# 1 Software Security

## 1.1 Einleitung und Grundlagen

### 1.1.1 Definition und Zielsetzung

**Software Security**

Software Security bezeichnet die Absicherung von Software (oft nicht sicherheitskritischer Anwendungen) gegen die Ausnutzung von Schwachstellen. Ziel ist es, Angriffe zu verhindern, die Eigenschaften von Programmiersprachen, Systemarchitekturen oder Ausführungsumgebungen missbrauchen.

Angreifer versuchen häufig, durch Ausnutzung dieser Schwachstellen (z.,B. Memory Corruption) schrittweise ihre Rechte zu erweitern (**Privilege Escalation**) und die Kontrolle über das System zu übernehmen.

### 1.1.2 Programm-Lifecycle und Angriffsflächen

Der Weg vom Quellcode bis zum laufenden Prozess bietet verschiedene Angriffsvektoren:

- **Source Code:** Bugs, Logikfehler, Undefined Behavior.
- **Kompilierung:** Compiler-Optimierungen, die Sicherheitschecks entfernen könnten.
- **Binary (ELF):** Manipulation, Reverse Engineering.
- **Laufzeit (Prozess):** Injection, Exploits, Timing-Attacken.

### 1.1.3 Speichermodell (Linux x86_64)

Das Verständnis des virtuellen Speichers ist essenziell für Exploits. Ein Prozessspeicher ist typischerweise wie folgt aufgebaut (von niedrigen zu hohen Adressen):

1. **Text Segment (.text):** Ausführbarer Maschinencode (Read-Only).
2. **Data Segments (.data, .bss):** Globale/statische Variablen.
3. **Heap:** Dynamischer Speicher (wächst nach oben, verwaltet durch `malloc/free`).
4. **Memory Mapping Segment (mmap):** Dynamische Bibliotheken (.so), Thread Stacks.
5. **Stack:** Lokale Variablen, Rücksprungadressen (wächst nach unten Richtung Heap).

**Der Stack Frame** Bei jedem Funktionsaufruf wird ein neuer Stack Frame angelegt. Dieser enthält:

- **Return Address (RIP):** Wohin springt das Programm nach der Funktion zurück?
- **Saved Frame Pointer (RBP):** Referenz auf den vorherigen Stack Frame.
- **Lokale Variablen:** Puffer und Variablen der Funktion.

**Gefahr:** Da Metadaten (Return Address) und Benutzerdaten (lokale Puffer) auf dem Stack benachbart liegen, kann ein Überlauf der Daten die Steuerdaten überschreiben.

## 1.2 Schwachstellen und Exploits

Schwachstellen werden oft nach *CWE (Common Weakness Enumeration)* klassifiziert.

### 1.2.1 Arithmetik-Fehler

Programmiersprachen wie C prüfen Rechenoperationen nicht standardmäßig auf Gültigkeit.

- **Integer Overflow/Underflow:** Ein Wert überschreitet das Maximum (Wrap-around). Dies kann bei der Berechnung von Puffergrößen fatal sein.
- **Off-By-One:** Ein Schleifendurchlauf zu viel oder zu wenig (oft bei String-Grenzen oder Array-Indizes).
- **Pointer-Arithmetik:** In C ist `array[i]` äquivalent zu `*(array + i)`. Eine falsche Berechnung von `i` oder des Typs führt zu **Out-of-Bounds (OOB)** Zugriffen.

### 1.2.2 Buffer Overflows

**Stack-based Buffer Overflow** Tritt auf, wenn mehr Daten in einen lokalen Puffer geschrieben werden, als dieser fassen kann, ohne Längenüberprüfung (z.,B. durch `gets`, `strcpy`).

- **Mechanismus:** Der Angreifer überschreibt lokale Variablen → Saved RBP → **Return Address**.
- **Folge:** Kontrolle über den *Instruction Pointer (RIP)* und damit über den Programmfluss.

