Authentifizierung und Autorisierung

Das Handbuch für die Webentwicklung

Kapitel 4
Das JSON Web Token

Das JSON Web Token (JWT) hat sich als Standard für die Umsetzung von Authentifizierungs- und Autorisierungsmechanismen in Webanwendungen etabliert. In kompaktem Format überträgt er Identitätsinformationen sicher zwischen Client und Server. Dieses Kapitel zeigt, wie JWTs aufgebaut sind, wie sie eingesetzt werden und worauf bei ihrer sicheren Verwendung zu achten ist.

Nachdem Sie die grundlegenden Konzepte von Authentifizierung und Autorisierung kennengelernt haben, widmet sich dieses Kapitel einem der derzeit am weitesten verbreiteten technischen Mechanismen zur Umsetzung dieser Konzepte: dem *JSON Web Token (JWT)*.

JWTs ermöglichen es, Identitäts- und Berechtigungsinformationen effizient, sicher und zustandslos zwischen authentifizierenden und autorisierten Systemen auszutauschen. Dies gilt sowohl für die Client-Server-Kommunikation als auch für Server-zu-Server-Szenarien, etwa in Microservice-Architekturen, wo JWTs entlang ganzer Aufrufketten weitergegeben werden. Ihre Flexibilität und Kompaktheit machen sie besonders attraktiv für moderne Webanwendungen, Single-Page Applications und Microservice-Architekturen. Die Möglichkeit zur unabhängigen Offline-Validierung durch lokales Caching des Public Keys erweitert das Einsatzspektrum auf weitere Systeme wie Plattformen zum *Business Process Management (BPM*), die nicht permanent mit einem zentralen Autorisierungsserver verbunden sein müssen.

Daher lohnt es sich, einen genauen Blick auf den Aufbau von JWTs, ihre Einsatzmöglichkeiten und typische Fallstricke zu werfen. Neben der technischen Funktionsweise stehen dabei auch empfohlene Algorithmen, Sicherheitsaspekte und bewährte Verfahren zur Validierung im Fokus, also alles, was für den sicheren und korrekten Umgang mit JWTs in der Praxis notwendig ist.

4.1 Einführung und Verwendung eines JSON Web Tokens

Ein *JSON Web Token (JWT)* ist ein offener Standard (RFC 7519[1]), der eine kompakte und in sich geschlossene Methode zur sicheren Übermittlung von Informationen zwischen zwei Parteien in Form eines JSON-Objekts beschreibt. Die enthaltenen Daten können

1 *https://datatracker.ietf.org/doc/html/rfc7519*

dank der digitalen Signatur auf ihre Echtheit überprüft und als vertrauenswürdig eingestuft werden.

Werfen wir einen genaueren Blick auf beide Konzepte:

- **Kompakt**: Das JSON Web Token wurde speziell für Umgebungen mit begrenztem Speicherplatz entwickelt, wie etwa *HTTP-Authorization-Header* oder *URI-Query-Parameter*. Um möglichst kompakt zu bleiben, wird der gesamte Inhalt base64url-kodiert, und es kommen in der Regel kurze, prägnante Schlüssel zum Einsatz.
- **In sich geschlossen**: Der Inhalt eines JWTs, der auch als *Payload* bezeichnet wird, enthält alle wichtigen Informationen über den Nutzer und seine etwaigen Berechtigungen. Da diese Daten bereits im Token selbst mitgeliefert werden, kann der Empfänger in der Regel auf zusätzliche Datenbank- oder Service-Anfragen verzichten.

JSON Web Tokens kommen in verschiedenen Szenarien zum Einsatz, in denen es darum geht, Identitäts- oder Berechtigungsinformationen sicher und effizient zwischen Systemen zu übertragen. Im Folgenden werden zwei besonders relevante Anwendungsbereiche näher erläutert: die Authentifizierung und die Autorisierung.

4.1.1 Authentifizierung

JWTs werden häufig verwendet, um die Identität eines Benutzers nach erfolgreichem Login zu repräsentieren. Nach der Anmeldung stellt der Authentifizierungsserver ein JWT aus, das unter anderem die Benutzer-ID oder den Namen enthält. Bei jeder weiteren Anfrage wird dieses Token mitgeschickt – meist im `Authorization`-Header als Bearer-Token.

Die Anwendung oder API kann anhand des Tokens prüfen, ob der Nutzer authentifiziert ist, ohne eine neue Session verwalten oder eine Datenbankabfrage durchführen zu müssen.

4.1.2 Autorisierung

Neben der Authentifizierung ist die Autorisierung ein ebenso zentraler Anwendungsfall für JWTs. Ein JWT kann neben Identitätsinformationen auch Berechtigungen enthalten – etwa Rollen, Rechte oder Zugriffsebenen.

Typische Inhalte sind:

- *role*: z. B. `admin`, `editor`, `viewer`
- *scope*: definierte Aktionsbereiche wie `read:articles` `write:comments`
- *permissions*: benutzerdefinierte Zugriffsrechte

Die API kann diese Informationen direkt aus dem Token auslesen und auf dieser Basis entscheiden, ob eine bestimmte Anfrage erlaubt ist – ohne zusätzliche Datenbank- oder Service-Calls. Das ermöglicht eine feingranulare Zugriffskontrolle bei gleichzeitig hoher Performance und Skalierbarkeit.

