Einleitung

Dieses Programm implementiert einen Webserver, der die Verifizierung von Attestation Reports ermöglicht. Es empfängt die Reports über einen HTTP-Endpunkt, verarbeitet und verifiziert sie mithilfe des " -sev-guest/verify"-Pakets und speichert die Ergebnisse in einer SQLite-Datenbank. Des Weiteren wird das Gin-Webframework für die HTTP-Server Funktionalität genutzt.

Übersicht über die Programmstruktur

Das Programm gliedert sich in folgende Hauptkomponenten:

- **Datenbankinitialisierung und -verwaltung**: Einrichtung der SQLite-Datenbank und der benötigten Tabellen.
- **Report-Verarbeitung:** Funktionen zur Verifizierung und Speicherung der Reports.
- **HTTP-Server und Routen**: Definition der HTTP-Endpunkte und der zugehörigen Handler.
- **Fehlerbehandlung und Logging**: Protokollierung von Fehlern und wichtigen Ereignissen.
- **Sicherheit und Best Practices**: Maßnahmen zur Sicherstellung der Sicherheit und Robustheit des Programms.

Inhaltsverzeichnis

Detaillierte Beschreibung der Komponenten

1. Datenbankinitialisierung und -verwaltung

Globale Variable für die Datenbankverbindung:

```
var db *sql.DB
```

Diese Variable hält die Verbindung zur SQLite-Datenbank.

Funktion zur Initialisierung der Datenbank:

```
func initDB()
```

Öffnen der Datenbankverbindung:

- Verwendet `sql.Open`, um die SQLite-Datenbank `reports.db` zu öffnen oder zu erstellen.
- Behandelt Fehler beim Öffnen der Datenbank und beendet das Programm bei Fehlschlägen.

Überprüfen der Datenbankverbindung:

- Verwendet `db.Ping()`, um sicherzustellen, dass die Verbindung zur Datenbank erfolgreich hergestellt wurde.

Erstellen der Tabelle `reports`:

- Bereitet eine SQL-Anweisung vor, um die Tabelle zu erstellen, falls sie noch nicht existiert.
- Die Tabelle enthält folgende Spalten:
  - `id`: Primary Key (automatisch inkrementiert)
  - `report_hash`: Unique Text (um Duplikate zu vermeiden)
  - `report`: Der eigentliche Report als Text
  - `verified`: Integer (0 oder 1), gibt den Verifizierungsstatus an

Ausführen der SQL-Anweisung:

- Führt die vorbereitete Anweisung aus und behandelt mögliche Fehler.

2. Report Verarbeitung

Berechnung des SHA256-Hashs eines Reports:

```
func hashReport(report string) string
```

- Verwendet das "Crypto/sha256"-Paket, um einen Hash des Reports zu erstellen.
- Gibt den Hash als hexadezimale Zeichenkette zurück.

Überprüfung und Hinzufügen eines Reports zur Datenbank:

```
func checkAndAddReport(report string) (bool, bool, error)
```

Überprüfung auf vorhandenen Report:

- Führt eine SQL-Abfrage durch, um festzustellen, ob der Report bereits in der Datenbank vorhanden ist.
- Verwendet den Hash des Reports als eindeutigen Identifikator.

Verifizierung des Reports:

- Wenn der Report nicht existiert, wird er dekodiert und verifiziert.
- Verwendet `decodeBase64`, um den Report aus Base64 zu dekodieren.
- Nutzt `verify.RawSnpReport`, um den Report zu verifizieren.
- Setzt den Verifizierungsstatus entsprechend dem Ergebnis.

Speichern des Reports:

- Fügt den neuen Report zusammen mit seinem Hash und Verifizierungsstatus in die Datenbank ein.
- Behandelt mögliche Datenbankfehler.