### 1.2.3 Heap-Schwachstellen

- **Unzureichende Initialisierung:** Speicher wird reserviert (`malloc`), aber nicht geleert. Sensible Altdaten können ausgelesen werden.
- **Double-Free:** Speicher wird zweimal freigegeben. Dies korrumpiert die Verwaltungsstrukturen des Allocators.
- **Use-After-Free (Dangling Pointer):** Ein Zeiger wird weiterverwendet, nachdem der Speicher freigegeben (und ggf. neu vergeben) wurde.

### 1.2.4 Injections

- **Code Injection:** Einschleusen von Shellcode in ausführbare Segmente.
- **Deserialization:** Unsicheres Laden von Objekten (z.,B. Python `pickle`), bei dem beim Wiederherstellen des Objekts beliebiger Code ausgeführt wird.

### 1.2.5 Timing Angriffe & Race Conditions

**TOCTTOU (Time Of Check To Time Of Use)**

Eine Schwachstelle, bei der sich der Systemzustand zwischen der Überprüfung einer Bedingung (Check) und der Nutzung der Ressource (Use) ändert.

**Beispiel:** Ein Programm prüft, ob der User Schreibrechte auf `/tmp/file` hat (Check). Der Angreifer tauscht `/tmp/file` schnell gegen einen Symlink auf `/etc/passwd` aus, bevor das Programm schreibt (Use).

### 1.2.6 Format String Angriffe

Funktionen wie `printf` nutzen Format-Strings (`%s`, `%x`).

- Wenn User-Input direkt als Format-String genutzt wird (`printf(user_input)` statt `printf("%s", user_input)`):
- **Read:** `%x` liest Werte vom Stack (Information Leak).
- **Write:** `%n` schreibt die Anzahl der bisher ausgegebenen Bytes an eine Adresse auf dem Stack (Arbitrary Write).

## 1.3 Mitigierung und Hardening

### 1.3.1 Security by Design

Der beste Schutz ist die Vermeidung von Fehlern durch:

- Verwendung von *memory-safe languages* (z.,B. Rust, Java, Go).
- Strenge Code-Reviews und hohe Testabdeckung.
- Security Engineering Practices.

### 1.3.2 Binary Hardening (Compiler & Linker Flags)

Diese Maßnahmen erschweren die Ausnutzung von Sicherheitslücken in C/C++ Programmen.

**NX (No-Execute) / W^X** Speicherseiten sind entweder beschreibbar (Write) ODER ausführbar (Execute), niemals beides gleichzeitig.

- **Effekt:** Verhindert die Ausführung von Code auf dem Stack oder Heap (klassische Shellcode-Injection).
- Auch bekannt als DEP (Data Execution Prevention).

**RELRO (Relocation Read-Only)** Schützt die *Global Offset Table (GOT)*, die Funktionsadressen dynamischer Bibliotheken enthält.

- **Partial RELRO:** GOT steht nach dem Linken fest, bleibt aber schreibbar.
- **Full RELRO:** GOT wird beim Start komplett aufgelöst und danach auf *Read-Only* gesetzt. Verhindert GOT-Overwrites.

**Stack Canaries** Ein Zufallswert ("Kanarienvogel") wird zwischen lokalen Puffer und Rücksprungadresse (Return Address) auf dem Stack platziert.

- Vor dem `ret` prüft die Funktion, ob der Canary noch intakt ist.
- Bei einem Buffer Overflow wird der Canary zwangsläufig mit überschrieben → Programmabbruch, bevor der Angreifer den Kontrollfluss umleiten kann.
- **Limitierung:** Kann durch Info-Leaks oder Brute-Force (siehe Fork-Server) umgangen werden.

### 1.3.3 OS-Level Schutzmaßnahmen

**ASLR (Address Space Layout Randomization)** Randomisiert die Adressen von Stack, Heap und Bibliotheken bei jedem Programmstart.

- Angreifer kennen die Adressen für Sprungziele oder Shellcode nicht.
- **PIE (Position Independent Executable):** Nötig, damit auch das Hauptprogramm-Segment (Text-Segment) an zufällige Adressen geladen werden kann.
- **Schwäche:** Ein einziges *Information Leak* (Offenlegung einer Speicheradresse) kann ausreichen, um die Offsets zu berechnen und ASLR zu umgehen.

