Prof. Dr. Tobias Urban
Vorlesung: IT-Sicherheit

Technology
Arts Sciences
TH Köln

Praktikum 2: „Erraten“ von Passwörtern und Passwort-Hashes in Java

In diesem Praktikum beschäftigen Sie sich mit Sicherheitsaspekten rund um Passwörter. Sie sollen zum einen die Perspektive von Angreifenden (Aufgabe 1) und zum anderen von Dienstanbietern (Aufgabe 2) einnehmen. Für beide Aufgaben wird Ihnen ein Java-Projekt zur als Grundlage zur Verfügung gestellt. Bei dem Projekt handelt es sich um ein Maven Projekt[1] mit zwei Abhängigkeiten (siehe in der `pom.xml`). Für die Lösung der Aufgabe brauchen Sie keine weiteren Bibliotheken.

Aufgabe 1: Erraten von Passwörtern

In dieser Aufgabe nehmen Sie die Position der Angreifenden ein. Ihre Aufgabe in den folgenden drei Teilaufgaben ist es jeweils das korrekte Passwort für die Nutzer:innen herauszufinden.

In der bereitgestellten Java-Klasse `PasswordCracker` befinden sich drei Methoden, in denen jeweils auf unterschiedliche Art und Weise Passwörter erraten bzw. herausgefunden werden sollen.

In der Klasse finden sie eine „login-Methode“ (`private static boolean login(String username, String password)`). Der Methode werden zwei Parameter übergeben: zum einen ein Benutzername, der für jede Aufgabe anders ist, und ein Passwort. Die Methode liefert `true` zurück, wenn das Passwort zu dem Benutzernamen passt und ansonsten `false`. Zur Lösung der Aufgabe müssen Sie das jeweils passende Passwort für den Benutzer herausfinden.

Aufgabe 1.1: Erraten eines häufig verwendeten Passwortes

Diese Teilaufgabe befasst sich mit der Methode `public static void crackSimple()`. Sie sollen das Passwort für den Benutzer `tobiasurban` herausfinden. Das Passwort ist korrekt, wenn die Methode `login("tobiasurban", "IHR PASSWORT")` `true` zurückliefet.

[1] https://maven.apache.org/

Sie wissen, dass `tobiasurban` sich nicht viel um Sicherheit kümmert und wahrscheinlich eines der am häufigsten verwendete Passwörter nutzt.

Tipp: Wenn Sie Aufgabe 1.2 lösen, hilft die Lösung auch bei dieser Aufgabe. Die Aufgabe ist aber auch eigenständig lösbar.

Aufgabe 1.2: Nutzung von Passwort-Listen

Diese Teilaufgabe befasst sich mit der Methode `public static void crackAdvanced()` Sie sollen das Passwort für den Benutzer `johndoe` herausfinden.

Der Benutzer scheint keines der am gängigsten genutzte Passwörter zu nutzen. Aber Sie haben noch zwei Passwortlisten (siehe im Ordner `resources` innerhalb des Java-Projekts), die hier helfen könnten.

Aufgabe 1.3: Brute-Forcing von Passwörtern.

Diese Teilaufgabe befasst sich mit der Methode `public static void crackBruteForce()`. Sie sollen das Passwort für die Benutzerin `martinamusterfrau` herausfinden.

Das Anwenden der Passwortlisten funktioniert hier leider nicht. Sie haben aber gesehen, wie `martinamusterfrau` ihr Passwort eingeben hat. Das Passwort scheint die folgenden Eigenschaften zu haben:

- Es ist 5 Zeichen lang und
- Es besteht nur aus Kleinbuchstaben, Zahlen und den Sonderzeichen #, *, !, ? und &. (siehe die statische Variable `ALPHABET` im Quellcode).

Aufgabe 2: Passwort-hashing in Java

In dieser Aufgabe sollen Sie mit Hilfe der Bibliothek `argon2-jvm`[2] das sichere hashen von Passwörtern in Java implementieren.

Für die Aufgabe werden Ihnen drei Java-Klassen zu Verfügung gestellt:

- Die Klasse `AbstractUser` dient als abstrakte Oberklasse und muss zur Lösung der Aufgabe nicht unbedingt angepasst werden.

