Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

# SECURITY
## Softwaresicherheit

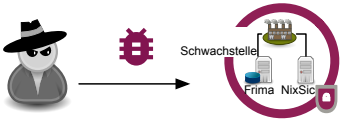
June 23, 2023

Marc Stöttinger

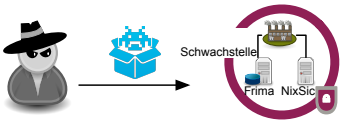
> Security is not a product, but a process. It's about building a culture of secure coding, continuous testing, and proactive vulnerability management throughout the software development lifecycle.
>
> Bruce Schneier

# VORGEHEN EINES ANGREIFERS

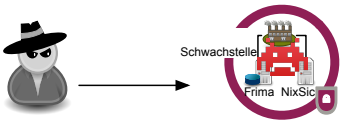
1. Analyse des Ziels auf Software **Schwachstellen**




2. Ausnutzung der Schwachstellen durch **Exploits**
   → Schwachstellen und Exploits bedingen sich gegenseitig

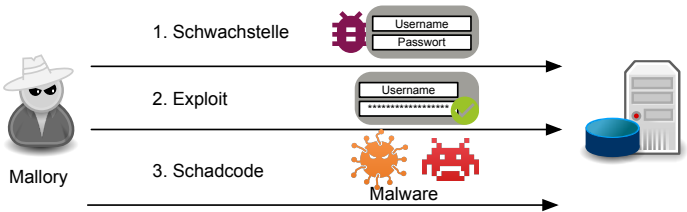



3. **Schadhafte Aktionen**
   → Malware kann Exploits beinhalten, um sich selbstständig zu verbreiten



# BEISPIEL ANGRIFF [XBOX]

1. **Schwachstelle**: Login ohne Passwort mittels langem String möglich
2. **Exploit**: Angreifer loggt sich mittels langem String ein
3. **Schadcode**: Angreifer installiert Programm, das Daten verschlüsselt



# MELDUNG VON SCHWACHSTELLEN

Entdeckte Schwachstellen und zusätzliche Informationen werden via verschiedener Schwachstellendatenbanken publiziert

→ Bekannteste Datenbank: „Common Vulnerabilities and Exposures **(CVE)**“

→ Datenbank ermöglicht gezielte Suche. z.B. Schwachstellen bestimmter Programme

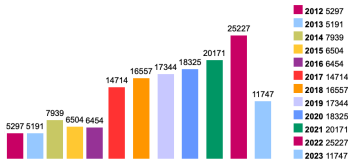
## Suchergebnisse nach Schwachstellen

**Search Results**

There are **641** CVE Records that match your search.

| Name | Description |
|---|---|
| CVE-2023-31127 | libspdm is a sample implementation that follows the DMTF SPDM specifications. A vulnerability has been identified in SPDM session establishment in libspdm prior to version 2.3.1. If a device supports both DHE session and PSK session with mutual authentication, the attacker may be able to establish the session with `KEY_EXCHANGE` and `PSK_FINISH` to bypass the mutual authentication. This is most likely to happen when the Requester begins a session using one method (DHE, for example) and then uses the other method's finish (PSK_FINISH in this example) to establish the session. The session hashes would be expected to fail in this case, but the condition was not detected. This issue only impacts the SPDM responder, which supports `KEY_EX_CAP=1 and `PSK_CAP=10b` at same time with mutual authentication requirement. The SPDM requester is not impacted. The SPDM responder is not impacted if `KEY_EX_CAP=0` or `PSK_CAP=0` or `PSK_CAP=01b`. The SPDM responder is not impacted if mutual authentication is not required. libspdm 1.0, 2.0, 2.1, 2.2, 2.3 are all impacted. Older branches are not maintained, but users of the 2.3 branch may receive a patch in version 2.3.2. The SPDM specification (DSP0274) does not contain this vulnerability. |
| CVE-2023-28238 | Windows Internet Key Exchange (IKE) Protocol Extensions Remote Code Execution Vulnerability |
| CVE-2023-24859 | Windows Internet Key Exchange (IKE) Extension Denial of Service Vulnerability |

## CVE Meldungen pro Jahr




Quelle: **https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Exchange** und **https://www.cvedetails.com/browse-by-date.php**

