Passwortsicherheit und Authentication

IT-Sicherheit

ITT-Net-IS

30. März 2025

1 Einführung

Passwörter sind nach wie vor die verbreitetste Methode zur Authentifizierung in digitalen Systemen. Trotz neuerer Technologien wie biometrischer Authentifizierung oder Zwei-Faktor-Authentifizierung bilden sie das Rückgrat der meisten Sicherheitssysteme. Daher ist ein fundiertes Verständnis von Passwortsicherheit sowohl für Endnutzer als auch für Entwickler unerlässlich.

1.1 Grundlegende Anforderungen an sichere Passwörter

Ein sicheres Passwort sollte folgende Eigenschaften aufweisen:

- **Komplexität**: Kombination aus Groß- und Kleinbuchstaben, Zahlen und Sonderzeichen
- **Länge**: Mindestens 12 Zeichen, besser noch länger
- **Einzigartigkeit**: Keine Wiederverwendung desselben Passworts für verschiedene Dienste
- **Zufälligkeit**: Keine leicht zu erratenden Muster oder persönlichen Informationen

1.2 Häufige Angriffsmethoden

Passwörter können auf verschiedene Arten kompromittiert werden:

- **Brute-Force-Angriffe**: Systematisches Durchprobieren aller möglichen Kombinationen
- **Wörterbuch-Angriffe**: Verwendung von Listen häufig genutzter Passwörter
- **Phishing**: Täuschung des Nutzers zur Preisgabe seiner Anmeldedaten
- **Keylogger**: Überwachung der Tastatureingaben
- **Credential Stuffing**: Nutzung gestohlener Anmeldedaten für andere Dienste

Notes

- **Brute-Force-Angriffe**: Ein Angriffsmuster, bei dem systematisch alle möglichen Zeichenkombinationen durchprobiert werden, bis das richtige Passwort gefunden wird. Die Wirksamkeit hängt von der Komplexität und Länge des Passworts ab.
- **Credential Stuffing**: Eine Angriffsmethode, bei der Angreifer automatisiert gestohlene Anmeldedaten (Benutzername/Passwort-Kombinationen) aus einem Datenleck für andere Websites oder Dienste ausprobieren. Basiert auf der Tatsache, dass viele Nutzer ihre Passwörter wiederverwenden.
- **Keylogger**: Schadsoftware, die Tastatureingaben aufzeichnet. Kann sowohl als Software als auch als Hardware implementiert werden, wobei die Software-Variante häufiger vorkommt.

1.3 Passwort-Manager

Passwort-Manager sind eine effektive Möglichkeit, komplexe und einzigartige Passwörter zu verwalten, ohne sie sich merken zu müssen:

- Generieren zufälliger, starker Passwörter
- Verschlüsselte Speicherung aller Passwörter
- Zugriff über ein einziges Master-Passwort
- Oft mit Browser-Integration und Mobilanwendungen

Notes

Passwort-Manager funktionieren typischerweise durch:

- Verschlüsselung der Passwortdatenbank mit einem Master-Passwort oder biometrischen Daten
- Unterstützung für Browser-Erweiterungen zur automatischen Formularausfüllung
- Cloud-Synchronisierung zwischen verschiedenen Geräten (optional)
- Passwortgeneratoren mit einstellbaren Parametern für Länge und Komplexität

Bekannte Passwort-Manager-Lösungen sind:

- Bitwarden (Open Source)
- KeePass (Open Source, lokale Speicherung)
- LastPass (kommerziell)
- 1Password (kommerziell)
- Dashlane (kommerziell)

2 Sicheres Speichern von Passwörtern in Systemen

Für Entwickler und Systemadministratoren ist die sichere Speicherung von Benutzerpasswörtern eine kritische Aufgabe. Passwörter sollten niemals im Klartext gespeichert werden.

