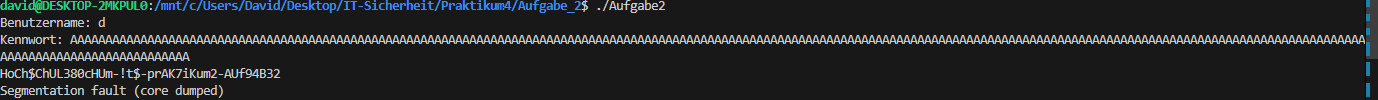
# **Aufgabe 1 – CrackMe Analyse**

### **Allgemein**

Binary: Aufgabe1  
Verwendete Tools: Ghidra (Analyse), Bash (testen) in Linux oder in Windows ein Subsystem (WSL)

### **Gefundene Flag**

Durch einen Buffer-Overflow kann man eine relevante Prüfstelle im Code überschreiben.

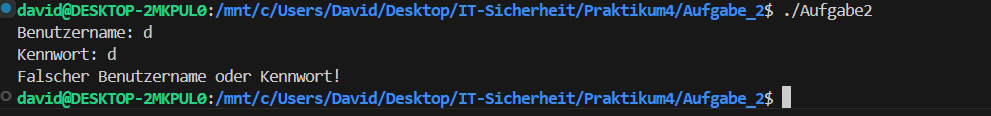


Dadurch erhält man die Flag: HoCh$ChUL380cHUm-!t$-prAK7iKum2-AUf94B32

### **Analyse des Programms – Schritt für Schritt**

**Voraussetzung**: siehe verwendete Tools

* **Vorbereitung**:
  + Datei: "Aufgabe2" wurde unter Linux mit dem Befehl:  
    chmod+x Aufgabe2
  + Testdurchlauf:



* **Binary Öffnen mit Ghidra:**
* Neues Projekt in Ghidra anlegen und Binary z.B. über Drag und Drop reinziehen => genau gleich wie bei Aufgabe1

Ein Bild, das Screenshot, Reihe, Text enthält.

KI-generierte Inhalte können fehlerhaft sein.

* Auf das Drachen-Symbol klicken, dann öffnet sich ein neues Fenster, wo die Datei auf der Registerkarte: File 🡪 open 🡪 Binary: Aufgabe1, geöffnet werden kann. Nach dem Öffnen erhält man dieses Fenster:

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

* Ein Bild, das Text, Screenshot, Schrift enthält.

  KI-generierte Inhalte können fehlerhaft sein.Dann nochmal Analyze klicken:
* Ein Bild, das Text, Schrift, Screenshot enthält.

  KI-generierte Inhalte können fehlerhaft sein.Breakpoint auf main setzen oder nach main in Assembly suchen

* Pseudocode wird rechts angezeigt

Ein Bild, das Text, Software, Screenshot, Computersymbol enthält.

KI-generierte Inhalte können fehlerhaft sein.

* **Analyze mit Ghidra**
  + Ein Bild, das Text, Screenshot, Software, Display enthält.

    KI-generierte Inhalte können fehlerhaft sein.Decompiled Code (Ghidra)
  + Ein Bild, das Text, Screenshot, Software enthält.

    KI-generierte Inhalte können fehlerhaft sein.Zur Vereinfachung hier ein Pseudocode der main methode:

### **Stack-Aufbau und Überlaufanalyse**

Im dekompilierten Code wird der Benutzername und das Kennwort jeweils über **std::cin >>** eingelesen. Beide Eingaben besitzen keine Längenprüfung, wodurch ein Buffer Overflow entstehen kann.

Die Lokalen Variablen liegen auf dem Stack von niedriger zu höherer Adresse in folgender Reihenfolge:

#### **Stackaufbau**:

#################

* Rücksprungadresse

--------------------------

* local\_1c (int) 🡨 Authentifizierungsvariable

--------------------------

* local\_48 [44 Bytes] 🡨 Entschlüsselter Flag-Puffer

--------------------------

* local\_68 [32 Bytes] 🡨 Benutzername-Puffer

--------------------------

* local\_78 [16 Bytes] 🡨 Kennwort-Puffer

################# niedrigste Adresse

**Ein Bild, das Text, Screenshot, Schrift, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.Auszug aus Ghidra Kommentare im Screenshot:**

Für den Exploit verwenden wir den Kennwort-Puffer. Obwohl der Benutzer-Puffer näher an local\_1c (Authentifizierungsvariable) liegt, ist nur der Overflow über den Kennwort-Puffer nutzerbar.