# EFFEKTE VON SCHWACHSTELLEN UND EXPLOITS

- Schwachstellen können verschiedene **Effekte** zulassen:
  - Veränderung der Funktionsweise des Systems (z.B. Preise reduzieren)
  - Auslesen geheimer Informationen (z.B. Ausgabe aller Daten einer Datenbank)
  - Störung der Verfügbarkeit des Systems (z.B. Absturz eines Programms)
  - Ausführen eigener Programme auf dem Zielsystem (z.B. Ausführen von Malware)

- Stark variierende **Voraussetzungen zur Ausnutzung** einer Schwachstelle
  - Automatisiert über Internet auf alle Installationen eines Programms
  - Lokal am PC mit Adminrechten bei spezieller Konfiguration

- Priorisierung der Schwachstellen ist notwendig, wenn viele gleichzeitig anfallen

# BEWERTUNG VON SOFTWARE SCHWACHSTELLEN

- der Basic Score vom Common Vulnerability Scoring System **(CVSS)** bewertet Software Schwachstellen basierend auf
  - Komplexität der Ausnutzung
  - Auswirkungen auf Vertraulichkeit, Integrität und Verfügbarkeit
- Die komplette Bewertung im [CVSS] berücksichtigt den Basic Score, einen Temporal und einen Enviromental Score.

| CVSS Score | Number Of Vulnerabilities | Percentage |
|---|---|---|
| 0-1 | 23965 | 11.70 |
| 1-2 | 1198 | 0.60 |
| 2-3 | 8337 | 4.10 |
| 3-4 | 9494 | 4.70 |
| 4-5 | 42988 | 21.10 |
| 5-6 | 34098 | 16.70 |
| 6-7 | 27167 | 13.30 |
| 7-8 | 35998 | 17.60 |
| 8-9 | 898 | 0.40 |
| 9-10 | 19973 | 9.80 |
| Total | 204116 | |

Quelle: **https://www.cvedetails.com/**

**cvss-score-distribution.php**

- Komplexität der Ausnutzung wird angegeben via:
  - **Angriff möglich aus**: Internet, LAN, System Zugang, Physischer Zugang
  - **Angriffskomplexität**: keine Bedingungen, Wissen / Zugang benötigt
  - **Benötigte Rechte**: Keine, User, Admin
  - **Opfer Interaktion nötig**: Keine, Interaktion benötigt
  - **Reichweite**: Beschränkt auf anfällige Komponente, Sprung aus Komponente möglich

# CVSS BEISPIELE

- → **Log4j**: Nachladen und Ausführen von Programmen möglich, wenn ein bestimmter String an Anwendung mit Log4j Framework gesendet wird (z.B. Apache Webserver oder Minecraft Server)

- → **WhatsApp**: Interaktion mit WhatsApp via Siri möglich, selbst bei aktivem Sperrbildschirm auf IPhone

| Kriterium | Log4j: CVE-2021-44228 | WhatsApp: CVE-2020-1908 |
|---|---|---|
| Angriff möglich aus | Netzwerk | Physikalischer Zugang |
| Angriffskomplexität | Niedrig | Niedrig |
| Benötigte Rechte | Keine | Keine |
| Opfer Interaktion nötig | Keine | Keine |
| Reichweite | Sprung möglich | Kein Sprung möglich |
| Vertraulichkeit | Hoch | - |
| Integrität | Hoch | Hoch |
| Verfügbarkeit | Hoch | - |
| Gesamtscore | 10,0 | 4,6 |

# HÄUFIG AUFTRETENDE SCHWACHSTELLEN

Häufig auftretende Schwachstellen werden analysiert und publiziert

- → **Generelle Schwachstellen**: Common Weaknesses Enumeration [CWE]
- → **Schwachstellen in Webanwendungen**: Open Web Application Security Project [OWASP]

| Platz | Schwachstellenbeschreibung | Häufigkeit in 2022 | Mittl. CVSS 2022 |
|---|---|---|---|
| 1 | Schreiben außerhalb des festgelegten Bereichs | 4123 | 7,93 |
| 2 | Cross-Site Scripting | 4740 | 5,73 |
| 3 | SQL Injection | 1263 | 8,66 |
| 4 | Unzureichende Eingabevalidierung | 1520 | 7,19 |
| 5 | Lesen außerhalb des festgelegten Bereichs | 1489 | 6,54 |