[2] https://javadoc.io/doc/de.mkammerer/argon2-jvm/2.3/de/mkammerer/argon2/Argon2Factory.html

- Die Klasse `PasswordHashing` enthält die `main` Methode und zwei weitere Methoden:
    - `generateDummyUsers()` generiert drei Testnutzer und schreibt diese in eine Datei (`resources/user_db.txt`). Sie können in die Datei schauen, um zu prüfen, ob die Nutzer richtig abgespeichert werden. Stellen Sie dabei sicher, dass die Passwörter nicht im Klartext oder einer anderen leicht zu brechenden Variante abgelegt werden.
    - `testPasswordsforUsers()` prüft, ob die Passwörter passend abgelegt wurden und ob die Passwörter passend verifiziert werden können
- In der Klasse User müssen sie das hashen und verifizieren der Passwörter mittels argon2 Algorithmus umsetzen.
    - Über den Aufruf `Argon2 argon2 = Argon2Factory.create(Argon2Factory.Argon2Types type, int saltLen, int hashLen)` der Bibliothek `argon2-jvm` initialisieren sie das Objekt, welches zum Erstellen und Verifizieren der Hashwerte nötigt ist. Wählen sie für die Parameter `saltLen` und `HashLen` passende und sichere Längen. Für den Parameter `Argon2Factory.Argon2Types type` können Sie den Wert `Argon2Factory.Argon2Types.ARGON2i` verwenden, der definiert, dass das Objekt zum hashen von Passwörtern genutzt werden soll.
    - Über die Methode `hash(int iterations, int memory, int parallelism, char[] password)` des Objektes `Argon2` generieren sie die argon2 Hash-Werte. Wählen Sie auch hier passende Größen für die Parameter.
    - Die Methode `verify(java.lang.String hash, char[] password)` kann zum Verifizieren von Passwörtern genutzt werden.

Wie bei dem vorherigen Aufgabenblatt ist die Aufgabe wahrscheinlich richtig gelöst, wenn die `main` Methode fehlerfrei durchläuft.

Hashing mit Argon2-JVM gilt als sicherer aus mehreren Gründen, die auf die Design-Philosophie und spezifischen Features von Argon2 zurückzuführen sind. Argon2 ist ein fortschrittlicher Passwort-Hashing-Algorithmus, der 2015 den Password Hashing Competition (PHC) gewonnen hat. Hier sind einige der Hauptgründe, warum Argon2-JVM als sicherer betrachtet wird:

1. **Speicherhärte**: Argon2 wurde entwickelt, um Speicher-hart (memory-hard) zu sein, was bedeutet, dass es eine signifikante Menge an RAM benötigt, um effizient zu arbeiten. Dies erschwert Angriffe, insbesondere solche, die auf spezialisierter Hardware wie GPUs oder ASICs basieren. Die Speicherhärte macht es teuer, groß angelegte Brute-Force-Angriffe durchzuführen.

2. **Konfigurierbarkeit**: Argon2 bietet eine hohe Konfigurierbarkeit hinsichtlich der Speichergröße, der Zeitkosten (wie viele Iterationen durchgeführt werden) und der Parallelität (wie viele parallele Threads verwendet werden). Diese Parameter können angepasst werden, um den Algorithmus an spezifische Sicherheitsanforderungen und verfügbare Hardware-Ressourcen anzupassen.

3. **Widerstand gegen Seitenkanalangriffe**: Argon2 wurde unter Berücksichtigung von Widerstand gegen Seitenkanalangriffe wie Cache-Timing-Angriffe entworfen. Dies bedeutet, dass die Ausführung des Algorithmus weniger anfällig für solche Angriffe ist, die die Timing-Informationen nutzen, um sensible Daten wie Passwort-Hashes zu extrahieren.

4. **Angriffsspezifische Varianten**: Argon2 bietet drei Varianten: Argon2d, Argon2i und Argon2id.
   - **Argon2d** (data-dependent) ist resistent gegen Tradeoff-Angriffe und gut geeignet, wenn keine Angriffe mit großem Parallelismus erwartet werden.
   - **Argon2i** (data-independent) ist gegen Seitenkanalangriffe wie Timing-Attacken optimiert.
   - **Argon2id** kombiniert Eigenschaften von Argon2d und Argon2i und bietet einen guten Kompromiss zwischen beiden Varianten, indem es Sicherheit gegen beide Arten von Angriffen bietet.

5. **Breite Unterstützung und Implementierung**: Argon2 wird in vielen Programmiersprachen und Umgebungen unterstützt, einschließlich Java über Bibliotheken wie Argon2-JVM. Diese Implementierungen werden regelmäßig aktualisiert und überprüft, um sicherzustellen, dass sie aktuelle Sicherheitsstandards und Best Practices einhalten.

6. **Gemeinschafts- und Expertenunterstützung**: Argon2 wird von der Krypto-Community und Sicherheitsexperten weitgehend akzeptiert und unterstützt. Es gibt umfangreiche Dokumentation, Forschung und Diskussionen über seine Sicherheit und optimale Nutzung.

Durch die Verwendung von Argon2-JVM in Java-Anwendungen profitieren Entwickler von diesen Sicherheitsmerkmalen, die dazu beitragen, Passwörter und andere sensible Daten effektiv vor verschiedenen Angriffen zu schützen.