2.1 Hashfunktionen

Hashfunktionen wandeln Eingaben beliebiger Länge in Ausgaben fester Länge um. Sie sind Einwegfunktionen, was bedeutet, dass es praktisch unmöglich ist, vom Hash auf das ursprüngliche Passwort zu schließen. Beispiele für moderne, sichere Hashfunktionen:

- bcrypt
- Argon2
- PBKDF2
- scrypt

Notes

Detaillierte Erläuterungen zu den Hashfunktionen:

- **bcrypt**: Eine adaptive Hashfunktion, die auf dem Blowfish-Cipher basiert. Der große Vorteil ist der integrierte Salt und der Kostenfaktor, der die Rechenintensität bestimmt und mit wachsender Rechenleistung erhöht werden kann. Wird seit 1999 verwendet und gilt nach wie vor als sicher.
- **Argon2**: Gewinner des Password Hashing Competition (PHC) im Jahr 2015. Argon2 bietet Schutz gegen verschiedene Angriffsarten, einschließlich Tradeoff-Attacken, brute-force und Seitenkanalangriffe. Es gibt drei Varianten: Argon2d (schneller, aber anfälliger für Seitenkanalangriffe), Argon2i (langsamer, aber resistenter gegen Seitenkanalangriffe) und Argon2id (Kombination beider Ansätze).
- **PBKDF2** (Password-Based Key Derivation Function 2): Eine einfache aber effektive Methode, die auf wiederholten Anwendungen einer kryptografischen Hash-Funktion basiert. Der Hauptnachteil besteht darin, dass sie nicht speicherintensiv ist und daher anfälliger für Hardware-beschleunigte Angriffe sein kann.
- **scrypt**: Wurde entwickelt, um speziell gegen Hardware-Angriffe resistent zu sein. Die Funktion ist sowohl rechen- als auch speicherintensiv, was Angriffe mit spezialisierter Hardware (wie ASICs oder FPGAs) erheblich erschwert.

Vergleich der Eigenschaften:

Hashfunktion	Rechenintensiv	Speicherintensiv	Parallelisierungsresistent	Salt integriert
bcrypt	Ja	Nein	Teilweise	Ja
Argon2	Ja	Ja	Ja	Ja
PBKDF2	Ja	Nein	Nein	Nein
scrypt	Ja	Ja	Teilweise	Nein

2.2 Rainbow-Table-Angriffe

Rainbow-Tables sind eine spezielle Form von vorberechneten Tabellen, die zur Beschleunigung von Passwort-Cracking-Angriffen verwendet werden. Sie stellen einen Kompromiss zwischen Rechenzeit und Speicherplatz dar.

2.2.1 Funktionsweise von Rainbow-Table-Angriffen

Bei einem Rainbow-Table-Angriff hat der Angreifer bereits vorab eine große Anzahl von möglichen Passwörtern und deren Hashes berechnet und in Tabellen organisiert. Statt bei einem Angriff jedes mögliche Passwort neu zu hashen und zu vergleichen, kann der Angreifer einfach in der Tabelle nachschlagen, welches Klartext-Passwort zu einem bestimmten Hash gehört.

```
// Vereinfachte Darstellung einer Rainbow-Table
Hash                                     | Passwort
-----------------------------------------+----------
5f4dcc3b5aa765d61d8327deb882cf99          | password
e10adc3949ba59abbe56e057f20f883e          | 123456
25d55ad283aa400af464c76d713c07ad          | 12345678
... (Millionen weiterer Einträge)         | ...
```

2.2.2 Effektivität von Rainbow-Tables

Rainbow-Tables sind besonders effektiv gegen ungesalzene Hashes, da der gleiche Hash immer zum gleichen Passwort gehört. Ein einfacher Hash wie MD5 oder SHA-1 kann mit modernen Rainbow-Tables in Sekunden bis Minuten geknackt werden, wenn das Passwort in der Tabelle enthalten ist.
Rainbow-Tables können mehrere Terabyte groß sein, decken aber oft gängige Passwortmuster bis zu einer bestimmten Länge ab. Spezialisierte Versionen existieren für bestimmte Zeichensätze oder Passwortmuster.