# HÄUFIGE SOFTWAREFEHLER

- Zugriffe außerhalb eines festgelegten Bereiches beinhalten häufig die Fehler
  - Buffer Overflow [Buf] und
  - Fehlerhafte Integer Behandlung [Calc]

- Fokus auf die Programmiersprache C, aufgrund weiter Verbreitung und einfacher Erklärung

- Viele Fehlerquellen in C werden in moderneren Sprachen vermieden, z.B.:
  - **Java**: Laufzeitprüfung und Exception Handling (z.B. ArrayIndexOutOfBoundsException)
  - **Rust**: Kapazitätsprüfung der Puffer vor Zugriff und Ownership Mechanik

# C BASICS

→ Primitive Datentypen können unterschiedliche Wertebereiche darstellen

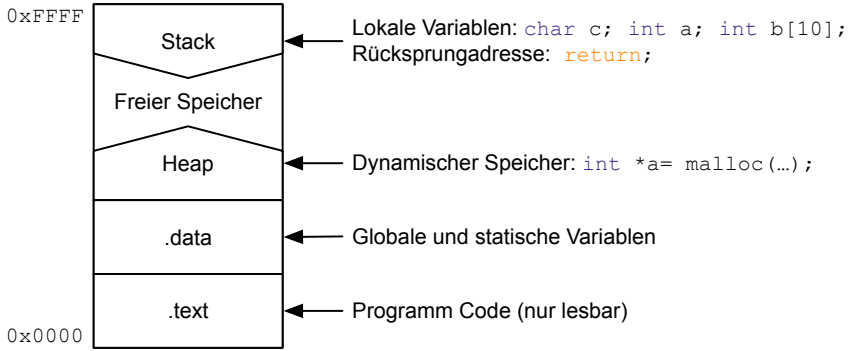
| Daten-typ | Länge (Bit) | Wertebereich (unsigned) | Wertebereich (signed) |
|---|---|---|---|
| char | 8 | -128 bis 127 | 0 bis 255 |
| short | 16 | -32.769 bis 32.767 | 0 bis 65.535 |
| int | 32 | -2.147.483.648 bis 2.147.483.647 | 0 bis 4.294.967.295 |
| long | 64 | -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 | 0 bis 18.446.744.073.709.551.615 |

Wertebereiche für eine 32-Bit Maschine

→ Direkte Operation auf Speicher möglich
  → Reservierung: malloc(len) – Reserviere einen Speicherbereich von len Bytes
  → Kopieren: memcpy(Ziel, Quelle, len) – Kopiere len Bytes von Quelle nach Ziel

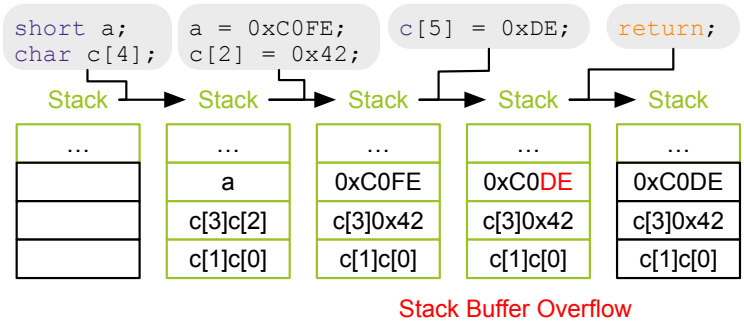
# SPEICHERLAYOUT EINES C-PROGRAMMS

Die Speicherorganisation eines C-Programms ist wie folgt aufgebaut:



# STACK- UND HEAP BUFFER OVERFLOWS

```
void foo(){
  short a;
  char c[4];

  a = 0xC0FE;
  c[2] = 0x42;

  c[5] = 0xDE;

  return;
}
```



