UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Übungsblatt 9

Reverse Engineering: Statische Analyse

Abgabe bis 20. Juni 2023 um 12:00 Uhr im Moodle

Cybersecurity im Sommersemester 2023

Thomas Eisenbarth, Jan Wichelmann, Anja Köhl

Einführung

Oh nein! Aktuell geht irgendwie alles schief: Die Ankündigung des innovativen neuen Kontextwechsel-Abrechnungssystems ist bei einigen Kunden offenbar nicht so gut angekommen – das wäre jedenfalls eine Erklärung dafür, warum diese plötzlich zur Konkurrenz abwandern. Und jetzt ist der Implementierungsabteilung bei der Umsetzung des neuen in C programmierten Systems auch noch ein schwerer Fehler unterlaufen.

Eine unbekannte Gruppierung (unter Verdacht steht *Extremely Villainous Enterprise*, kurz *EVE*) hat dies als einzigartige Möglichkeit erkannt, das inzwischen stark in Bedrängnis geratene Unternehmen endgültig loszuwerden – und hat auf einem wichtigen Linux-Server von ITS mindestens eine Malware installiert, die sämtliche Dateien verschlüsselt hat. Um das Management von ITS weiter zu ärgern, behauptet das Programm, nach Eingabe des richtigen Passworts alle Daten wiederherzustellen. Leider ist nicht klar, wie viele Versuche für die Passworteingabe zur Verfügung stehen, weshalb ein Brute-Force-Angriff ausgeschlossen ist.

Aufgrund der auffälligen Benennung des Malware-Prozesses (`itsvirus`) ist es einigen Technikern von ITS gelungen, den Maschinencode der Malware zu extrahieren und mit `objdump` zu disassemblieren. Außerdem haben die Angreifer ihren Compiler anscheinend nicht richtig konfiguriert, sodass das disassemblierte Programm noch Funktionsnamen enthält: Unter anderem findet sich darin eine Methode `check_password`, die es nun zu analysieren gilt.

Im Moodle finden Sie den Code der Methode `check_password`, den entsprechenden Teil der `main`-Funktion und das Datensegment `.rodata` des Programms.

Zur Bearbeitung

Die nachfolgenden Aufgaben sollen Ihnen eine Hilfestellung zum Vorgehen bei einer statischen Analyse geben. Anders als bei den vorherigen Übungszetteln handelt es sich um eine Abfolge von Zwischenschritten, deren Ziel ein umfassendes Verständnis der analysierten Funktion ist. Es genügt also bei den meisten Teilaufgaben, kurze Stichpunkte als Antwort zu geben, sofern nicht explizit anders gefordert.

Weiterhin lohnt es sich oft, bereits verstandene Teile der Funktion in Pseudocode-Form (oder gar C-Code) zu notieren, um einen Überblick zu erhalten und die weitere Analyse zu vereinfachen. Dies kann sowohl separat auf einem Blatt Papier, als auch als Kommentar im Assembly-Code geschehen. Pseudo- bzw. C-Code darf gern mit der Abgabe eingereicht werden, ist aber nicht notwendig.

Weitere Hinweise:

- Eine Zeichenfolge der Länge n wird in C mit $n + 1$ Bytes dargestellt, da zum Markieren des Endes ein 0-Byte angefügt wird.
- Manche Compiler fügen `nop`-Anweisungen in den Maschinencode ein; diese führen keinerlei Aktion aus und dienen lediglich zur Performanceoptimierung, um z.B. Ziele von Jumps an 8-Byte-Speicheradressen auszurichten (sogenanntes *Alignment*). Derartige Instruktionen können hier ignoriert werden.

Aufgabe 1 Programmstruktur (muss nicht abgegeben werden)

Bevor die Analyse beginnen kann, sollte der Assembly-Code in sogenannte *Basic Blocks* (*BB*) zerlegt werden. Ein *BB* hat hierbei genau einen Endpunkt, der aus einer Jump-Anweisung (bedingt oder unbedingt) oder einer Return-Anweisung besteht. Dies gibt eine schnelle Übersicht, welche Stellen des Programms zusammenhängen und wo sich Schleifen befinden könnten.

Fügen Sie hierzu in den Assembly-Code nach jedem Sprung/Return Leerzeilen ein, um diese Blöcke zu identifizieren. Ergänzend können die Blöcke auch in einem Zeichenprogramm oder auf Papier mit Pfeilen verbunden werden.

Aufgabe 2 Aufrufkonvention I (7 Punkte)

1. Wie viele Parameter werden der Funktion `check_password` übergeben? Welche Datentypen haben diese?
2. Werden spezielle Anforderungen an die Übergabeparameter gestellt? Falls ja, nennen Sie diese, und beschreiben Sie kurz was bei Nichterfüllung der Anforderungen geschieht.
3. Welche *non-volatile* Register werden in der Funktion benutzt und müssen daher auf dem Stack gesichert werden?
4. Wie groß ist der in der Funktion für lokale Variablen reservierte Bereich auf dem Stack?
5. Welche relativen Adressen haben die auf dem Stack befindlichen lokalen Variablen (im Format `rsp+X`)? Wie groß sind sie?

   Tipp: Es lohnt sich oft, den Stack einer Funktion wie in dem Beispiel aus der Vorlesung einmal aufzumalen. Weiterhin können Sie den Variablen temporäre Namen geben (z.B. `var1`) und die entsprechenden `[rsp+X]`-Verweise im Programm durch diese ersetzen, um die Lesbarkeit zu erhöhen. Sie dürfen dies gern mit abgeben.
6. Welche möglichen Rückgabewerte gibt es?

Aufgabe 3 Aufrufkonvention II (3 Punkte)

1. Wie lautet die Formatzeichenfolge für `scanf`?
2. Wo werden die eingelesenen Werte gespeichert?
3. Welcher Rückgabewert wird von `scanf` erwartet? Was passiert, wenn dieser nicht eintritt?

Aufgabe 4 Schleife (6 Punkte)

Nach dem Aufruf von `scanf` und der Prüfung von dessen Rückgabewert folgt ein Basic Block mit einer Schleife.

1. Mit welchem Wert beginnt der Schleifenzähler, mit welchem endet er?
2. Welche Operation wird im Schleifenkörper ausgeführt?
3. Beschreiben Sie das Endergebnis nach Ausführung der Schleife, unter Beachtung der in der `main`-Funktion übergebenen Parameter.

Hinweis: `lea` (für „Load Effective Address“) berechnet den Ausdruck auf der rechten Seite und speichert das Ergebnis im Operanden auf der linken Seite; es wird kein tatsächlicher Speicherzugriff ausgeführt. Compiler benutzen diese Instruktion gern für Optimierungen, da diese mehrere Rechenschritte bündelt, für die sonst mehrere einzelne Instruktionen nötig wären. Beispiel:

```
mov rdx, 3
mov rax, 2
lea rcx, [rdx + rax * 4 + 4]
```

Nach diesen drei Befehlen hat `rcx` den Wert $3 + 2 \cdot 4 + 4 = 15$.

Aufgabe 5 if-Statements (4 Punkte)

Nach der Schleife folgen noch einige `if`-Statements, in denen das Ergebnis der Schleife verwendet wird.

1. Welche Werte werden dort verglichen?
2. Wie lautet das korrekte Passwort?