4.1.3 Sicherer Informationsaustausch

JWTs sind nicht nur bei der Authentifizierung hilfreich – sie sind auch ein praktisches Format, um strukturierte Informationen sicher zwischen zwei oder mehr Parteien auszutauschen.

Da ein JWT digital signiert werden kann (z. B. mit einem privaten Schlüssel), kann der Empfänger verifizieren, dass der Inhalt tatsächlich vom angegebenen Absender stammt und auf dem Übertragungsweg nicht manipuliert wurde. Die Signatur wird auf Basis von Header und Payload berechnet – jede Änderung am Inhalt würde die Signatur ungültig machen.

Je nach Anwendungsfall kann ein JWT zusätzlich auch verschlüsselt werden, um den Inhalt vor unbefugtem Zugriff zu schützen. So eignet es sich beispielsweise auch zur Übertragung sensibler Nutzerdaten zwischen Diensten.

4.2 Anatomie eines JWT

Ein JSON Web Token (JWT) ist eine base64url-kodierte Zeichenkette, die eine Menge von sogenannten Claims in Form eines JSON-Objekts darstellt. Ein *Claim* ist ein deklaratives Datenfeld innerhalb eines JWT, das strukturierte Informationen über ein Subjekt enthält. Formal handelt es sich um ein Name-Wert-Paar, wobei der Claim-Name ein String ist und der Claim-Value ein beliebiger gültiger JSON-Wert (z. B. String, Zahl, Boolean, Array oder Objekt) sein kann.

JSON Web Tokens gibt es in zwei Ausprägungen: als signierte und als unsignierte Tokens. *Signierte JWTs* werden durch *JSON Web Signature (JWS)* kryptografisch abgesichert. Die technischen Details dieser Signierung behandele ich Abschnitt 4.4. In der Praxis sollten ausschließlich signierte JWTs verwendet werden, da nur diese die Integrität und Authentizität der enthaltenen Daten garantieren. *Unsignierte JWTs* bieten keinen Schutz gegen Manipulation und sollten nur in sehr speziellen, abgeschlossenen Szenarien mit bereits vorhandener Sicherung (z. B. beim Transport über TLS in geschlossenen Systemen) eingesetzt werden – wenn überhaupt.

Ein signiertes JWT besteht aus drei Teilen, die durch einen Punkt (`.`) getrennt sind:

- aus dem *JavaScript Object Signing and Encryption (JOSE-)Header*: Er enthält Metadaten über das Token, insbesondere den verwendeten Signaturalgorithmus.
- aus der *JWT-Payload* (auch *JWT-Claims-Set* genannt): Sie enthält die eigentlichen Claims (Aussagen über den Benutzer oder andere Informationen).
- aus der *Signatur*: Sie gewährleistet die Integrität und Authentizität des Tokens.

Ein unsigniertes JWT besteht nur aus zwei Teilen:

- aus dem JOSE-Header mit dem Algorithmus-Parameter `"alg": "none"`
- aus der JWT-Payload: Das sind die Claims.

Die Signatur fehlt, wodurch das Token keine kryptografische Sicherung bietet. Der dritte Teil (nach dem zweiten Punkt) bleibt leer, sodass ein unsigniertes JWT auf zwei Punkte endet (z. B. `eyJ...eyJ...`).

Betrachten wir zuerst ein Beispiel für einen JOSE-Header:

```
{"typ":"JWT",
"alg":"HS256"}
```

Listing 4.1: JOSE-Header

Der Header enthält in der Regel zwei Angaben: den Typ des Tokens (meist `JWT`) sowie den verwendeten Signaturalgorithmus (beispielsweise HMAC SHA256 oder RSA).

Hinweis

Da JSON-Daten je nach Plattform unterschiedlich formatiert werden können (etwa durch unterschiedliche Zeilenumbrüche oder Leerzeichen), ist es wichtig, bei der Erzeugung eines Tokens eine einheitliche Darstellung zu verwenden. Meist wird hierfür die UTF-8-Codierung genutzt, da sie standardisiert ist und eine konsistente Umwandlung der Daten in eine Zeichenkette ermöglicht.

Wird dieser Header im nächsten Schritt base64url-codiert, ergibt sich folgender Wert:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

Im nächsten Schritt folgt die JWT-Payload bzw. das *JWT-Claims-Set*, also die eigentlichen Nutzdaten des Tokens. Diese enthalten Informationen, die Claims, wie etwa den Aussteller des Tokens, ein Ablaufdatum oder anwendungsspezifische Angaben.

