

# Lösungsansätze für "Introduction to Reverse Engineering 1"

---

Reverse Engineering eine Kunstform. Die Kunst, einen Mechanismus Schritt für Schritt zu begreifen, um ihn am Ende so gut verstehen dass man ihn ausnutzen oder sogar umgehen kann. Konkret handelt Reverse Engineering von dem Analysieren von binären, ausführbaren Dateien. Das können Programme unterschiedlichster Programmiersprachen auf unterschiedlichsten Betriebssystemen sein. Insbesondere im Themenbereich des Reverse Engineerings lernt man daher in flexibel zu sein und schnell neue Zusammenhänge zu begreifen.

Häufig hat man in CTFs mit kompilierten Unix-Programmen zu tun. Daher wurde für diese Introduction Aufgabe ebenfalls eine ELF Datei erstellt, die es mit verschiedensten Herrangehensweisen zu analysieren gilt. In den folgenden Abschnitten verschiedene Techniken und Programme vorgestellt, die eine solche Analyse ermöglichen. Nicht alle Methoden führen zum Ziel, aber jede Methode gibt etwas über das zu analysierende Programm preis.

Der Aufbau eines binären Linux Programms würde den Rahmen dieser Einleitung sprengen. Es sei allerdings gesagt, dass das Programm in Form von Maschineninstruktionen vorliegt. Diese werden vom Prozessor verstanden und sequenziell abgearbeitet. Der Beginn eines jeden Programms bezeichnet man als `EntryPoint`, meist in der Funktion `main`.

Die nun besprochenen Programme und Methoden sind nur angeschnitten und verdienen eigentlich eigene, mehrseitige Erklärungen. Aber mit diesen grundlegenden Informationen ist es möglich, mehr über die Werkzeuge und Methoden zu erfahren und selbst zu experimentieren!

## Lerne das Programm kennen

---

Zu Beginn ist es sinnvoll das zu analysierende Programm kennen zu lernen. Das geht am einfachsten, indem man es ausführt und durch herrumspielen die Funktionalitäten kennen lernt:

```
$ ./rev1
Give me your password:
I_DONT_KNOW
Thats not the password!
```

Das Programm scheint nach einem Passwort zu fragen und beendet sich, sobald es einer fehlerhafte Passwordeingabe, in diesem Fall `I_DONT_KNOW`, erfährt. Wir müssen also das im Programm versteckte Passwort herausfinden, um an die begehrte Flag zu kommen.

Das so ermittelte Passwort muss später am Server der CSCG Plattform eingegeben werden, um die finale Flag zu erhalten.

# Statische Methoden

---

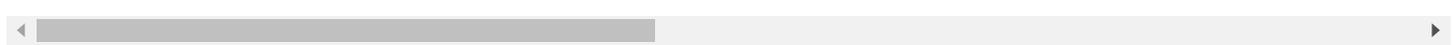
Statische Methoden eignen sich zur gründlichen, strukturierten Analyse von binären Programmen. Diese erfolgt vor allem in Form von Disassemblern und Decompilern. Disassembler machen aus den binären Maschineninstruktionen lesbaren Assembler-Code. Diese niedrigste Form einer jeden Programmiersprache reduziert das Programm auf arithmetische Operationen, Vergleiche und Sprünge innerhalb des Programms. Die Maschineninstruktionen sind dabei in der CPU-Architektur in welcher das Programm später ausgeführt wird. In der Regel ist das `x86_64`, kann aber auch `ARM`, `MIPS` oder `RISC-V` sein.

## file

Wenn man keine Ahnung hat, um was für eine Datei es sich handeln könnte, so hilft oftmals `file` weiter. Dieses kleine Programm erkennt viele verschiedene Dateiformate und spuckt Informationen dazu aus. In einem CTF vor einiger Zeit hat `file` den entscheidenden Hinweis erbracht, indem es in einer binären Datei einen `MS DOS Bootloader` erkannte.

Ein Aufruf von `file` erfolgt über die Konsole:

```
file rev1
rev1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, inte
```



Diese Ausgabe verrät uns, dass es sich um ein `x86_64`, also 64 Bit, ELF Programm in der `x86` Architektur handelt, dessen Metadaten nicht entfernt wurden (`non stripped`).

