ATIS - Software Reverse Engineering

Prof. Dr. Christian Dietrich
dietrich@internet-sicherheit.de

4 Lab: Static Code Analysis

Aufgabe

Das Ziel dieser Praktikumsaufgabe besteht darin, sich mit Kontrollflussgraphen (control flow graphs, CFG) für die x86-Architektur vertraut zu machen. Dazu soll der CFG eines Programms extrahiert werden. Wir betrachten hier lediglich den CFG der Ausgangsfunktion, d.h. das Ziel einer CALL-Instruktion wird nicht weiter verfolgt. Als Eingabe dient ein Programmausschnitt in Binärform wie Sie es bereits von vorhergehenden Praktikumsaufgaben kennen. Gehen Sie davon aus, dass der Eingabecode zur Laufzeit an die Speicheradresse 0x1000 gemapped wird. Idealerweise entwickeln Sie ein Programm, beispielsweise in Python oder Java, um den CFG zu extrahieren und auszugeben.

Zur Verdeutlichung der Aufgabe sei als Beispiel folgender Assembly-Code aus der Vorlesung gegeben:

```
1  push ebp
2  mov  ebp, esp
3  sub  esp, 0x10
4  mov  dword ptr [ebp - 4], 0x1337
5  cmp  dword ptr [ebp - 4], 0x1337
6  jne  0x101b
7  call 0xffa
8  mov  eax, 0
9  leave
10 ret
```

Der resultierende CFG besteht aus 3 Basic Blocks bb1, bb2, bb3 sowie 3 Kanten $\{(\text{bb1} \to \text{bb2}), (\text{bb1} \to \text{bb3}), (\text{bb2} \to \text{bb3})\}$. Dem Ziel der CALL-Instruktion (hier 0xffa) wird nicht gefolgt. Dies sind die Basic Blocks mit den dazugehörigen Instruktionen:

```
1  bb1 at 0x1000
2      push ebp
3      mov ebp, esp
4      sub esp, 0x10
5      mov dword ptr [ebp - 4], 0x1337
6      cmp dword ptr [ebp - 4], 0x1337
7      jne 0x101b
8
9  bb2 at 0x1016
10     call 0xffa
11
12 bb3 at 0x101b
13     mov eax, 0
14     leave
15     ret
```

Darüber hinaus enthält der CFG 3 Kanten, die wie folgt ausgegeben werden sollen:

```
1  bb1 -> bb2
2  bb1 -> bb3
3  bb2 -> bb3
```

Das Ergebnis darf darüber hinaus auch gerne grafisch dargestellt werden (beispielsweise mit graphviz/dot). Für den oben genannten Beispielcode sieht die Visualisierung dazu etwa wie folgt aus:

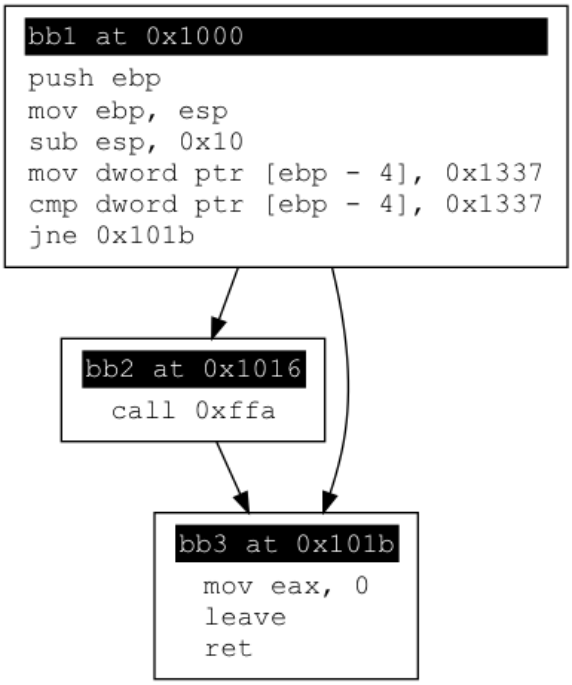

Figure 1: Visualisierung des CFG für den o.g. Beispielcode

Eingaben

- Datei mit dem Namen `whileloop1_mainfunction`
- Speicheradresse 0x1000; an diese Adresse wird der Code aus der Datei gemapped

Abzuliefernde Ergebnisse

Kontrollflussgraph (CFG) als gerichteter Graph, bestehend aus

- V: Die Menge an Basic Blocks (BB), inklusive der dazugehörigen Instruktionen pro Basic Block (eingerückt)
- E: Die Kanten zwischen den Basic Blocks

Die Ausgabe soll in Textform erfolgen, und zwar wie folgt:

```
1  bb1 at 0x1000
2      <Instruktion 1>
3      <Instruktion 2>
4      ...
5      <Instruktion n>
6
7  bb2 at 0x...
8      <Instruktion 1>
9      ...
10
11 bb3 at 0x...
12     <Instruktion 1>
13     ...
14
15 ...
16
17 bb1 -> bb2
18 ...
```

Hinweise

- Als Eingabe dient ein Programmausschnitt in Binärform, der in der Datei mit dem Namen `whileloop1_mainfunction` vorliegt. Zu diesem Code sollen Sie den CFG extrahieren. Gehen Sie davon aus, dass der Code ausschließlich x86-Instruktionen für 32-bit enthaelt.
- Versuchen Sie im ersten Schritt, den vorliegenden Binärcode disassemblen zu lassen. Dazu können Sie Code der vorhergehenden Praktikumsaufgaben wiederverwenden.
- In der Vorlesung wurde ein Verfahren zur Identifikation von Basic Blocks besprochen, das zur Lösung der Aufgabe verwendet werden kann.
- Es wird empfohlen, zunächst manuell den resultierenden CFG zu ermitteln. Daraufhin können Sie ein Programm schreiben, das den CFG extrahiert. Mit Hilfe des manuell ermittelten CFGs können Sie das Resultat Ihres Programms verifizieren.
- Die SREVM enthält den Disassembler **capstone** (`www.capstone-engine.org`), der u.a. aus Python oder Java heraus angesprochen werden kann. Sie können capstone (oder auch jeden beliebigen anderen Disassembler) zur Disassemblierung benutzen. Benutzen Sie jedoch kein Tool, das bereits fertige Funktionalität zur Extraktion eines CFG bereitstellt.
- Beachten Sie: Falls Sie auf eine CALL-Instruktion stoßen sollten, brauchen Sie dem Ziel nicht zu folgen.