Ein Beispiel für ein solches Claims-Set sieht so aus:

```
{"iss": "my-identity-provider.de",
  "exp": 1182142302,
  "sub": "martina"}
```

Listing 4.2: JWT-Payload

Wenn man das komprimiert (ohne Zeilenumbrüche und unnötige Leerzeichen) und dann base64url-codiert, ergibt sich:

```
eyJpc3MiOiJteS1pZGVudGl0eS1wcm92aWRlci5kZSIsImV4cCI6MTE4MjE0MjMwMiwic3
ViIjoibWFydGluYSJ9
```

Hinweis

Die genaue Codierung hängt davon ab, wie strikt man Leerzeichen handhabt. Für JWTs empfiehlt es sich, ein minimiertes JSON zu verwenden – also ohne Leerzeichen

zwischen Keys, Doppelpunkten und Werten. Deshalb sieht der »richtige« Payload-Teil in einem echten JWT so aus:

```
{"iss":"my-identity-provider.de","exp":1182142302,"sub":"martina"}
```

Um eine JWT-Payload mit Base64url zu codieren (so, wie es in einem JWT verwendet wird), gehen Sie also wie folgt vor:

1. Wandeln Sie die JSON-Daten in eine Zeichenkette um (ohne unnötige Leerzeichen).
2. Codieren Sie die Zeichenkette mit UTF-8.
3. Codieren Sie die Zeichenkette dann mit Base64url.

Um zu guter Letzt die Signatur eines JWT zu erzeugen, werden der base64url-codierte Header und die codierte Payload zu einer Zeichenkette verbunden – wiederum getrennt durch einen Punkt. Diese Zeichenkette wird anschließend mit einem geheimen Schlüssel und dem im Header definierten Signaturalgorithmus signiert.

Wenn beispielsweise der Algorithmus HMAC SHA256 verwendet wird, erfolgt die Berechnung der Signatur so:

```
HMACSHA256(
  base64urlEncode(header) + "." +
  base64urlEncode(payload),
  secret)
```

Nach dem Berechnen der Signatur und dem anschließenden Base64url-Codieren des HMAC-Werts ergibt sich folgende codierte Signatur:

```
dBjftJeZ4CVPmk-B92K27tkbUJU1p1r_wW1g118OEjXk
```

Diese Signatur stellt sicher, dass das Token tatsächlich vom angegebenen Absender stammt und seit der Erstellung nicht manipuliert wurde. Sie dient also sowohl der Authentizität als auch der Integrität des JWT.