## objdump

`objdump` ist ein oftmals vorinstalliertes Programm unter Linux Umgebungen. Es eignet sich unter anderem dazu ELF Dateien zu untersuchen und die dort enthaltenen Maschineninstruktionen zu disassemblieren. Die Maschineninstruktionen erhält man durch den Aufruf von `objdump -M intel -d rev1`.

Von den vielen Instruktionen wird man oftmals erschlagen, daher eignet sich `objdump` meist nur als grobes Hilfsmittel zur grundlegenden Orientierung. Man merkt hier sehr schnell, dass es sinnvoll ist einen richtigen Disassembler zu verwenden.

## Ghidra

Die komplette IT Security Szene war begeistert, als dieses von der NSA entwickelte Werkzeug vor einigen Jahr zur kostenlosen Verwendung freigegeben wurde. Ghidra kann verschiedenste Architekturen und Dateiformate öffnen und die dort vorhandenen Metadaten sinnvoll kombinieren.

Neben einer übersichtlichen Disassembly-Anzeige versucht Ghidra den C-Code, der zur Generierung der Maschineninstruktionen geführt hat, zu rekonstruieren. In anspruchsvolleren Aufgaben ist der

rekonstruierte C-Code allerdings oftmals fehlerhaft und verwirrender als die einzelnen Instruktionen des Programms. Daher sollte man sich nie auf die Zuverlässigkeit dieses rekonstruierten Codes verlassen.

Nach dem Laden und der anfänglichen Analyse des Programmes durch Ghidra werden die ELF Header angezeigt. Diese sind für die vorliegende Aufgabe nicht von belang. Über den Tastendruck `g` (go to) und der Eingabe von `main` kann direkt zum Entry Point gesprungen werden. In dieser Funktion beginnt das eigentliche Programm nachdem es vom sogenannten `Loader` in den Speicher geladen wurde. In der Disassembly identifiziert man verschiedene Calls und Bedingungen:

```
CALL    initialize_flag => Ließt die Flag aus dem aktuellen Verzeichnis
[...]
```

```
CALL    puts => Konsolenausgabe von "Give me your password"
[...]
```

```
CALL    read => Einlesen einer Konsoleneingabe
[...]
```

```
CALL    strcmp => Vergleich von zwei Strings, Rückgabe landet in EAX
TEST    EAX,EAX => Wenn EAX == 0
JNZ     LAB_0010093a => Springe, wenn das letzte Resultat nicht gestimmt hat
CALL    puts => Ausgabe von "Thats the right password!"
[...]
```

Die C Funktionen wie `puts`, `read` und `strcmp` können in [Referenzen](#) nachgelesen werden. Für diesen Fall ist vorallem die Funktion `strcmp` von Interesse. Man kann der Dokumentation entnehmen, dass diese Funktion `0` zurückliefert, wenn die beiden zu vergleichenden Strings exakt gleich sind. Daraus kann man den Rückschluss ziehen, dass wohl dort ein Passwort überprüft wird. Wenn die beiden Strings identisch sind, erfolgt eine Ausgabe von `Thats the right password!` und die Flag wird ausgegeben.

Das Passwort selbst ist direkt in der C-Repräsentation, dem sogenannten Dekompilat, zu lesen. Vor dem Aufruf der `strcmp` Funktion wird der Speicher, der auf das Passwort zeigt aber auch einige Instruktionen vor dem `strcmp` in das Register `RDI` geladen. Dies entspricht dem ersten Argument in der `x86_64` calling convention und somit dem ersten Parameter der `strcmp` Funktion.

Diesen String zu finden ist dem Leser überlassen.

## Strings

Strings sind Repräsentationen von ASCII Zeichen, also Zahlenwerten die in einem bestimmten Bereich liegen. Ein `A` entspricht beispielsweise der Nummer `0x41 = 65` im Dezimalsystem. Mit der Zeit kennt man den ASCII Bereich und ist in der Lage, ASCII in Zahlenwerten zu erkennen. Bis dahin hilft allerdings eine [ASCII Tabelle](#). Das Programm `strings` versucht, möglichst viele ASCII Werte aus binären Daten zu lesen. Wenn also viele Zahlenwerte hintereinander in den darstellbaren ASCII-Bereich fallen, werden diese als String erkannt und ausgegeben.

