

Aufgabe 1 – CrackMe Analyse

Allgemein

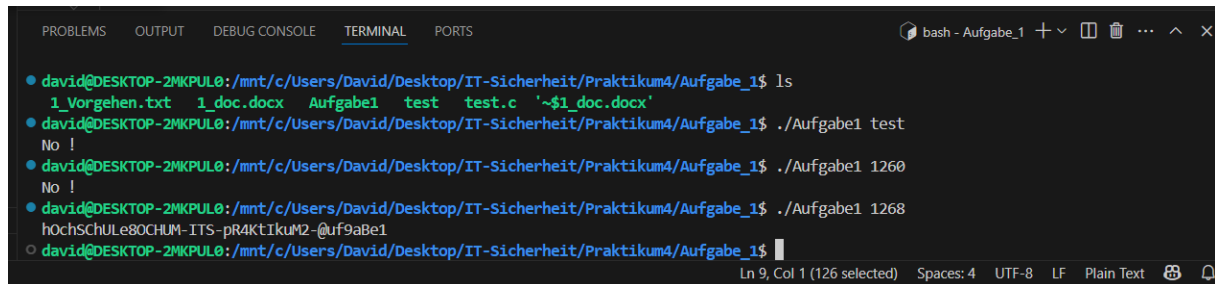
Binary: Aufgabe1

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



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash - Aufgabe_1 + - [ ] ... ^ x

• david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ls
  1_Vorgehen.txt  1_doc.docx  Aufgabe1  test  test.c  '~$1_doc.docx'
• david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ./Aufgabe1 test
No !
• david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ./Aufgabe1 1260
No !
• david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ./Aufgabe1 1268
hOchSchULE8OCHUM-ITS-pR4KtIkuM2-@uf9aBe1
• david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$
```

Analyse des Programms – Schritt für Schritt

Voraussetzung: siehe verwendete Tools

- **Vorbereitung:**

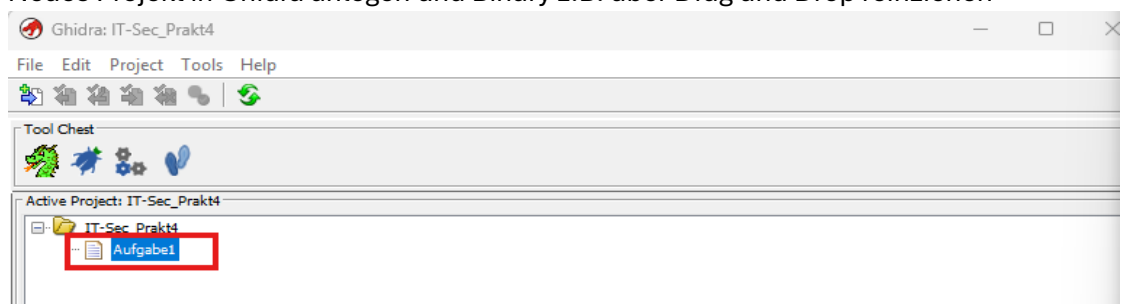
- Datei: "Aufgabe1" wurde unter Linux mit dem Befehl:
`chmod+x Aufgabe1`

- Testdurchlauf:

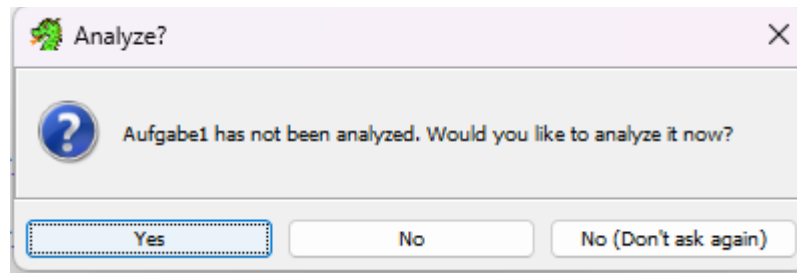
```
david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ./Aufgabe1 test
No !
```

- **Binary Öffnen mit Ghidra:**

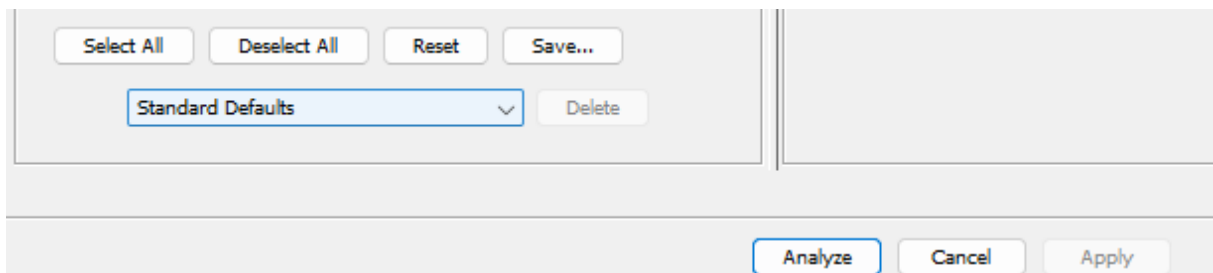
- Neues Projekt in Ghidra anlegen und Binary z.B. über Drag und Drop reinziehen



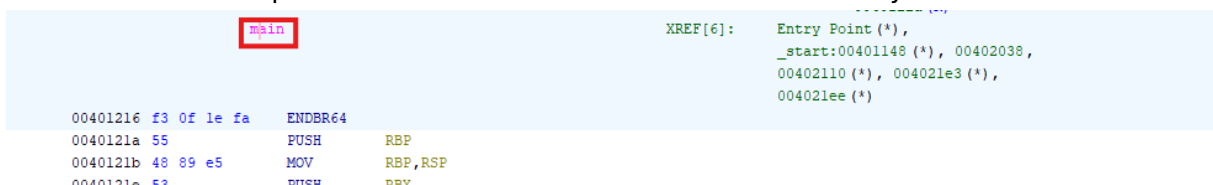
- Auf das Drachen-Symbol klicken, dann öffnet sich ein neues Fenster, wo die Datei auf der Registerkarte: File → open → Binary: Aufgabe1, geöffnet werden kann. Nach dem Öffnen erhält man dieses Fenster:



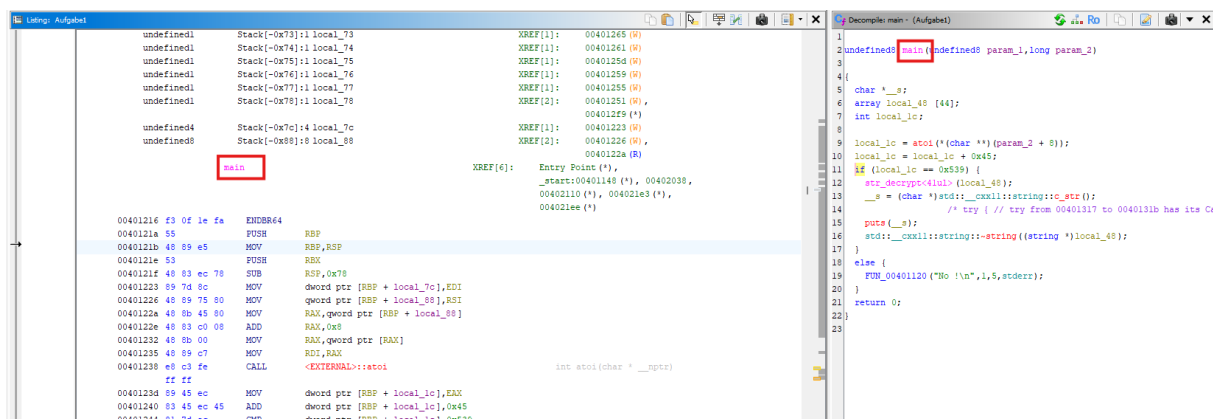
- Dann nochmal Analyze klicken:



- Breakpoint auf main setzen oder nach main in Assembly suchen



- Pseudocode wird rechts angezeigt



- **Analyse mit Ghidra**
 - Decompiled Code (Ghidra)