Wenn wir nun alle drei Teile – Header, Payload und Signatur – miteinander verbinden, erhalten wir das vollständige, kompakte JSON Web Token:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiJteS1pZGVudGl0eS1wcm92aWRlci5kZSIsImV4cCI6MTE4MjE0MjMwMiwic3
ViIjoibWFydGluYSJ9.
dBjftJeZ4CVPmk-B92K27tkbUJU1p1r_wW1g118OEjXk
```

Um zu sehen, welche Informationen in einem JSON Web Token enthalten sind, genügt es, die einzelnen Bestandteile – also Header und Payload – wieder mit Base64url zu decodieren.

Praktischerweise gibt es dafür auch benutzerfreundliche Online-Tools wie *https://jwt.ms* oder *https://jwt.io*. Dort kann man ein codiertes JWT einfach in eine Eingabemaske einfü-

gen und erhält eine übersichtliche Darstellung aller enthaltenen Claims (siehe Abbildung 4.1). Die Signatur wird dabei allerdings nicht überprüft, denn zur Validierung wäre der geheime Schlüssel erforderlich, der selbstverständlich nicht öffentlich zugänglich ist.

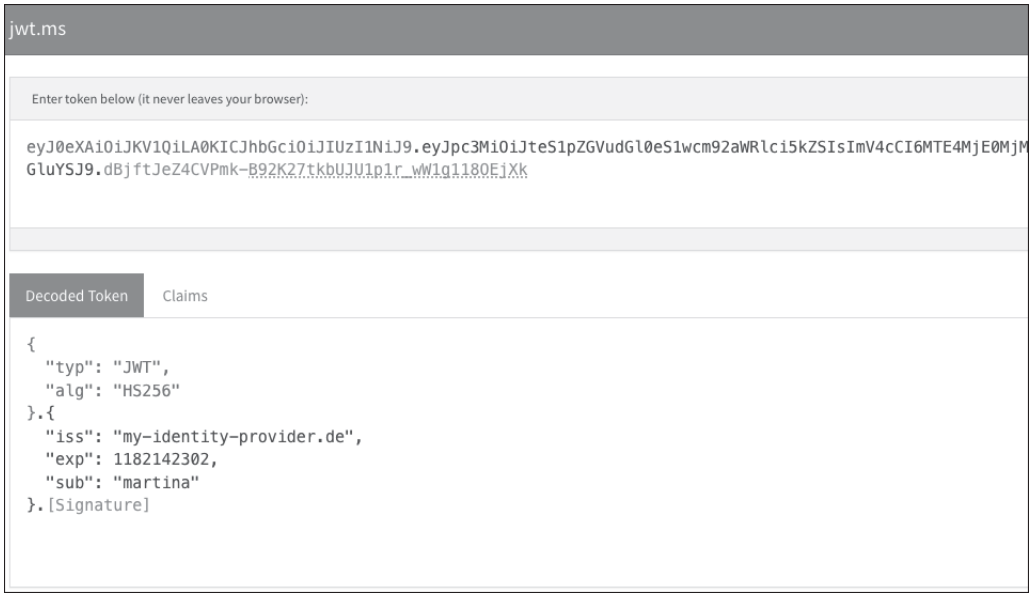

Abbildung 4.1: Das decodierte JWT

Wichtig

Verwenden Sie niemals ein echtes JSON Web Token aus Ihrer Produktionsumgebung in einem Online-Tool.

Solche Dienste sind ausschließlich dazu gedacht, Beispiel-JWTs aus der Entwicklungs- oder Testphase zu analysieren und zu veranschaulichen.

Echte Tokens enthalten oft sensible Informationen und sollten niemals öffentlich im Internet preisgegeben werden.

Für die Analyse sensibler oder produktiver JWTs sollten stattdessen Offline-Tools verwendet werden, die lokal auf dem eigenen System ausgeführt werden und keine Daten ins Internet übertragen. Ein bewährtes Beispiel ist *CyberChef*.[2]

Im Beispiel wurden bereits spezifische Claims wie `iss`, `exp` und `sub` innerhalb der Payload verwendet.

Die Claim-Namen in einem JWT-Claims-Set müssen hierbei eindeutig sein. Das bedeutet, ein JWT-Parser muss entweder Tokens mit mehrfach vorkommenden Claim-Namen ab-

2 *https://github.com/gchq/CyberChef*

lehnen oder (entsprechend der ECMAScript-Spezifikation[3]) ausschließlich den letzten Eintrag mit diesem Namen berücksichtigen.

Da ein JWT möglichst kompakt und klein gehalten sein soll, sind alle Claim-Namen bewusst verkürzt.

Werfen wir also als Nächstes einen genaueren Blick auf die vordefinierten Claim-Namen und ihre jeweilige Bedeutung.

4.3 JWT-Claims im Detail

Im Allgemeinen lassen sich JWT-Claims in drei verschiedene Kategorien unterteilen:

- **Registrierte Claims**

  Das sind standardisierte Claims mit vordefinierter Bedeutung und vordefiniertem Datentyp (z. B. `iss`, `sub`, `aud`, `exp`, `nbf`, `iat`, `jti`).

- **Öffentliche Claims**

  Sie sind nicht reserviert, aber öffentlich nutzbar und sollten über die IANA registriert werden.

- **Private Claims**

  Das sind anwendungsspezifische, individuell vereinbarte Claims wie `userId`, `role` oder `tenantId`.

Gut zu wissen: JWT-Claims

Welche Claims ein JWT enthalten muss, um als gültig zu gelten, hängt vom jeweiligen Anwendungsfall ab und wird noch spezifischer innerhalb des Kapitels zu OAuth (Kapitel 6) und Open ID Connect (Kapitel 7) beleuchtet.

Spezifische Anwendungsfälle von JWTs erfordern es, dass Implementierungen bestimmte Claims auf bestimmte Weise interpretieren und verarbeiten.

Fehlen solche Anforderungen, müssen alle Claims, die von der jeweiligen Implementierung nicht erkannt oder unterstützt werden, ignoriert werden.

4.3.1 Registrierte Claims

Registrierte Claims sind standardisierte Schlüssel-Wert-Paare innerhalb eines JSON Web Tokens (JWT), die bei der *Internet Assigned Numbers Authority* (IANA)[4] registriert und in der JWT-Spezifikation definiert sind. Sie dienen dazu, eine gemeinsame Basis für häufig

3 *https://datatracker.ietf.org/doc/html/rfc7519#ref-ECMAScript*