Dieses kleine Hilfsprogramm ist meist vorinstalliert und kann über `strings rev1` aufgerufen werden:

```
Give me your password:
*****

Give me your password:
***** [REDACTED PASSWORD] *****

Thats the right password!
Flag: %s
Thats not the password!
./flag.txt
flag.txt
File "%s" not found. If this happens on remote, report to an admin. Otherwise, please
;*3$"
GCC: (Debian 8.3.0-6) 8.3.0

[...]
strcmp@@GLIBC_2.2.5
[...]
main
fopen@@GLIBC_2.2.5
initialize_flag
```

Strings findet viele Informationen im zu analysierenden Programm, z.B. den verwendeten Compiler `gcc 7.4.0`, die bereits bekannten Konsolenausgaben und im Programm registrierte Funktionen wie `strcmp` und `initialize_flag`. Da das zu erratenden Passwort ebenfalls über `strings` zu finden und wieder dem Leser zur Übung überlassen. In der obigen Ausgabe wurde das Passwort durch den String `***** [REDACTED PASSWORD] *****` ersetzt.

## Dynamische Methoden

Oftmals berechnen Programme aufwändige Prüfsummen oder entschlüsseln zur Laufzeit ihren Programmcode. Diese Tricks, die einem Reverse Engineer das Leben schwer machen sollten, sind durch statische Methoden nur mühsam zu erfassen. Es würde also helfen, einzelne Funktionen sowie deren Parameter und Rückgabewerte zur Laufzeit beobachten zu können.

### ltrace und strace

Ein Trace-Programm beobachtet den Programmfluss und speichert währenddessen die aufgerufenen Funktionen. Dabei hängt sich der Tracer in der Regel zwischen das auszuführende Programm und das Betriebssystem. `ltrace` beobachtet die `usercalls`, also die schon bekannten C-Funktionen wie `puts` und `strcmp`, während `strace` die `syscalls` mitschneidet. `syscalls` sind direkte Interaktionen mit dem Betriebssystem, z.B. um eine Konsolenausgabe zu machen.

Für das vorliegende Programm sind die Usercalls und damit `ltrace` interessanter. Eine Ausführung von `ltrace ./rev1` schneidet alle bekannten C Funktionen mit und gibt deren Aufruf, Parameter und Rückgabewert auf der Konsole aus. Neben den schon bekannten Strings der Konsolenausgabe sieht man auch den Vergleich zwischen der Nutzereingabe und dem zu findenden Passwort:

```
fopen("./flag", "r")
fread(0x5598546e2040, 256, 1, 0x559855172260)
fclose(0x559855172260)
puts("Give me your password: "Give me your password:
)                                     = 24
read(0I_DONT_KNOW
, "I_DONT_KNOW\n", 31)
strcmp("I_DONT_KNOW", "*****")
puts("Thats not the password!"Thats not the password!
)                                     = 24
+++ exited (status 0) +++
```

Zunächst wird die `flag` Datei via `fopen` eingelesen und im Arbeitsspeicher hinterlegt. Danach erfolgt der Aufruf von `puts`, gefolgt von einem `read` zum Einlesen der Konsoleneingabe. Danach findet der Vergleich zwischen dem geheimen Passwort und der Nutzereingabe statt, die einen Rückgabewert ungleich `0` liefert. Damit wird die Fehlermeldung ausgegeben. Das Mitscheiden des richtigen Passworts über `ltrace` ist wieder dem Leser überlassen.

## **gdb**

Der wohl bekannteste Debugger unter Unix ist `gdb`, die Abkürzung für `GNU Debugger`. Dieser konsolenbasierte Debugger ist sehr mächtig, aber ohne Erweiterungen insbesondere anfangs unintuitiv zu bedienen. Daher empfehle ich die Verwendung einer GDB Erweiterung wie `pwndbg` oder `gef`. Diese Plugins haben Vor- und Nachteile, jedes einzelne ermöglicht allerdings einen wesentlich verbesserten Umgang mit `gdb`. Im vorliegenden Beispiel wird `pwndbg` verwendet.