2.2.3 Schutz vor Rainbow-Table-Angriffen

Die primäre Verteidigung gegen Rainbow-Table-Angriffe ist das Salting, da es sicherstellt, dass selbst identische Passwörter unterschiedliche Hashes erzeugen. Ein individueller, zufälliger Salt für jedes Passwort macht vorberechnete Rainbow-Tables praktisch nutzlos, da der Angreifer für jeden einzelnen Salt eine neue Rainbow-Table erstellen müsste.
Zusätzlich erhöhen moderne, langsame Hashfunktionen (bcrypt, Argon2) die Kosten für die Erstellung von Rainbow-Tables so drastisch, dass sie für diese Algorithmen praktisch nicht mehr realisierbar sind.

2.3 Salting

Salting ist eine Technik, bei der jedem Passwort vor dem Hashen ein zufälliger Wert (der ”Salt”) hinzugefügt wird. Dies schützt vor verschiedenen Angriffsarten:

```
hashedPassword = hash(password + salt)
```

2.3.1 Vorteile von Salting:

- Verhindert, dass identische Passwörter denselben Hash erzeugen
- Schützt vor Rainbow-Table-Angriffen (vorberechnete Hash-Tabellen)
- Macht Wörterbuch-Angriffe aufwändiger

2.3.2 Best Practices für Salting:

- Individueller Salt für jedes Passwort
- Ausreichend langer Salt (mindestens 16 Bytes)
- Zufällig generierter Salt
- Salt sollte zusammen mit dem Hash gespeichert werden

2.4 Peppering

Peppering erweitert das Konzept des Saltings um eine zusätzliche Sicherheitsebene. Dabei wird ein geheimer Wert (der ”Pepper”) zum Passwort hinzugefügt, bevor es mit dem Salt gehasht wird:

```
hashedPassword = hash(password + pepper + salt)
```

2.4.1 Wichtige Unterschiede zum Salt:

- Der Pepper ist für alle Passwörter **gleich**
- Der Pepper wird **nicht** in der Datenbank gespeichert
- Der Pepper wird stattdessen in der Anwendungskonfiguration oder besser in einem Hardware-Sicherheitsmodul (HSM) gespeichert

2.4.2 Vorteile des Pepperings:

- Selbst bei einem Datenbankzugriff können die Passwörter ohne Kenntnis des Peppers nicht geknackt werden
- Fügt eine zusätzliche Sicherheitsebene hinzu, die unabhängig von der Datenbank ist
- Erhöht die Entropie der Passwörter zusätzlich

2.5 Implementierungsbeispiel in Pseudocode

```
// Registrierung eines neuen Benutzers
function registerUser(username, password):
    // Generiere einen zufälligen Salt
    salt = generateRandomBytes(16)

    // Hole den Pepper aus einer sicheren Quelle (nicht in der DB)
    pepper = getSecretPepperFromConfig()

    // Berechne den Hash mit Salt und Pepper
    hashedPassword = hash(password + pepper + salt)

    // Speichere Benutzername, Salt und Hash in der Datenbank
    storeInDatabase(username, salt, hashedPassword)


// Überprüfung beim Login
function verifyLogin(username, password):
    // Hole Salt und Hash aus der Datenbank
    (salt, storedHash) = getUserDataFromDatabase(username)

    // Hole den Pepper aus einer sicheren Quelle
    pepper = getSecretPepperFromConfig()

    // Berechne den Hash mit dem eingegebenen Passwort
    computedHash = hash(password + pepper + salt)

    // Vergleiche den berechneten Hash mit dem gespeicherten Hash
    return (computedHash == storedHash)
```

