# Versuch: Buffer Overflow Dokumentation

Expertenversuch Lara Quitte

---

## Notizen Buffer Overflow:

- Mögliche Ursachen:
  - zu große Datenmenge, zu kleiner reservierter Speicher (Puffer/Stack) => nebenliegende Speicherstelle wird überschrieben
  - Überlauf (bzw. Fehlerhafte Berechnung) der Zieladresse, die Ziel für Datensatz im Puffer anzeigt (=pointer overflow)
  - Programmiersprachen, die keine Möglichkeit bieten, Speicherbereichsgrenzen automatisch zu überwachen, um Überschreitungen zu verhindern (z.B. C, C++)
- Gefahren:
  - Absturz des Programms
  - Verfälschung von Daten
  - Beschädigung von Datenstrukturen der Laufzeitumgebung des Programms => Rücksprungadresse des Unterprogramms kann mit beliebigen Daten überschrieben werden (Angriffe möglich)
  - von-Neumann-Architektur für Computersysteme: Daten und Programm im gleichen Speicher (Hardwarenähe) => nur unter assemblierten oder kompilierten Sprachen ein Problem
- Angriffspunkte:
  - Unix-Systeme: Root-Zugang (erlaubt sämtliche Zugriffsrechte) -> über Zugang von „normalem“User ist es leichter an Root Zugang zu gelangen, daher sind auch die gefährdet (Rechteerweiterung)
  - Über jegliche Art von Netzwerken, aber auch lokal auf System verursachbar
  - Rücksprungadressen von Unterprogrammen+deren lokale Variablen werden auf Stack gespeichert (erst Adresse, dann Variablen abgelegt) -> Stack wächst bei modernen Prozessoren nach UNTEN, Beschreibung von Feldern in Var. nach OBEN => wird Feldgrenze nicht geprüft, so kann man ggf. Rücksprungadresse erreichen und modifizieren
- Gegenmaßnahmen:
  - Kurzfristig gelieferte Fehlerkorrekturen(Patches) der Hersteller
  - Protected Mode (80286 Prozessor) trennt linearen Speicher(Programm-/Daten-/Stapelspeicher) voneinander -> Zugriffschutz über Speicherverwaltung der CPU (OS muss sicherstellen, dass nur so viel Speicher zur Verfügung gestellt wird, wie auch wirklich linear vorhanden ist) -> OS/2 nutzt Speichersegmentierung
  - Verwendung von Programmiersprachen, die konzeptionell sicherer als C oder C++ sind verringert Gefahr von Pufferüberläufen
  - Einhaltung von Speicherbereichen beim compilieren oder zur Laufzeit überwachen -> Veränderung von Pointern nur nach strengen Regeln ermöglichen
  - Automatische Speicherbereinigung nur durch Laufzeitsystem
  - Überprüfung des Programmcodes (Feldgrernzenüberprüfung meist fehlerhaft, da oft nicht getestet)
  - Unterstützung durch Compiler (Prüfcode-Erzeugung bei Übersetzung) -> Einfügen einer Zufallszahl(Canary), welche bei Programmstart ermittelt wird(jedes mal untersch. Werte)->bei Unterprogrammaufruf wird sie in vorgesehenen Bereich geschrieben und bei Verlassen des Programms über Rücksprungadresse fügt Compiler Code ein, der auf die Ziffer prüft -> ist sie falsch ist der Adresse nicht zu trauen und Programm wird mit Meldung abgebrochen
  - Kopie von Rücksprungadresse unterhalb der lokalen Felder erzeugen
- Allgemeines:
  - Interpretierte Sprachen i.d.R nicht anfällig, da Speicherbereich für Daten unter Kontrolle des Interpreters (nur Fehler beim Interpreter möglich)