Der Debugger kann mit dem Befehl `gdb ./rev1` gestartet werden. Um das Programm auszuführen, reicht es in der Debugger-Konsole `run` einzugeben. Da keine weiteren Einstellungen getroffen wurden, folgt das Programm seinem Programmfluss bis es beendet wird.

```
pwndbg> run
Starting program: /home/theuser/Downloads/cscg20/challenges/intro_rev/deploy/rev1/rev1
Give me your password:
I_DONT_KNOW
Thats not the password!
[Inferior 1 (process 12333) exited normally]
pwndbg>
```

Spannend wird es, das Programm an der spannenden Stelle Schritt für Schritt zu verfolgen. Dazu reicht es, einen Haltepunkt (Breakpoint) am EntryPoint `main` zu setzen:

```
pwndbg> br main
Breakpoint 1 at 0x11a9
```

Der Debugger hält das Programm an, sobald eine Funktion mit einem Haltepunkt erreicht wurde. Von nun an können in Ruhe die Werte der Register inspeziert werden und die Ausführung in einzelnen Maschineninstruktionen fortgesetzt werden. Mit `n` wird die nächste Instruktion ausgeführt. Vor jedem Funktionsaufruf zeigt `pwndbg`, sofern die Funktion bekannt ist, die übergebenen Parameter an. Beispielsweise beim ersten `puts` Aufruf nach der siebenfachen Eingabe von `n`:

```
0x555555551a9 <main+4>    sub    rsp, 0x30
0x555555551ad <main+8>    mov    eax, 0
0x555555551b2 <main+13>   call   initialize_flag          <initialize_flag>

0x555555551b7 <main+18>   lea   rdi, [rip + 0xe4a]
► 0x555555551be <main+25>   call   puts@plt                <puts@plt>
    s: 0x555555556008 ← 'Give me your password: '

0x555555551c3 <main+30>   lea   rax, [rbp - 0x30]
0x555555551c7 <main+34>   mov   edx, 0x1f
0x555555551cc <main+39>   mov   rsi, rax
0x555555551cf <main+42>   mov   edi, 0
0x555555551d4 <main+47>   call   read@plt                <read@plt>
```

Auf diesem Weg kann die `strcmp` Funktion erreicht werden:

```
pwndbg>
0x0000555555551f7 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
----- [ REGISTERS
RAX 0x7fffffffef4c0 ← 'I_DONT_KNOW'
RBX 0x0
RCX 0x7ffff7ece311 (read+17) ← cmp rax, -0x1000 /* 'H=' */
RDX 0x1f
*RDI 0x7fffffffef4c0 ← 'I_DONT_KNOW'
RSI 0x555555556020 ← 'm4gic_passw0rd'
R8 0x3
R9 0x77
R10 0x0
R11 0x246
R12 0x555555550c0 (_start) ← xor ebp, ebp
R13 0x7fffffffef5d0 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffef4f0 → 0x555555552b0 (__libc_csu_init) ← push r15
```

```
RSP 0x7fffffffef4c0 ← 'I_DONT_KNOW'
*RIP 0x555555551f7 (main+82) ← call 0x55555555080
```

[ D

```

0x555555551e2 <main+61>    cdqe
0x555555551e4 <main+63>    mov     byte ptr [rbp + rax - 0x30], 0
0x555555551e9 <main+68>    lea   rax, [rbp - 0x30]
0x555555551ed <main+72>    lea   rsi, [rip + 0xe2c]
0x555555551f4 <main+79>    mov   rdi, rax
▶ 0x555555551f7 <main+82>    call  strcmp@plt                <strcmp@plt>
    s1: 0x7fffffffef4c0 ← 'I_DONT_KNOW'
    s2: 0x555555556020 ← '*****'

0x555555551fc <main+87>    test  eax, eax
0x555555551fe <main+89>    jne  main+129                    <main+129>
```

Sowohl in den angezeigten Parameter, aber auch in den entsprechenden Registern RDI und RSI kann das gesuchte Passwort ausgelesen werden.

## Abschluss

Wie in den letzten Abschnitten gezeigt kann in dieser sehr einfachen Aufgabe das Passwort auf verschiedene Weisen gefunden werden. Jedes der Tools verfolgt dabei einen anderen Ansatz und ermöglicht es, aus den binären 1en und 0en sinnvolle Einblicke zu geben.

Schwere Reverse Engineering Aufgaben treffen aktive Gegenmaßnahmen um solche Techniken zu verhindern: Strings werden häufig "verschlüsselt" abgespeichert, dass sie nicht als ASCII zu erkennen sind. Es gibt Programmierertechniken, die das Programm sofort beendenn sobald ein aktiver Debugger erkannt wurde. So entsteht ein Katz-und-Maus-Spiel zwischen dem Reverser und dem Programmierer, der das Reverse Engineering verhindern will. In den nächsten beiden Aufgaben wurden solche ersten Maßnahmen getroffen, um das einfache Auslesen des Passworts mit zumindest einigen der hier gezeigten Techniken zu verhindern. Viel Spass! ▶