3 Weitere Sicherheitsmaßnahmen

Neben der sicheren Speicherung von Passwörtern gibt es weitere wichtige Sicherheitsmaßnahmen:

3.1 Passwort-Richtlinien

- Durchsetzung von Mindestanforderungen an Passwörter
- Regelmäßige Passwortänderungen (kontrovers, da dies oft zu schwächeren Passwörtern führt)
- Blockieren häufig verwendeter Passwörter

Notes

Die Kontroverse um regelmäßige Passwortänderungen:

- Traditionelle Sicherheitsrichtlinien forderten oft Passwortänderungen alle 30-90 Tage
- Neuere Forschungen (u.a. vom NIST) zeigen, dass erzwungene häufige Änderungen zu schwächeren Passwörtern führen
- Nutzer neigen dazu, vorhersehbare Muster bei der Änderung zu verwenden (z.B. password1, password2…)
- Moderne Empfehlungen raten zu längeren, komplexeren Passwörtern, die nur bei Verdacht auf Kompromittierung geändert werden müssen

Blockieren häufig verwendeter Passwörter:

- Verwendung von ”Blacklists” bekannter kompromittierter Passwörter (z.B. ”Have I Been Pwned”-Datenbank)
- Verhinderung der Nutzung von Passwörtern, die in Datenlecks aufgetaucht sind
- Ablehnung von offensichtlichen Passwörtern wie ”password”, ”123456”, etc.

3.2 Rate-Limiting

- Begrenzung der Anzahl von Anmeldeversuchen
- Temporäre Kontosperrung nach mehreren fehlgeschlagenen Versuchen
- Verzögerung bei fehlgeschlagenen Anmeldeversuchen

Notes

Rate-Limiting schützt vor automatisierten Angriffen und kann auf verschiedene Weisen implementiert werden:

- **Exponentielles Backoff**: Die Wartezeit zwischen erlaubten Anmeldeversuchen steigt exponentiell an (z.B. 1s, 2s, 4s, 8s…)
- **IP-basiertes Rate-Limiting**: Begrenzt die Anzahl der Anmeldeversuche von einer bestimmten IP-Adresse
- **Account-basiertes Rate-Limiting**: Begrenzt die Anzahl der Anmeldeversuche für ein bestimmtes Benutzerkonto
- **CAPTCHA**: Nach mehreren fehlgeschlagenen Versuchen muss ein CAPTCHA gelöst werden, um menschliche Benutzer von Bots zu unterscheiden

Zu beachten ist, dass Rate-Limiting allein keinen vollständigen Schutz bietet, da Angreifer ihre Angriffe über längere Zeiträume oder mehrere IPs verteilen können. Es sollte daher als Teil einer umfassenderen Sicherheitsstrategie betrachtet werden.

3.3 Zwei-Faktor-Authentifizierung (2FA)

- Zusätzliche Sicherheitsebene neben dem Passwort
- ”Etwas, das man weiß” (Passwort) + ”etwas, das man hat” (z.B. Smartphone)
- Implementierungsmöglichkeiten: SMS, Authenticator-Apps, Sicherheitsschlüssel

Notes

Die verschiedenen Implementierungsmöglichkeiten von 2FA haben unterschiedliche Sicherheitsniveaus:

- **SMS-basierte 2FA**: Am weitesten verbreitet, aber am anfälligsten für Angriffe (SIM-Swapping, SS7-Schwachstellen)
- **Authenticator-Apps**: Generieren zeitbasierte Einmalpasswörter (TOTP) und bieten besseren Schutz als SMS
- **FIDO2-Sicherheitsschlüssel**: Bieten den stärksten Schutz gegen Phishing durch kryptografische Herausforderungs-Antwort-Verfahren
- **Biometrische Verfahren**: Fingerabdruck, Gesichtserkennung - meist nur auf dem Gerät, nicht als echter zweiter Faktor

Manche Systeme bieten auch **Multi-Faktor-Authentifizierung (MFA)**, die mehr als zwei Faktoren kombiniert:

- Wissen (Passwort, PIN)
- Besitz (Smartphone, Sicherheitsschlüssel)
- Inhärenz (Biometrische Daten)
- Standort (Netzwerkstandort, GPS)

4 OAuth-Verfahren

OAuth (Open Authorization) ist ein offenes Protokoll für die sichere Autorisierung auf eine standardisierte und einfache Methode. Es bietet eine Alternative zur direkten Verwendung von Passwörtern.