    - Programmiersprachen mit hohem Risiko : C und C++
    - Programmiersprachen mit niedrigerem Risiko: Pascal-Familie, wie Modula, Object Pascal oder Ada
    - Programmiersprachen mit fast ausgeschlossenem Risiko: C#, Java (Ausführung im Bytecode überwacht) -> Pufferüberläufe mit Ursache im Laufzeitsystem möglich ->Java wirft java.lang.StackOverflowError , wenn Endlos-Rekursion den Methoden-Stapel-Aufruf überläuft (logischer Fehler des Programmierers)
- Heap-Überlauf:
    - Pufferüberlauf auf Heap(dynamischer Speicher) -> Speicher wird zugewiesen, wenn Programme dynamischen Speicher anfordern
    - Werden in Puffer auf Heap Daten ohne Längenüberprüfung geschrieben und Datenmenge ist größer als Puffer => Speicherüberlauf
    - Durch überschreiben von Zeigern auf Funktionen kann beliebiger Code ausgeführt werden (FreeBSD hat Heap-Schutz)
    - Kann nur in Programmiersprachen auftreten, wo keine Längenüberprüfung stattfindet (anfällig: C, C++, Assembler, weniger anfällig: Java, Pearl)

---

## Notizen String Attacks:

- Allgemeines:
    - Leichter auffindbar als Buffer Overflow und basiert auf einfachen Techniken
    - Format Parameter: %d(decimal), %u(unsigned decimal), %x(unsigned hexadecimal), %s(string), %n(number of bytes written so far)
    - Verhalten der Format Funktion wird durch format String kontrolliert -> Funktion ruft Parameter vom Stack ab, die von format string angefordert werden
    - „Channeling Problem“ kann auftauchen, wenn zwei Typen von Informationen zu einem vereint werden(z.B. data channel & controlling channel) -> nicht selbst ein Sicherheitsproblem, machen aber Bugs exploitable
    - Kommen in C leichter und häufiger vor, als in Sprachen, die type-safe sind

- Definitionen:
    - Format String: ASCIIZ String der Text und Format Parameter (z.B. %d) enthält
- Gefahren:
    - printf(buffer) interpretiert buffer als format string und verarbeitet jede formatting instruction, die enthalten sein könnte -> richtig wäre printf(„%s“, buffer)
    - Crash Programs or execute harmful code
    - Typ 1: format string ist zum Teil userseitig geliefert -> automatische Tools entdecken diesen Typ von Vulnerabilitäten
    - Typ 2: ein teilweise oder vollständig userseitig gelieferter String wird indirekt einer format function (z.B. printf) übergeben
    - Prozesse können zerstört / abgebrochen werden (z.B. sinnvoll, wenn man Netzwerke angreift und möchte, dass eine Services nicht laufen)
    - p***rinft(„%s%s%s%s%s%s%s%s%s%s%s“);*** macht es u.U möglich von illegalen Adressen zu lesen, da %s Speicher von Adressen abbildet, die auf dem Stack liegen
    - Mehrfache Nutzung von %n (kann genutzt werden um auf Adressen im Stack zu schreiben) führt auch zu Crash
    - Stack sichtbar machen: ***printf („%08x.%08x.%08x.%08x.%08x\n");*** funktioniert, da die Funktion 5 Parameter vom Stack holt und Sie als 8-digit Hexadezimalziffern abbildet -> man kann so u.U sogar den gesamten Stack Speicher erhalten -> man erhält wichtige Infos über Programmfluss und lokale Funktionsvariablen, wodurch man korrekte Offsets erhält und exploitation möglich ist
- Gegenmaßnahmen:
    - Quasi alle UNIC Systeme fangen illegale pointer Zugriffe mit dem Kernel ab -> Prozess sendet SIGEGV signal -> Programm Abbruch
    - Vorbeugen mittels Compiler (z.B. gcc) mit Flags, wie -Wall, -Wfirmat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, -Wformat=2

- Einfaches Herausfinden von fehlerhaftem Code: abzählen der Argumente in der printf-Funktion: sind es weniger als zwei, so kann ein Missbrauch vorliegen

---

## Notizen printf( )-Funktion:

- Anfälligkeiten:
  - Mit %n lässt sich auf den Speicher schreiben
  - Ergebnisse der Funktion sind undefiniert und führen zu Problemen, wenn zu wenige oder falsche Werte übergeben werden
  - Wenn zu weniger Werte übergeben werden liest printf über das Ende des jeweiligen Stackframes und der Angreifer kann vom Stack lesen
  - Mit %8 %8 %8 %8 %8 %8 %8 wird unter Umständen die Field Separation beibehalten -> einfache Lösung sind Leerzeichen zwischen den Werten, wie z.B %7 %7 %7 %7 %7 %7 wodurch nur bis dahin formatiert wird, bis die Zahl größer wird (funktioniert analog mit Strings)