Rückgabewerte:

- `verified`: Gibt an, ob der Report erfolgreich verifiziert wurde.
- `exists`: Gibt an, ob der Report bereits in der Datenbank vorhanden war.
- `error`: Enthält einen Fehler, falls einer aufgetreten ist.

Dekodierung einer Base64-kodierten Zeichenkette:

```
func decodeBase64(s string) ([]byte, error)
```

- Verwendet das "encoding/base64"-Paket, um eine Base64-kodierte Zeichenkette zu dekodieren.
- Gibt das dekodierte Byte-Array oder einen Fehler zurück.

Sichere Extraktion von Feldern aus JSON-Objekten:

```
func extractStringField(data map[string]interface{}, field string)
(string, bool)
func extractNestedMap(data map[string]interface{}, parent string)
(map[string]interface{}, bool)
```

- Diese Funktionen extrahieren sicher String-Felder oder verschachtelte Maps aus JSON-Objekten.
- Prüfen, ob das Feld existiert und vom erwarteten Typ ist.

Extraktion des `Product`-Feldes aus der Attestation:

```
func extractProductField(attestation map[string]interface{})
interface{}
```

- Versucht, das Feld `Product` aus dem attestation object zu extrahieren.
- Gibt den Wert des Feldes oder `nil` zurück, falls es nicht vorhanden ist.

3. HTTP-Server und Routen

Initialisierung des HTTP-Servers:

```
r := gin.Default()
```

- Erstellt einen neuen Gin-Router mit Standard-Middleware (Logger und Recovery).

Definition des `/verify` Endpunkts:

```
r.POST("/verify", func(c *gin.Context) { ... })
```

Empfangen der hochgeladenen Datei:

- Verwendet `c.FormFile("file")`, um die hochgeladene Datei zu erhalten.
- Gibt einen Fehler zurück, wenn keine Datei hochgeladen wurde.

Öffnen und Lesen der Datei:

- Öffnet die Datei und liest ihren gesamten Inhalt in ein Byte-Array.

Vorverarbeitung des Inhalts:

- Konvertiert den Byte-Array in einen String.
- **Korrektur des fehlerhaften `pcrs` Schlüssels:**
  - Ersetzt alle Vorkommen von `pcrs:{` durch `"pcrs":{`.
  - Dies behebt ein bekanntes Formatierungsproblem in einigen JSON-Dateien.

Parsen des JSON-Inhalts:

- Verwendet `json.Unmarshal`, um den korrigierten String in ein JSON-Objekt zu parsen.
- Gibt einen Fehler zurück, wenn das Parsen fehlschlägt.

Extraktion der erforderlichen Felder:

- Verwendet die zuvor definierten Extraktionsfunktionen, um `Source`, `Protocol` und `Attestation` zu extrahieren.
- Gibt einen Fehler zurück, wenn diese Felder fehlen.

Verarbeitung des Attestation Reports:

- Extrahiert das `Report`-Feld aus der Attestation.
- Ruft `checkAndAddReport` auf, um den Report zu verifizieren und zu speichern.
- Behandelt mögliche Fehler während der Verifizierung.

Extraktion optionaler Felder:

- Versucht, zusätzliche Felder wie `Product`, `Data`, `Runtime`, `EventLog` und `Quote` zu extrahieren.

Vorbereitung der Antwort:

- Erstellt ein Antwortobjekt mit dem Verifizierungsergebnis und den Report details.
- Gibt an, ob der Report bereits verifiziert wurde und liefert relevante Informationen zurück.

Senden der Antwort:

- Verwendet `c.JSON`, um die Antwort als JSON mit dem Statuscode 200 zu senden.

Bereitstellung statischer Dateien und der Indexseite:

```
r.Static("/static", "./static")
r.GET("/", func(c *gin.Context) { c.File("static/index.html") })
```

- Ermöglicht das Bereitstellen von statischen Dateien wie CSS oder JavaScript.
- Legt die Indexseite fest, die bei Zugriff auf die Root-URL (`/`) angezeigt wird.