```

1
2 undefined8 main(undefined8 param_1,long param_2)
3
4 {
5     char *__s;
6     array local_48 [44];
7     int local_lc;
8
9     local_lc = atoi(*(char **) (param_2 + 8));
10    local_lc = local_lc + 0x45;
11    if (local_lc == 0x539) {
12        str_decrypt<4lul>(local_48);
13        __s = (char *)std::__cxx11::string::c_str();
14        /* try { // try from 00401317 to 0040131b has its Ca
15        puts(__s);
16        std::__cxx11::string::~~string((string *)local_48);
17    }
18    else {
19        FUN_00401120("No !\n",1,5,stderr);
20    }
21    return 0;
22 }
23

```

- Zur Vereinfachung hier ein Pseudocode der main methode:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      char buffer[100];
6      char *decrypted_string;
7      int input_value;
8
9      input_value = atoi(argv[1]);           // Eingabe als Zahl interpretieren
10     input_value = input_value + 69;        // Konstante Addition wie im Assembly
11     if (input_value == 1337) {             // Vergleich mit Zielwert
12         str_decrypt(buffer);               // verschlüsseltes Array wird entschlüsselt
13         decrypted_string = buffer;
14         puts(decrypted_string);            // Ausgabe der Flag
15     } else {
16         fprintf(stderr, "No !\n");         // Fehlermeldung bei falscher Eingabe
17     }
18
19     return 0;
20 }

```

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
- Bei falschem Wert wird: „No!“ ausgegeben.

```
00401238 e8 c3 fe      CALL    <EXTERNAL>::atoi
          ff ff
0040123d 89 45 ec      MOV     dword ptr [RBP + local_1c],EAX
00401240 83 45 ec 45    ADD     dword ptr [RBP + local_1c],0x45
00401244 81 7d ec      CMP     dword ptr [RBP + local_1c],0x539
```

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:

```
00401303 e8 70 00      CALL    LAB_00401303
          00 00
LAB_00401303 |
XREF[1]: 004021e0 (*)
string str_decrypt<4lul>(array * ...
```

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**

```
david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_1$ ./Aufgabe1 1268
hOchSchULe8OCHUM-ITS-pR4KtIkuM2-@uf9aBe1
```

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