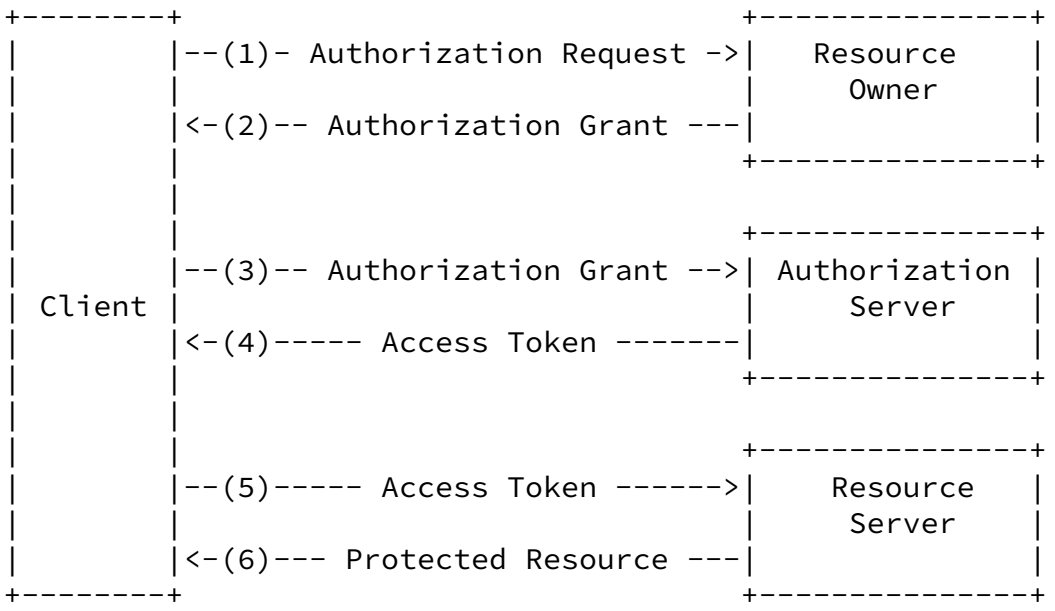
4.0.1 Grundprinzipien von OAuth

OAuth ermöglicht Benutzern, Anwendungen (Clients) einen eingeschränkten Zugriff auf ihre Ressourcen zu gewähren, ohne ihre Anmeldedaten direkt weiterzugeben. Stattdessen werden Tokens verwendet, die eingeschränkte Zugriffsrechte haben.
Die Hauptkomponenten eines OAuth-Systems sind:

- **Resource Owner**: Der Benutzer, der den Zugriff auf seine Daten genehmigt
- **Client**: Die Anwendung, die Zugriff auf die Daten des Benutzers anfordert
- **Authorization Server**: Der Server, der die Zugriffstoken ausstellt
- **Resource Server**: Der Server, der die geschützten Ressourcen hostet

4.0.2 OAuth 2.0 Ablauf

Der typische OAuth 2.0 Ablauf besteht aus folgenden Schritten:



Notes

1. Der Client fordert eine Autorisierung vom Resource Owner an
2. Der Resource Owner erteilt die Autorisierung (z.B. durch Anmeldung und Zustimmung)
3. Der Client tauscht die Autorisierung gegen ein Access Token
4. Der Authorization Server stellt ein Access Token aus
5. Der Client nutzt das Access Token, um auf geschützte Ressourcen zuzugreifen
6. Der Resource Server liefert die geschützten Ressourcen aus