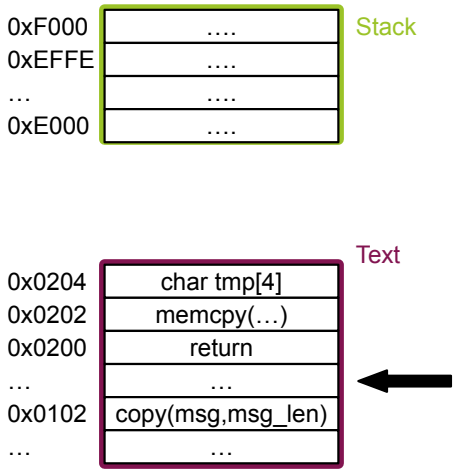

→ Heap Buffer Overflow: Analog für dynamisch allozierten Speicher im Heap

# AUSNUTZUNG VON BUFFER OVERFLOWS

- → Bei Buffer Overflows werden Daten außerhalb des festgelegten Bereiches geschrieben
- → Wie können Buffer Overflows ausgenutzt werden?
  - → Programmabstürze aufgrund inkonsistenter Daten (z.B. falsche Pointer)
  - → Gezielte Modifikation der Daten im Speicher (z.B. Änderung eines Preises)
  - → Ausführung wahlfreien Codes (z.B. Ausführung der Eingabedaten)
- → **Beispiel im Folgenden**: Ausführung wahlfreien Codes