**Guard Pages** Nicht-zugreifbare Speicherseiten (`PROT_NONE`) werden zwischen Speicherbereichen platziert. Ein Zugriff (Überlauf) löst einen `SIGSEGV` (Segfault) aus.

### 1.3.4 Obfuscation

Verschleierung des Codes, um Reverse Engineering zu erschweren (z.,B. Entfernen von Symboltabellen/Stripping, Umbenennen von Funktionen). Dies ist jedoch nur "Security by Obscurity".

## 1.4 Fortgeschrittene Angriffe

Wenn Code Injection durch NX verhindert wird, nutzen Angreifer **Code Reuse Attacks**.

### 1.4.1 ROP (Return-Oriented Programming)

Statt eigenen Code einzuschleusen, nutzt der Angreifer vorhandene Code-Schnipsel (**Gadgets**) im Programm oder in Bibliotheken (libc).

- **Gadget:** Eine kurze Instruktionsfolge, die mit `ret` endet (z.,B. `pop rdi; ret`).
- **Funktionsweise:** Der Angreifer manipuliert den Stack so, dass er eine Kette von Rücksprungadressen legt. Jedes `ret` springt zum nächsten Gadget.
- Damit lassen sich Register füllen und Systemaufrufe (z.,B. `execve("/bin/sh")`) konstruieren.

### 1.4.2 Fork-Server Brute-Force

Gegenmaßnahme gegen ASLR und Canaries bei Servern, die `fork()` nutzen.

- `fork()` erzeugt eine exakte Kopie des Elternprozesses (identisches Speicherlayout, identischer Canary).
- Der Angreifer kann den Canary Byte für Byte erraten:
  1. Rate Byte 1.
  2. Crash? → Falsch, neuer Fork, nächster Versuch.
  3. Kein Crash? → Richtig, weiter zu Byte 2.
- Da der Canary bei einem Crash im Kindprozess nicht neu generiert wird (da der Elternprozess weiterläuft und neu forkt), ist Brute-Force möglich.

## 1.5 Hardware-gestützte Sicherheit (CFI)

*Control-Flow Integrity (CFI)* soll sicherstellen, dass der Kontrollfluss nur vorgegebene Pfade nimmt. Hardware-Lösungen wie **Intel CET** (Control-flow Enforcement Technology) bieten effizienten Schutz.

### 1.5.1 Shadow Stack (SHSTK)

Ein zweiter, isolierter Stack, der nur Rücksprungadressen speichert.

- Bei `CALL`: Rücksprungadresse kommt auf den normalen Stack UND den Shadow Stack.
- Bei `RET`: Der Prozessor vergleicht die Adressen von beiden Stacks.
- Bei Ungleichheit (durch Buffer Overflow auf dem normalen Stack) wird das Programm gestoppt.

### 1.5.2 Indirect Branch Tracking (IBT)

Schutz gegen ROP/JOP bei indirekten Sprüngen.

- Compiler markiert valide Sprungziele mit einer speziellen Instruktion (`ENDBR`).
- Ein indirekter Sprung muss auf einer `ENDBR`-Instruktion landen, sonst wirft die CPU eine Exception.

## 1.6 Erkennung von Schwachstellen

### 1.6.1 Statische Analyse

Untersuchung des Quellcodes ohne Ausführung.

- Findet Strukturfehler und bekannte Muster.
- Probleme: Viele False-Positives, "State Explosion" bei komplexen Pfaden.

### 1.6.2 Dynamische Analyse

Untersuchung während der Ausführung.

- **Sanitizer (z.,B. ASan):** Kompilierzeit-Instrumentierung, die Speicherzugriffe zur Laufzeit prüft (erkennt OOB, Use-after-Free). Kostet Performance.

### 1.6.3 Fuzzing

Automatisiertes Testen mit zufälligen oder mutierten Eingaben, um Crashes zu provozieren.

- **Orakel:** Kriterium, um Fehler zu erkennen (z.,B. Crash, oder AddressSanitizer-Meldung).
- **Coverage-guided Fuzzing:** Der Fuzzer bevorzugt Eingaben, die neue Code-Pfade erreichen, um die Testabdeckung zu maximieren.