**Grund ist die Reihenfolge der Eingabe:**

* Der Benutzername-Puffer kann nicht überfüllt werden, ohne dass der Rest der Eingabe im Kennwort-Puffer landet.
* Dadurch wird der Exploit wieder zerstört und der Kennwort-Puffer überschreibt anschließend wieder alle darüberliegenden Speicherbereiche inklusive der local\_1c (Authentifizierungsvariable).

### **Wie viele Bytes werden zum Überschreiben der Auth-Variable benötigt?**

Um die Authentifizierungsvariable über dem Kennwort-Puffer zu überschreiben, müssen wir:

* 16 Bytes (Kennwort-Puffer),
* 32 Bytes (Benutzername-Puffer),
* 44 Bytes (Flag-Puffer)

=> 92 Bytes befüllen, bevor wir die local\_1c (Authentifizierungsvariable) überschreiben können.

Um den int-Wert vollständig zu befüllen sind 4 Bytes notwendig. Also brauchen wir **insgesamt**: 96 Bytes.

**Für den Exploit wurde folgendes Skript genutzt:**

* python3 -c 'print("2\n" + "A"\*92 + "\x01\x00\x00\x00")' | ./Aufgabe2
* **„ITS\n“,** für den Benutzernamen, was eingegeben wird ist irrelevant, da der Speicherbereich für den Benutzernamen sowieso überschreiben wird.  
  🡪 **„\n“** simuliert die Enter-Taste
* **„A“\*92,** erzeugt 92x den Buchstaben A und füllt damit den gesamten Puffer
* **„\x01\x00\x00\x00“ (Little Endian),** überschreibt die Authentifizierungsvariable mit dem Wert: 1

**TIPP:** Hier der Tipp im Prinzip ist egal mit welcher Zahl wir überschreiben, die Zahl muss nur größer 1 sein!  
  
**🡪 Little Endian:** Schreibweise, kleinster Byte kommt zuerst im Speicher. Standard auf fast allen modernen (x86, x64) PCs

* **| ./ Aufgabe2,** leitet die komplette Ausgabe des Python-Skripts direkt an das Programm Aufgabe2 weiter

### **Flag Ausgabe:**

* Flag: HoCh$ChUL380cHUm-!t$-prAK7iKum2-AUf94B32

### **Schutzmaßnahmen gegen Buffer Overflows**

Moderne Systeme verwenden mehrere Techniken, um Buffer Overflows zu verhindern:

**Stack Canaries:** Vor der Rücksprungadresse wird ein „Canary-Wert“ gespeichert. Wenn dieser Wert durch einen Overflow verändert wird, beendet sich das Programm.

**ASLR (Adress Space Layout Randomization):** Zufällige Adressvergabe für Stack, Heap und Libraries erschwert gezielte Angriffe.

**DEP/NX (Data Execution Prevention):** Der Stack wird als nicht ausführbar markiert, wodurch z.B. Shellcode nicht ausgeführt werden kann.

**Sichere Eingabefunktionen:** Statt **cin** sollte **std::getline()** oder **fgets()** mit Längenbegrenzung verwendet werden!

### **Demonstration verschiedener Payload-Längen und deren Auswirkung**

**Eingabe-Länge Wirkung**

16 Bytes Kennwort-Puffer vollständig gefüllt, kein Overflow

32 Bytes Zusätzlich: Beginn der Überschreibung des Benutzernamen-Puffers

60 Bytes Kennwort-Puffer Benutzernamen-Puffers überschrieben, Flag-Puffer beginnt

92 Bytes alle drei Puffer gefüllt, Auth-Variable **normal befüllt**

93–95 Bytes NUR Teile von Auth-Variable überschrieben (nicht zuverlässig)

**96 Bytes** Auth-Variable vollständig überschrieben