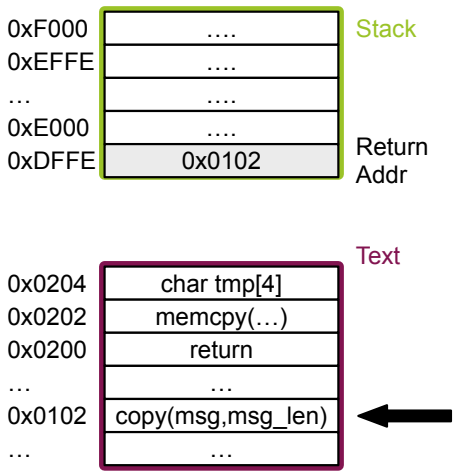
# STACK BUFFER OVERFLOW AUSNUTZUNG I

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



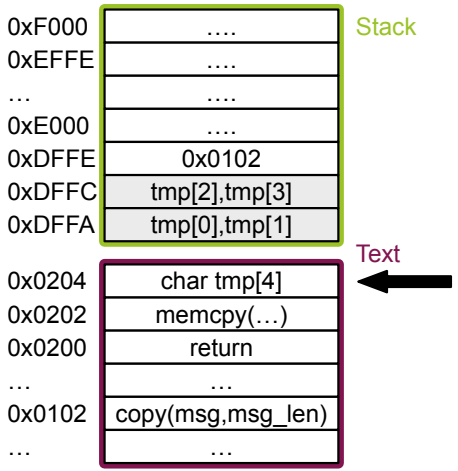
# STACK BUFFER OVERFLOW AUSNUTZUNG II

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



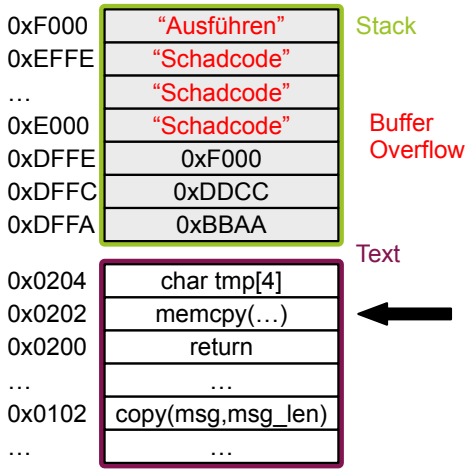
# STACK BUFFER OVERFLOW AUSNUTZUNG III

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



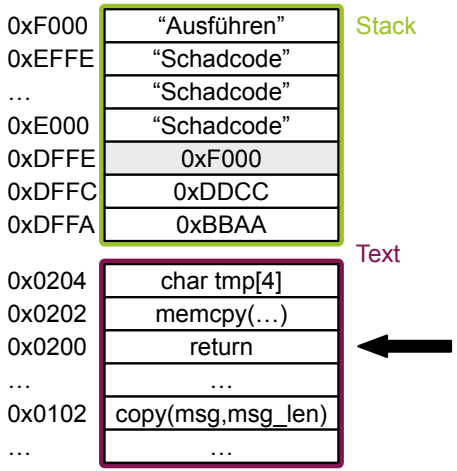
# STACK BUFFER OVERFLOW AUSNUTZUNG IV

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



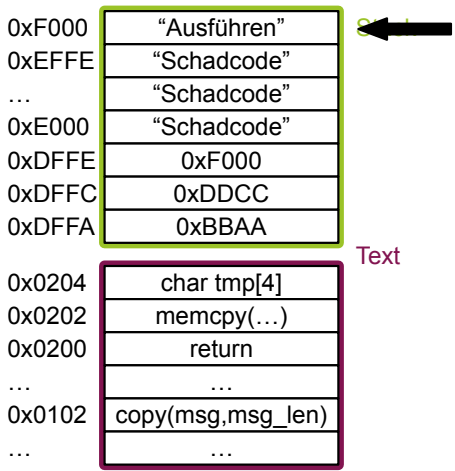
# STACK BUFFER OVERFLOW AUSNUTZUNG V

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



# STACK BUFFER OVERFLOW AUSNUTZUNG VI

```
char msg[] = {0xAA,0xBB,0xCC,0xDD,
              0x00,0xF0,"Schadcode",
              "Ausführen"};

void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}

int main(){
…
  copy(msg, sizeof(msg));
…
}
```



# AUSLÖSUNG EINES BUFFER OVERFLOWS

→ Einfache Vermeidung von Buffer Overflows: Prüfe zu kopierende Menge an Daten gegen Puffergröße

```
void copy(char *msg, int msg_len){
  char tmp[4];
  memcpy (tmp, msg,msg_len);
  return;
}
```

```
void copy(char *msg, int msg_len){
  char tmp[msg_len];
  memcpy (tmp, msg,msg_len);
  return;
}
```

→ Häufig werden mehrere Fehler mit dem Buffer Overflow ausgenutzt
  → Hafnium Hack zu Microsoft Exchange: 4 Schwachstellen [HAF]

# DISKUSSION IN KLEINEN GRUPPEN

In welchen Beispiel wird ein Buffer Overflow ausgelöst und warum?

Beispiel 1

```
int main() {
    int num = receive(4);
    int* buf = (int*) malloc(num * 4);
    for (int i = 0; i <= num; i++)
        buf[i] = receive(4);

    return 0;
}
```

Beispiel 2

```
int main() {
    int bytes = receive(4);
    char *source = receive(bytes);
    int obj_size = 1000;
    int num_obj = bytes / obj_size;

    char* obj_buf = (char*) malloc(num_obj * obj_size);

    memcpy(obj_buf, source, bytes);
    return 0;
}
```

Beispiel 3

```
int main() {
    short buf_bytes = 1000;
    char* buf = (char*) malloc(buf_bytes);
    int bytes = receive(4);
    char* source = receive(bytes);

    if((short) bytes > buf_bytes)
        return 1;

    memcpy(buf, source, bytes);
    return 0;
}
```

Hinweis:
Die Funktion `receive(int n)` gibt `n` Bytes zurrück.

# INTEGER WRAP AROUND/OVERFLOW

- Primitive Datentypen besitzen begrenzten Zahlenraum
  - Verlassen des Zahlenwertes muss überprüft und abgefangen werden

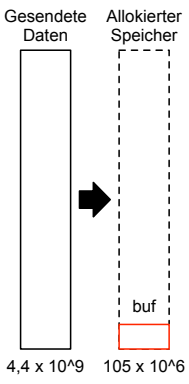
```
//Empfange einen Vektor von Integer Werten
bool receive_integer_vector() {

  //Anzahl der Integer Werte die empfangen werden müssen (als 4 Byte Wert)
  unsigned int num_values = receive(4);

  //Alloziere Puffer richtiger Größe
  int* buf = (int*) malloc(num_values * 4);

  //Prüfe ob ein Puffer alloziert wurde
  if(buf == NULL)
    return false;

  //Empfange jeden Integer Wert (4 Byte) und speichere ihn im Puffer
  for(unsigned int i = 0; i < num_values; i++)
    buf[i] = receive(4);

  free(buf);
  return true;
}
```

-1 < unsigned int < 4.294.967.296

1.100.000.000

4.400.000.000 > 4.294.967.295
Differenz= 105.032.705

Ab I = 105.032.705/4= 262.258.176
Buffer Overflow



# FEHLINTERPRETATION BEI INTEGER TYPECASTING

- Beim Arbeiten mit verschiedenen primitiven Datentypen können Werte fehlinterpretiert werden
  - Verlust von Informationen beim expliziten Typecasting auf einen kleineren Typen
  - Kontraintuitive Interpretation beim impliziten Typecasting