4.0.3 Vorteile von OAuth

- **Keine Passweitweitergabe**: Benutzer müssen ihre Passwörter nicht an Drittanbieter-Apps weitergeben
- **Granulare Zugriffsrechte**: Zugriff kann auf bestimmte Aktionen oder Daten beschränkt werden
- **Widerrufbarkeit**: Zugriffsberechtigungen können jederzeit widerrufen werden, ohne das Passwort ändern zu müssen
- **Zeitliche Begrenzung**: Tokens können automatisch nach einer bestimmten Zeit ablaufen
- **Delegierte Autorisierung**: Ermöglicht serverseitige Anwendungen, im Namen eines Benutzers zu handeln

4.0.4 OAuth vs. OpenID Connect

Während OAuth 2.0 ein Autorisierungsframework ist (regelt den Zugriff auf Ressourcen), ist OpenID Connect eine Identitätsschicht auf OAuth 2.0, die Authentifizierung ermöglicht:

- **OAuth 2.0**: ”Was darfst du tun?” (Autorisierung)
- **OpenID Connect**: ”Wer bist du?” (Authentifizierung)

OpenID Connect erweitert OAuth um einen standardisierten Mechanismus zur Übermittlung von Benutzerinformationen über einen ID Token im JWT-Format (JSON Web Token).

4.0.5 Sicherheitsaspekte bei OAuth-Implementierungen

Bei der Implementierung von OAuth sollten folgende Sicherheitsaspekte beachtet werden:

- Verwendung von TLS/SSL für alle OAuth-Kommunikation
- Implementierung von CSRF-Schutz durch State-Parameter
- Validierung aller Redirect URIs
- Sichere Speicherung von Client Secrets
- Verwendung kurzer Lebensdauern für Access Tokens (mit Refresh Tokens für längeren Zugriff)
- Implementierung von PKCE (Proof Key for Code Exchange) für öffentliche Clients
- Regelmäßige Audits und Überprüfung aller erteilten Zugriffsberechtigungen

Notes

CSRF (Cross-Site Request Forgery): Ein Angriff, bei dem ein Benutzer dazu verleitet wird, ungewollt eine Aktion auf einer Website auszuführen, bei der er bereits authentifiziert ist.
State-Parameter in OAuth: Ein zufälliger String, der vom Client generiert und an den Authorization Server gesendet wird. Der Server gibt diesen Wert unverändert zurück, was dem Client ermöglicht zu überprüfen, ob die Antwort tatsächlich auf seine eigene Anfrage zurückgeht und nicht von einem Angreifer stammt.
PKCE (Proof Key for Code Exchange): Eine Erweiterung von OAuth 2.0, die speziell für öffentliche Clients (wie mobile Apps) entwickelt wurde, die Client Secrets nicht sicher speichern können. PKCE verhindert Authorization Code Interception-Angriffe durch einen zusätzlichen kryptografischen Prüfschritt.

5 Fazit

Die Sicherheit von Passwörtern bleibt ein zentrales Element der Cybersicherheit. Durch die Kombination von benutzerfreundlichen Praktiken, wie die Verwendung von Passwort-Managern, und technischen Sicherheitsmaßnahmen wie Salting und Peppering können sowohl Benutzer als auch Entwickler dazu beitragen, Systeme besser zu schützen.
Die wichtigsten Punkte im Überblick:

- Für Benutzer: Starke, einzigartige Passwörter verwenden und Passwort-Manager nutzen
- Für Entwickler: Niemals Klartext-Passwörter speichern, moderne Hashing-Algorithmen mit Salt und Pepper einsetzen
- Für alle Systeme: Zusätzliche Schutzmaßnahmen wie 2FA implementieren
- Als Alternative zu passwortbasierten Authentifizierungsmethoden: OAuth und OpenID Connect in Betracht ziehen