4 *www.iana.org/assignments/jwt/jwt.xhtml*

verwendete Angaben wie den Aussteller (`iss`), das Ablaufdatum (`exp`) oder das Subjekt (`sub`) zu schaffen.

Diese Claims sind nicht verpflichtend, fördern aber natürlich die Interoperabilität zwischen verschiedenen Systemen und Anwendungen – insbesondere bei der Integration mit Drittanbietern oder offenen Standards wie *OpenID Connect*, bei dem bestimmten registrierten Claim sogar reserviert sind. Betrachten wir einmal die wichtigsten Claims anhand von Tabelle 4.1.

Claim-Name	Beschreibung	Beispielwert
`iss` (*Issuer*)	Kennzeichnet die Entität, die das JWT ausgestellt hat. Wer hat das Token ausgestellt?	`"https://my-auth-server.com"`
`sub` (*Subject*)	Definiert den *Principal* (meist den realen oder technischen User). Besonders wichtig hierbei ist, dass das *Subject* lokal per Issuer-Kontext oder global eindeutig sein muss. Welcher Nutzer wird das Token verwenden?	`"user-1234"`

`aud` (*Audience*)	Gibt an, für welche Empfänger das JWT bestimmt ist.
	Jede Entität, die das JWT verarbeiten soll, muss sich mit einem der Werte im `aud`-Claim identifizieren. Dadurch wird die Verwendung eines JWTs deutlich eingeschränkt.
	Für wen ist das Token bestimmt (z. B. für eine URL oder für einen Identifier)?
		`"https://my-api.com"`
		oder
		`["client-1", "client-2"]`

`exp` (*Expiration Time*)	Gibt den Zeitpunkt an, ab dem das JWT nicht mehr akzeptiert werden darf. Bis wann ist das Token gültig? (Sekunden seit Unix-Epoche)	`1713888000` (entspricht z. B. 24.04.2024, 00:00 UTC)
`nbf` (*Not Before*)	Gibt den frühesten Zeitpunkt an, ab dem das JWT akzeptiert werden darf. Ab wann darf das Token verwendet werden?	`1713884400` (z. B. 23.04.2024, 23:00 UTC)

Tabelle 4.1: Registrierte Claims eines JWTs

Claim-Name	Beschreibung	Beispielwert
`iat` (*Issued At*)	Gibt an, zu welchem Zeitpunkt das JWT ausgestellt wurde. Wann wurde das Token erstellt?	`1713884400` (Zeitpunkt der Ausstellung)
`jti` (*JWT ID*)	Das ist die JWT-ID, also eine eindeutige Kennung für das Token (z. B. zur Replay-Vermeidung).	`"e8a7f52d-b5a2-4e9a-93fd-7b3c4f65d2ae"`

Tabelle 4.1: Registrierte Claims eines JWTs (Forts.)

Wie ich bereits erwähnt habe, sind alle Claims optional, außer es wird in der jeweiligen Anwendung oder Spezifikation (z. B. OIDC) anders verlangt.

Hinweis

- Zeitwerte (`exp`, `nbf`, `iat`) werden im sogenannten NumericDate-Format angegeben – also als Unix-Zeit in Sekunden.
- Werte wie `iss`, `sub`, `aud` oder `jti` sind casesensitive Strings, wobei `aud` auch ein Array sein kann.

4.3.2 Öffentliche Claims

Öffentliche Claims sind frei definierbare Claims in einem JWT, die zwar nicht offiziell registriert sind, aber dennoch über Systemgrenzen hinweg verwendet werden können.

Damit es dabei nicht zu Namenskonflikten kommt, sollten sie so gestaltet sein, dass ihre Herkunft eindeutig erkennbar ist, z. B. durch die Verwendung eines kollisionsresistenten Namensraums wie eines URI oder einer firmeneigenen Domain (z. B. *https://example.com/claim*).

Auf diese Weise lassen sich öffentliche Claims sicher und interoperabel in unterschiedlichen Anwendungen einsetzen.

4.3.3 Private Claims

Private Claims sind Claim-Namen, die zwischen dem Ersteller und dem Empfänger eines JWT individuell vereinbart werden. Sie sind nicht registriert und auch nicht global eindeutig, weshalb sie anfällig für Kollisionen sind – insbesondere, wenn Tokens in unterschiedlichen Systemen verwendet oder weitergegeben werden. Daher sollten private Claims nur mit Bedacht und innerhalb eines klar abgegrenzten Kontexts verwendet werden.

Ein gängiges Beispiel für einen privaten Claim ist der `role`-Claim, der die Rolle(n) eines Benutzers enthält:

```
{
  "sub": "user-123",
  "role": "admin"
}
```

In Kapitel 6 werde ich noch einmal auf die Validierung der einzelnen Claims zu sprechen kommen. So ist es beispielsweise wichtig, außer den zeitbezogenen Claims auch zu überprüfen, ob beispielsweise der Aussteller (der Issuer-Claim) und der Empfänger (der Audience-Claim) zusammengehören. Nur weil ein Token von einem bestimmten Aussteller stammt, heißt das nicht automatisch, dass es für diese Anwendung gedacht ist. Deshalb sollten Sie beispielsweise immer prüfen, ob

- der `iss`-Wert einem vertrauten Aussteller entspricht und ob
- die Anwendung (API, Frontend etc.) im `aud`-Claim als Empfänger genannt ist.

Alle Werte innerhalb eines JWTs sind für jeden lesbar – schließlich handelt es sich bei dem JSON-Objekt lediglich um eine base64url-kodierte Zeichenkette.

Theoretisch können diese Werte also auch manipuliert werden, etwa um eine API dazu zu bringen, Daten an einen unbefugten Empfänger auszugeben.

Wird ein JWT jedoch verändert, bricht die Signatur, da sie nicht mehr zu den geänderten Inhalten passt.

Daher sollte eine API zuerst die Signatur validieren, bevor die API die enthaltenen Claims auswertet. So lässt sich zuverlässig erkennen, ob das Token manipuliert wurde.

Allerdings ist auch das in der Praxis oft komplexer, als es auf den ersten Blick scheint. Wird nämlich ein schwacher Algorithmus zur Signaturerstellung gewählt (etwa ein unsicher konfigurierter HMAC) und wird zusätzlich ein schwacher geheimer Schlüssel verwendet, dann können moderne Tools heutzutage sowohl den Schlüssel als auch die Signatur knacken. (Ein Beispiel für solch ein Tool ist das *JWT_Tool*[5], das außer Brute-Force-Angriffen auch viele andere bekannte Angriffsvektoren automatisiert ausführen kann.)

Ist das einmal gelungen, kann ein Angreifer mit dem kompromittierten Schlüssel und Algorithmus eigene Tokens generieren und signieren. Die API erkennt diese manipulierten Tokens dann als gültig, da die Signatur technisch korrekt ist, obwohl sie nicht vom legitimen Aussteller stammt.

Im nächsten Abschnitt werfen wir daher einen genaueren Blick auf die JSON Web Signature (JWS) – und darauf, wie Sie Ihre Tokens sicher und korrekt signieren können.

5 *https://github.com/ticarpi/jwt_tool*

4.4 JSON Web Signature (JWS)

Die *JSON Web Signature (JWS)* ist ein Standard (RFC 7515[6]) aus dem JOSE-(*JavaScript Object Signing and Encryption*-)Header-Framework und bildet die Grundlage für die Integrität und Authentizität von JSON Web Tokens.

Ihr Hauptzweck besteht darin, sicherzustellen,

- dass sowohl die übermittelten Claims in der Payload als auch die Header-Parameter nicht manipuliert wurden (*Integrity*),
- dass sie tatsächlich vom angegebenen Absender – in unserem Fall der Client-Applikation – stammen (*Authentication*) und
- dass der Absender die Erstellung der Nachricht nicht abstreiten kann (*Non-repudiation*).

Eine *JSON Web Signature (JWS)* kombiniert strukturierte JSON-Daten (also unsere Claims) mit einer digitalen Signatur. Diese Signatur wird in der Regel mithilfe eines asymmetrischen Schlüsselpaars erstellt, wie es beispielsweise bei RSA oder ECDSA zum Einsatz kommt. In bestimmten Fällen kann jedoch auch eine hashbasierte Prüfsumme verwendet werden, wie sie etwa bei *Message Authentication Codes* (MACs) bzw. HMACs zum Einsatz kommt.

JWS wird vor allem bei JWTs in Authentifizierungs- und Autorisierungsprozessen verwendet, etwa bei OAuth oder OpenID Connect. Ohne eine gültige Signatur ist ein JWT nicht vertrauenswürdig – ganz gleich, welche Claims es enthält. (Daher sollte die Signatur eines Tokens stets zuerst überprüft werden, bevor dessen Claims ausgewertet oder weiterverwendet werden.)

Bei einer JSON Web Signature enthält der JOSE-Header Informationen darüber, wie das Token signiert wurde (also welcher Signaturalgorithmus verwendet wurde), und gegebenenfalls weitere Angaben zur Signatur oder zur Verarbeitung. Er beschreibt außerdem, welche Teile der Nachricht (Header und Payload) durch die Signatur geschützt sind.

In Abschnitt 4.2 haben Sie bereits ein Beispiel für den Aufbau eines JOSE-Headers gesehen. Auch im JOSE-Header gibt es registrierte Parameter, die bestimmten Zwecken dienen und – genau wie bei den Claims – eindeutig benannt sein müssen. Betrachten wir auch hier einmal die für die JSON Web Signature relevanten Parameter:

Parameter-Name	Beschreibung	Beispielwert
`alg` (*Algorithm*)	Der Parameter `alg` gibt den kryptografischen Algorithmus an, der zur Absicherung der JWS verwendet wurde.	`"HS256, RS256"`

Tabelle 4.2: Header-Parameter eines JWTs

6 *www.rfc-editor.org/rfc/rfc7515.html*

Parameter-Name	Beschreibung	Beispielwert
typ (*Type*)	Der Parameter typ gibt an, welchen Medientyp das gesamte JWS-Objekt hat (z. B. "JWT"). Dies dient Anwendungen dazu, zwischen verschiedenen Objekttypen zu unterscheiden, da mehrere in einer Struktur vorkommen könnten.	"JWT"
jku (*JWK Set URL*)	Die URL zu einem Set aus öffentlichen JSON Web Keys (JWK)	"https://example.com/keys.json"
jwk (*JSON Web Key*)	Öffentlicher Schlüssel im JWK-Format	{"kty":"RSA", "n":"...","e":"AQAB"}