Starten des Servers:

```
if err := r.Run(":3001"); err != nil { ... }
```

- Startet den HTTP-Server auf Port 3001.
- Behandelt Fehler beim Starten des Servers.

Fehlerbehandlung und Logging

Logging von Fehlern und Ereignissen:

- Verwendet das `log`-Paket, um Fehler und wichtige Informationen zu protokollieren.
- Beispiel: `log.Printf("Fehlermeldung: %v", err)`

Umfassende Fehlerbehandlung:

- Überprüft die Rückgabewerte von Funktionen und Methoden auf Fehler.
- Gibt Fehlermeldungen an den Client zurück.
- Setzt angemessene HTTP-Statuscodes (z.B. 400 für fehlerhafte Anfragen, 500 für Serverfehler).

Rückmeldung an den Client:

- Informiert den Client über den Erfolg oder Misserfolg von Operationen.
- Liefert Details zu Fehlern, wenn angemessen.

Sicherheit und Best Practices

Verwendung vorbereiteter SQL-Anweisungen:

- Verhindert SQL-Injection-Angriffe durch Verwendung von Platzhaltern in SQL-Statements.
- Beispiel: `db.Exec("INSERT INTO reports (report_hash, report, verified) VALUES (?, ?, ?)", reportHash, report, verified)`

Eingabevalidierung:

- Überprüft Benutzer- und Dateneingaben sorgfältig.
- Stellt sicher, dass erforderliche Felder vorhanden und korrekt typisiert sind.

Fehlervermeidung durch Vorverarbeitung:

- Behebt bekannte Eingabeprobleme (z.B. fehlerhafte JSON-Schlüssel) vor der Verarbeitung.

Sicherheitsrelevante Pakete und Funktionen:

- Nutzt bewährte Pakete wie `crypto/sha256` für kryptografische Operationen.
- Verwendet etablierte Bibliotheken für die Verifizierung von Attestation Reports.

Mögliche Erweiterungen

Unterstützung weiterer Berichtstypen:

- Anpassung der Parsing- und Verifizierungslogik, um Berichte anderer Anbieter oder Formate zu unterstützen.
- Implementierung von Schnittstellen oder Plug-in-Mechanismen für erweiterbare Berichtsverarbeitung.

Authentifizierung und Autorisierung:

- Implementierung von Sicherheitsmechanismen, um unbefugten Zugriff zu verhindern.
- Nutzung von Tokens oder API-Schlüsseln für den Zugriff auf den `/verify` Endpunkt.

Unterstützung weiterer Datenbanken:

- Abstraktion der Datenbankebene, um andere Datenbanksysteme zu unterstützen (z.B. PostgreSQL, MySQL).

Asynchrone Verarbeitungsmodelle:

- Einführung von Hintergrundprozessen oder Warteschlangen, um die Verifizierung großer Reports zu handhaben.

Erweiterte Fehlerberichterstattung:

- Bereitstellung detaillierterer Fehlerberichte für Entwickler und Administratoren.

Skalierbarkeit und Deployment:

- Containerisierung mit Docker zur einfachen Bereitstellung.
- Einsatz in Kubernetes-Clustern für skalierbare Anwendungen.

Fazit

Das Programm bietet eine robuste Lösung zur Verifizierung und Speicherung von Attestation Reports über einen HTTP Webserver. Durch die Verwendung von Paketen und Frameworks, sowie Fehlerbehandlung und Sicherheitsmaßnahmen ist es sowohl funktional als auch sicher. Mit möglichen Erweiterungen kann das Programm weiter an spezifische Anforderungen angepasst und skaliert werden.

GitHub Bibliotheken

Verifizieren von Attestation-Reports:
https://github.com/google/go-sev-guest

Gin Framework:
https://github.com/gin-gonic/gin

SQLite für Go:
https://github.com/mattn/go-sqlite3