Typecasting auf kleineren Typen

```
int main() {
    short a = 42;
    int b = -65494;

    if(a == (short) b) {
        printf("a ist gleich b\n");
    } else {
        printf("a ist ungleich b\n");
    }
    return 1;
}
```

```
>gcc equals.c
>./a.out
a ist gleich b
```

a = 0x002A
b = 0xFFFF FFFF FFFF 002A

Implizites Typecasting

```
int main() {
    short a = 42;
    int b = -1;

    if(a == b) {
        printf("a ist gleich b\n");
    } else {
        printf("a ist ungleich b\n");
    }
    return 1;
}
```

```
>gcc compare.c
>./a.out
b ist größer als a
```

a = 0x002A
b = 0xFFFF

Vergleich beider Werte im unsigned short Format

# AUSLÖSUNG EINES BUFFER OVERFLOWS

- Weitere, Integer-bezogene Fehler:
  - Kleine Offsets, die nicht erkannt werden (z.B. Off-by-one Fehler [SSH])
  - Division durch 0 nicht abgefangen (z.B. Media Player DoS [MP])
  - Inkonsistente Eingaben (z.B. 64 KB Nachricht mit 16 Byte Größe angegeben [HB])
  - Erwartete Formate werden gebrochen (z.B. Nachricht ist immer 1000 Byte lang [BG])

- Weitere Programmierfehler:
  - Fehler beim Umgang mit Strings und Character Arrays ([Java], [C])
  - Fehler beim Umgang mit Expressions ([Java], [C])
  - Fehler beim Umgang mit Datei Input / Output ([Java], [C])

# ERKENNUNG UND VERHINDERUNG VON SOFTWARE EXPLOITS

- → Software Schwachstellen $\subseteq$ Software Bugs
  - → Weniger Bugs → Weniger Schwachstellen
  - → **Aber**: Alle Bugs in einem Programm zu beseitigen ist schwer

- → Möglichkeiten zur Erkennung von Software Schwachstellen
  - → Security Tests (z.B. Penetration Tests)
  - → Code Reviews (4-Augen Prinzip)
  - → **Statische**- und **dynamische Analyse**

- → Möglichkeiten zur Vermeidung der Ausnutzung
  - → Technische Mechanismen (z.B. **Stack Canaries** gegen Buffer Overflows)
  - → Isolierung kritischer Softwarekomponenten (z.B. Sandboxing durch Virtualisierung)

# PROGRAMM- UND CODEANALYSE

- → **Grundidee**: Implementierungsfehler sind oft schwer manuell zu erkennen
- → Automatisierter Ansatz, um Fehler zu highlighten oder auszulösen
  - → **Statische Analyse**: Code wird analysiert aber nicht ausgeführt
  - → **Dynamische Analyse**: Code wird mit Eingaben ausgeführt, die speziell gewählt wurden, um eine Fehlreaktion auszulösen (z.B. Grenzwerte, zu hohe Werte, NULL, ...)
- → Statische- und dynamische Analyse haben ihre jeweiligen Vor- und Nachteile
  - → **Statische Analyse**: Detaillierte Findings, von denen aber viele irrelevant sind
  - → **Dynamische Analyse**: Konzentration auf relevante Findings, aber hoher manueller Analyseaufwand, um Ursache zu identifizieren

# BEISPIEL STATISCHE ANALYSE

- Statische Programmanalyse sucht nach Mustern im Code
  - **Compilerwarnungen**: Häufige Fehlermuster (z.B. in gcc via „–Wall“ und „–Wextra“)
  - **Spezielle Codeanalysetools**: Überprüfung der Regeln von Coding Standards (z.B. CERT-C [CERTC], MISRA-C [MISRA], CERT-Java [CERTJ])

```c
int main(){
  unsigned int a = 42;
  Int b = -1;

  if(a < b){
    printf("a ist größer als b\n");
  } else{
    printf("b ist größer als a\n");
  }

  return 0;
}
```