kid (*Key ID*)	Key-ID zur Identifikation des verwendeten Schlüssels	"key-118"
cty (*Content Type*)	Typ der Payload (z. B. JWT als verschachtelter Inhalt)	"JWT"
crit (*Critical*)	Liste kritischer Header-Parameter, die zwingend verstanden werden müssen	["exp", "nbf"]

Tabelle 4.2: Header-Parameter eines JWTs (Forts.)

Die Parameter jku, jwk und kid dienen zur Auffindung des asymmetrischen Schlüssels. Daher muss in jedem JOSE-Header genau einer dieser Parameter vorhanden sein. Bei symmetrischen HMAC-Verfahren entfallen diese Parameter, da der gemeinsame geheime Schlüssel beiden Parteien bereits bekannt ist und aus Sicherheitsgründen nicht über JWK-Mechanismen exponiert werden darf.

Ähnlich wie bei den in Abschnitt 4.3.2 und Abschnitt 4.3.3 beschriebenen öffentlichen und privaten Claims besteht auch bei den Header-Parametern die Möglichkeit, benutzerdefinierte Parameter zu definieren – entweder als öffentliche oder als private Header-Parameter.

Im folgenden Abschnitt beschäftigen wir uns damit, wie eine JSON Web Signature (JWS) sicher erzeugt wird und welche Signaturalgorithmen sich in der Praxis bewährt haben.

4.4.1 Eine JSON Web Signature erstellen

Um eine JSON Web Signature zu erzeugen, wird eine Reihe von Verarbeitungsschritten durchlaufen.

Die genaue Reihenfolge ist dabei flexibel – solange keine Abhängigkeiten zwischen Eingaben und Ausgaben der einzelnen Schritte bestehen:

1. Erstellen Sie den Inhalt, der als JWS-Payload verwendet werden soll.
2. Kodieren Sie die Payload mit Base64url: `BASE64URL(JWS Payload)`
3. Erstellen Sie das JSON-Objekt bzw. die Objekte, die die gewünschten Header-Parameter enthalten. Diese bilden zusammen den JOSE-Header (den sogenannten *JWS Protected Header*).
4. Kodieren Sie den Protected Header: `BASE64URL(UTF8(JWS Protected Header))`
5. Nun wird die JWS-Signatur entsprechend dem verwendeten Algorithmus aus folgendem Eingabewert (der auch *JWS Signing Input* genannt wird) berechnet:

   ```
   ASCII(BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS
   Payload))
   ```

   Der Header-Parameter `alg` gibt eben diesen Algorithmus vor. Er muss daher zwingend im JOSE-Header enthalten sein und muss den tatsächlich verwendeten Algorithmus korrekt wiedergeben.
6. Kodieren Sie die Signatur mit Base64url: `BASE64URL(JWS Signature)`
7. Zuletzt wird die base64url-codierte Signatur an den in Schritt 5 erwähnten Eingabewert angehängt, um den vollständig signierten JWT zu erhalten:

   ```
   BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS
   Payload) || '.' || BASE64URL(JWS Signature)
   ```

Die Sicherheit einer Signatur steht und fällt mit dem verwendeten Algorithmus.

Deshalb ist es entscheidend zu verstehen, welche Verfahren zur Signaturbildung existieren und welche Algorithmen sich in der Praxis als sicher und empfehlenswert erwiesen haben.

Ein zentraler Bestandteil der Erstellung einer digitalen Signatur ist der Einsatz von öffentlichen und privaten Schlüsseln auf Basis bewährter kryptografischer Verfahren. Im Folgenden werfen wir daher einen genaueren Blick auf zwei der wichtigsten Vertreter: auf *RSA* und auf die *Elliptic Curve Cryptography (ECC)* – und ich zeige Ihnen, wie sie zur sicheren Signaturbildung bei JWS eingesetzt werden können.

Verfahren zur Erstellung einer digitalen Signatur

Im Zentrum der digitalen Signatur steht das Konzept eines kryptografischen Schlüsselpaares. Sie brauchen einen *privaten Schlüssel*, der geheim bleibt und zum Signieren verwendet wird, und einen dazugehörigen *öffentlichen Schlüssel*, der frei verteilt werden kann und zur Verifikation der Signatur dient.

Wie das Zusammenspiel von privatem und öffentlichem Schlüssel zur Erstellung einer Signatur funktioniert, haben wir bereits in Kapitel 2 im Zusammenhang mit der *Public-Key-Kryptografie* näher betrachtet.

Der Standard rund um die sogenannten *JSON Web Algorithms*[7] (JWA) stellt eine Vielzahl von Algorithmen bereit, die wir zur Signaturerstellung nutzen und im Header des JWT über den Parameter `alg` angeben können. Doch wie so oft gilt: Die Auswahl des richtigen Algorithmus erfordert besondere Sorgfalt. Nicht alle Bibliotheken implementieren die Spezifikation korrekt – und nicht jeder Secure-Token-Service unterstützt jeden verfügbaren Signaturalgorithmus.

Werfen wir daher einen genaueren Blick auf bewährte Empfehlungen und typische Fallstricke bei der Wahl des passenden Algorithmus. Jeder Algorithmus wird hierbei zusammen mit einer Hash-Funktion verwendet: SHA-256, SHA-384 oder SHA-512. Digitale Signaturen sind sehr rechenintensiv, besonders bei großen Datenmengen. Daher wird nicht die gesamte Nachricht, sondern nur der Hash der Nachricht signiert. Daraus folgt, dass die Größe der Payload und des Headers keinen Einfluss auf den Signaturalgorithmus haben. Der Unterschied zwischen SHA-256, SHA-384 und SHA-512 besteht im Wesentlichen in drei Punkten:

- **in der Ausgabelänge**: Je länger der Hash ist, desto schwerer ist es, Kollisionen (zwei gleiche Hashes für unterschiedliche Eingaben) zu finden oder eine Eingabe zum gewünschten Hash zu erzeugen. (SHA-256-Hashes sind hierbei 256 Bit lang, SHA-512 Hashes sind 512 Bit lang usw.)
