# **Aufgabe 1 – CrackMe Analyse**

### **Allgemein**

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

### **Gefundene Flag**

Die Flag wird beim Ausführen von >>./Aufgabe1 1268<< in der Bash ausgegeben.

Flag: hOchSChULe8OCHUM-ITS-pR4KtIkuM2-@uf9aBe1

Ein Bild, das Text, Software, Multimedia-Software, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

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

**Voraussetzung**: siehe verwendete Tools

* **Vorbereitung**:
  + Datei: "Aufgabe1" wurde unter Linux mit dem Befehl:  
    chmod+x Aufgabe1
  + Testdurchlauf:



* **Binary Öffnen mit Ghidra:**
* Neues Projekt in Ghidra anlegen und Binary z.B. über Drag und Drop reinziehen

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

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

  KI-generierte Inhalte können fehlerhaft sein.Pseudocode wird rechts angezeigt
* **Analyze mit Ghidra**
  + Decompiled Code (Ghidra)

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

KI-generierte Inhalte können fehlerhaft sein.

* + Ein Bild, das Text, Screenshot enthält.

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

### Erklärung der Validierungslogik der Binary

Das Programm erwartet als Eingabe eine Zahl, die über die **main-(…., char \*argv[])** übergeben wird. Diese wird mithilfe von **atoi()** in eine Ganzzahl umgewandelt. Danach wird dieser Wert genau mit **0x45 🡪 69** zur Eingabe addiert. Das Ergebnis wird mit **0x539 🡪 1337** verglichen. Wenn **input + 69 == 1337** wahr ist, dann wird eine Funktion **str\_decrypt()** aufgerufen. Diese Funktion entschlüsselt ein internes, verschlüsselte Byte-Array (NICHT im Klartext sichtbar), das die gesuchte Flag enthält. Nachdem dieses entschlüsselt wird, wird es über **puts()** auf der Konsole ausgegeben.

**Zusammengefasst**:

* Das Programm erwartet einen Parameter über argv[1]
* Dieser wird mit atoi() zu einem int konvertiert
* Der Wert wird um 0x45 🡪 69 erhöht
* Wenn das Ergebnis des erhöhten Inputs == 0x539 🡪 1337 ist, wird die Entschlüsselung des Bytearrays (der Flag) und diese in der Konsole über puts() ausgegeben
* Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

  KI-generierte Inhalte können fehlerhaft sein.Bei falschem Wert wird: „No!“ ausgegeben.

#### Anmerkung zur str\_decrypt()-Funktion

Die Flag liegt nicht im Klartext im Binary vor, sondern ist als verschlüsseltes Bytearray eingebettet. Dieses wird im Assembly Byte für Byte auf den Stack geschrieben. Erst wenn die Validierung des Inputs passt, wird dieses Array an die Funktion:



übergeben. Dieses entschlüsselt das Array durch eine Schleife und erzeugt die Flag zur LAUFZEIT. Die Char-Zeichenkette wird dann über puts() ausgegeben.

### Flag-Ausgabe:

Durch die Analyse ist zu erkennen, dass wir eine Zahl x ,übergeben müssen, die: **x + 69 = 1337**.  
=> **1268**



**FLAG**: hOchSChULe8OCHUM-ITS-pR4KtIkuM2-@uf9aBe1

### Mögliche Schutzmaßnahmen gegen Reverse Engineering

* Entfernen von Symbolinformationen mit strip
* Verschlüsselung des Vergleichswerts statt direktem Vergleich
* Anti-Debugging (z.B. ptrace)
* Komplexe verschachtelte Bedingungen oder XOR-Schleifen

### Zusätzlicher Hinweis:

Annäherung der Originalimplementierung ist unter export\_Aufgabe1 zu finden