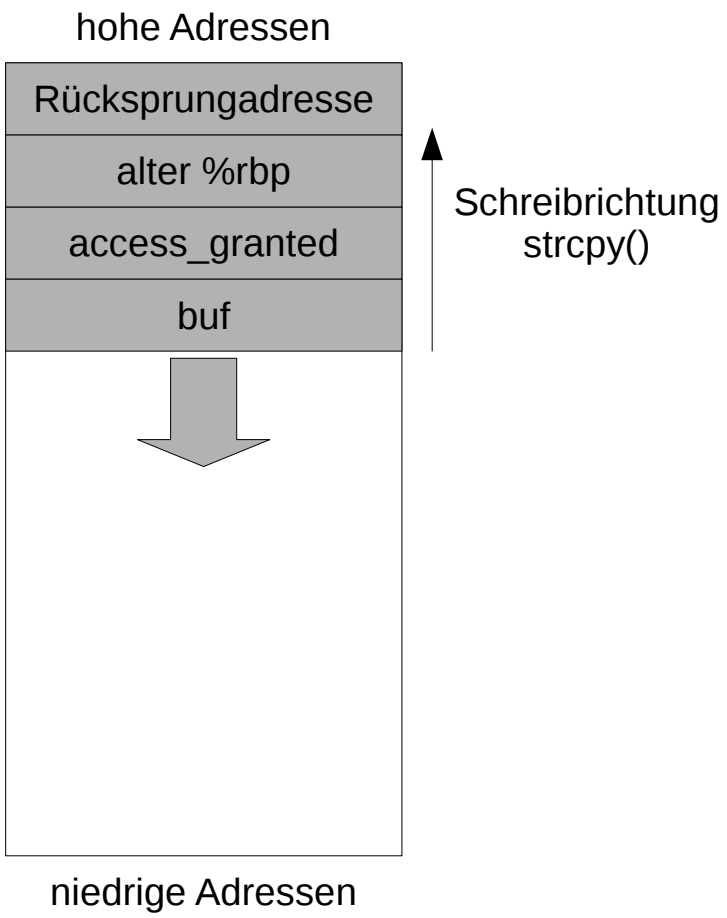
Beispiel Buffer Overflow

```
1  int main(int argc, char *argv[]) {
2    _Bool access_granted = 0; // lokale Variable --> Stack
3    char buf[8];              // lokale Variable --> Stack
4
5    strcpy(buf, argv[1]);
6
7    if(access_granted)
8      puts("Starte System ...");
9  }
```

Ziel:

- Ausgabe von „Starte System ...“ auf der Kommandozeile
- gut für uns: `access_granted` befindet sich ebenfalls auf dem Stack ;-)

Buffer Overflows - Beispiel




```
# ./a.out $(python2 -c 'print "\x41"*15 + "\x01"')
Starte System ...
```

Integer Signedness Bugs - Grundlagen

Darstellung negativer Zahlen (Zweierkomplement)

$|\text{-}1| = 1 = 00000001_2$
NOT 00000001 = 11111110
$11111110_2 + 00000001_2 = 11111111_2 = ff_{16}$

- ff_{16} interpretiert als `signed char`: -1
- ff_{16} interpretiert als `unsigned char`: 255

Integer Signedness Bugs - Beispiel

Beispiel Integer Signedness Bug

```
1  #define MAX 300l
2
3  char *copy_data(char *src, long len) {
4    char buf[MAX];
5
6    if(len > MAX)
7      return NULL;
8
9    memcpy(buf, src, len);
10   return buf;
11 }
```

Problem (Annahme: len = 0*xffffffffffffffff*):

- Größenüberprüfung in Zeile 6 vorzeichenbehaftet (-1 < 300)
- `memcpy()` erwartet Längenangabe vom Typ `size_t` (meist `unsigned long`) und interpretiert den Wert als sehr große Zahl –> Buffer Overflow)

Quiz

Fällt Ihnen an diesem C-Quellcode etwas auf?

```
1 if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
2   retval = -EINVAL;
```

Auflösung:

- es handelt sich um eine Backdoor für den Linux-Kernel ;-)
- = (Zuweisung) anstatt == (Vergleich)
- User ID von *root*: 0
- zweite Bedingung nie wahr

Path Traversal

Beispiel Path Traversal

```
1 int main(int argc, char *argv[]) {
2   FILE *fp;
3   fp = fopen(argv[1], "a+");
4   fwrite(argv[2], sizeof(char), strlen(argv[2]), fp);
5   fclose(fp);
6 }
```

Problem:

- Benutzer hat die Kontrolle über den Dateipfad
- mit Eingabe von „../“ oder eines absoluten Pfades ist es möglich, das aktuelle Verzeichnis zu verlassen
- je nach Benutzerrechten ist es außerdem möglich, anschließend Dateien zu verändern/löschen/erstellen oder zu lesen

Schutzmaßnahmen

Buffer Overflows

- Canaries (GCC-Option `-fstack-protector`)
- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)

Integer Signedness Bugs

- GCC-Option `-Wconversion`

Zuweisung anstatt Vergleich

- Yoda Notation: `if (42 == wert)`

Path Traversal

- Zeichen wie „/“ aus dem Pfad entfernen