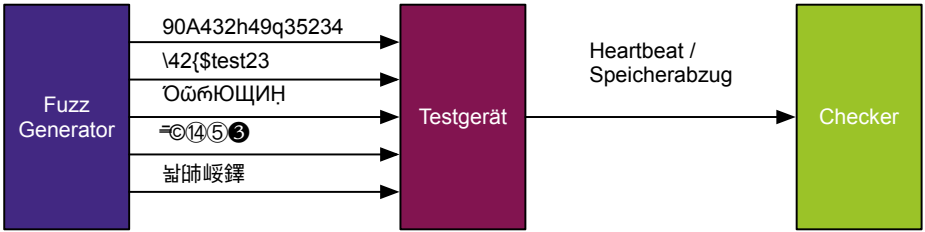
```
>gcc compare.c
>./a.out
b ist größer als a
```

```
>gcc compare.c -Wextra
compare.c: In function ‘main’:
compare.c:7:7: warning: comparison of integer expressions
 of different signedness: ‘unsigned int’ and ‘int’ [-Wsig
n-compare]
    7 |  if(a > b) {
      |       ^
```

# BEISPIEL DYNAMISCHE ANALYSE

- Dynamische Analyse unterstützt die Fehlersuche durch Ausführung des Codes
  - **Fuzzing**: Automatisiertes Testen mit zufällig und bewusst non-konform gewählten Werten mit dem Ziel Fehlverhalten auszulösen
  - **Taint Analyse**: Analyse des Einflusses durch externe Eingaben
  - **Symbolische Ausführung**: Testabdeckung aller Pfade in einem Programm durch Analyse und Auswahl geeigneter Eingaben

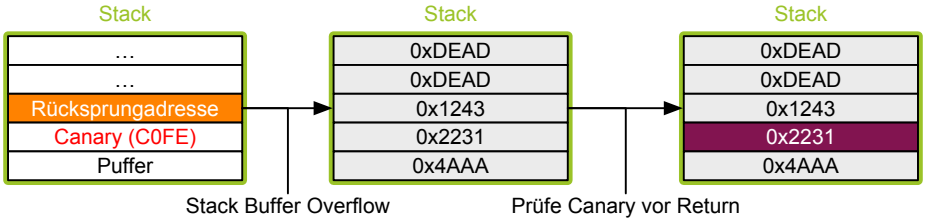
Beispielaufbau- und Ablauf von Fuzzing



# TECHNISCHE GEGENMASSNAHMEN BEISPIEL STACK CANARIES

- Stack Canaries sind zufällige Werte, die vor die Rücksprungadresse gelegt werden, um Stack Buffer Overflows vor dem Rücksprung festzustellen

- Die Sicherheit von Stack Canaries hängt davon ab, dass [BKK+18]:
  - Der Canary Wert nicht geraten werden kann
  - Der Referenzwert und Vergleich „sicher“ gespeichert und implementiert sind



# ERKENNUNG UND VERMEIDUNG VON SOFTWARE SCHWACHSTELLEN

- → Beseitigung aller Software Schwachstellen im Programm ist unmöglich
  - → Moderne Software umfasst oftmals hunderttausende bis Millionen Zeilen Code
  - → Statische und dynamische Analysen erkennen nur bekannte Muster
  - → Zeit- und Kostendruck verhindern häufig weitere Beseitigung

- → Selbst wenn der eigene Code frei von Schwachstellen ist
  - → Compiler verändern Code und führen Schwachstellen ein [MSC]
  - → Inkludierte Bibliotheken können Fehler beinhalten (z.B. log4j)
  - → Angreifer können gezielt korrumpierte Bibliotheken einfügen (z.B. Supply-Chain Angriffe)
  - → Ausführende Hardware kann Fehler beinhalten (z.B. Meltdown/Spectre [MELT])

- → Priorisierung der Mechanismen und der untersuchten Softwaremethoden sowie ein gutes Schwachstellenmanagement sind essentiell für sichere Software

# ZUSAMMENFASSUNG

- → Hintergrund von CVSS und kennen von fünf relevanten Kriterien zur Berechnung
- → Die fünf häufigsten auftretenden Software Schwachstellen
- → Definition und technischer Ablauf eines Buffer-Overflows in C
- → Schwachstellen basierend auf fehlerhafter Integer Behandlung
- → Die Effekte von Schwachstellen
- → Methoden, um Softwareschwachstellen zu erkennen bzw. zu vermeiden