- **im Sicherheitsniveau bezüglich der Kollisionsresistenz**: Mit der Länge der Hash-Ausgabe steigt auch die Widerstandsfähigkeit gegenüber Kollisionen. Eine Kollision würde bedeuten, dass ein Angreifer für unterschiedliche Eingaben denselben Hash-Wert erzeugen kann – mit potenziell gravierenden Folgen. Hierbei bieten SHA-256-Hashes eine Sicherheitsstufe von 128 Bit an. Ein Angreifer müsste im schlechtesten Fall 2^{128} Rechenschritte ausführen, um zwei verschiedene Eingaben zu finden, die denselben SHA-256-Hash haben – also eine Kollision erzeugen. Zum Vergleich: Bei einem SHA-512-Hash läge die Sicherheitsstufe bei 256 Bit.
- **in der internen Struktur und der Geschwindigkeit**: Alle drei Algorithmen basieren auf dem gleichen Prinzip: der SHA-2-Familie, aber:
  - SHA-384 und SHA-512 verwenden denselben internen Algorithmus, der mit 64-Bit-Wörtern arbeitet.
  - SHA-256 hingegen arbeitet mit 32-Bit-Wörtern.

  Das hat zur Folge, dass auf 64-Bit-Systemen SHA-384- und SHA-512-Hashes meist schneller ausgeführt werden können als SHA-256-Hashes, dass jedoch auf 32-Bit-Systemen SHA-256 effizienter ist.

7 *www.rfc-editor.org/rfc/rfc7518.html*

Praxisempfehlung: SHA-256

Auch wenn SHA-384 und SHA-512 auf den ersten Blick sicherer wirken, hat sich SHA-256 in der Praxis als Standard durchgesetzt. SHA-256 ist weltweit am weitesten verbreitet und wird von nahezu allen Systemen und Protokollen unterstützt. Im Vergleich dazu sind SHA-384 und SHA-512 nicht überall verfügbar und oft nicht die Voreinstellung – SHA-256 ist gewissermaßen der kleinste gemeinsame Nenner.

Zudem bietet SHA-256 mit seiner 128-Bit-Kollisionssicherheit ein Schutzniveau, das für heutige Anwendungen völlig ausreichend ist. Eine Kollision per Brute-Force anzugreifen, würde selbst bei kurzen Eingaben Milliarden Jahre dauern. Zwar liefern SHA-384 und SHA-512 theoretisch noch mehr Sicherheitsreserven, doch die sind aktuell in den allermeisten Fällen schlicht nicht erforderlich.

Das am häufigsten verwendete Signaturverfahren in der JWT-Welt ist RSA, besser bekannt unter dem Kürzel *RS256*. Dahinter steckt nichts anderes als die Kombination aus dem bewährten RSA-Algorithmus und der Hashfunktion SHA-256, wie sie im Standard für JSON Web Signatures (JWS) definiert ist.

Dabei wird zunächst aus dem JWS-Signing-Input ein SHA-256-Hash berechnet, der anschließend mit dem privaten Schlüssel nach dem Standard *RSASSA-PKCS1-v1_5*[8] signiert wird. Um RSA zur Erstellung einer digitalen Signatur für unsere JWTs nutzen zu können, müssen wir den passenden Wert in den JOSE-Header eintragen.

Zur Auswahl stehen dafür im `alg`-Parameter die Werte RS256, RS384 oder RS512, je nachdem, welche Hashfunktion verwendet werden soll.

Hinweis zur Schlüssellänge

Der Standard rund um RSA erfordert eine Schlüssellänge von mindestens 2048 Bit. Dies liegt daran, dass RSA auf der Schwierigkeit beruht, große Zahlen in ihre Primfaktoren zu zerlegen. Bei kürzeren Schlüsseln (z. B. 1024 Bit) ist dieses Problem mit heutiger Rechenleistung nicht mehr ausreichend schwer – sie gelten als nicht mehr sicher. Ein 2048-Bit-Schlüssel bietet ein Sicherheitsniveau von etwa 112 Bit – das wird aktuell als Mindeststandard für langfristige Sicherheit betrachtet. Organisationen wie NIST, das BSI oder ENISA empfehlen bereits seit Jahren mindestens 2048 Bit für RSA.

Viele moderne Anwendungen setzen sogar auf 3072 oder 4096 Bit, insbesondere wenn eine langfristige Archivierung vorgesehen ist oder hohe regulatorische Anforderungen bestehen.

Wichtig zu erwähnen ist, dass mit RS256 bei gleicher Nachricht und gleichem privaten Schlüssel immer exakt dieselbe Signatur erzeugt wird. Dies ist für die meisten Web-

8 *https://datatracker.ietf.org/doc/html/rfc3447#section-8.2*

anwendungen kein Problem. In manchen Kontexten allerdings – etwa bei sehr hohen Sicherheitsanforderungen oder zur Abwehr bestimmter Angriffe wie Replay- oder Padding-Attacken – kann das problematisch sein.

Hier setzt *RSA-PSS* (bzw. *PS256*, *PS384* und *PS512*) an, ein weiterentwickeltes RSA-basiertes Verfahren, das durch den Einsatz eines zufälligen Wertes jedes Mal eine neue Signatur erzeugt. Es bietet die gleiche Sicherheit, aber deutlich besseren Schutz gegen Attacken, die auf gezielten Kryptoanalysen beruhen. Aus diesem Grund ist RSA-PSS in sicherheitskritischen Anwendungen wie *Open Banking* und in der *Financial-grade API (FAPI)* inzwischen sogar verbindlich vorgeschrieben.[9]

Die Unterstützung von RSA-PSS wird bei gängigen Identity-Providern zunehmend besser. In vielen Fällen muss der Algorithmus jedoch explizit konfiguriert werden, da die Provider standardmäßig oftmals auf RS256 setzen.

Neben RSA-basierten Verfahren erfreut sich auch der *Elliptic Curve Digital Signature Algorithm (ECDSA)* zunehmender Beliebtheit. ECDSA ermöglicht die Verwendung von Elliptischer-Kurven-Kryptografie, die bei kürzeren Schlüssellängen eine vergleichbare Sicherheit wie RSA bietet, aber dabei in vielen Operationen eine höhere Verarbeitungsgeschwindigkeit erreicht.

Effizienz durch elliptische Kurven

Ein 256-Bit-Schlüssel, der auf elliptischen Kurven basiert, bietet dieselbe Sicherheit wie ein 3072-Bit-RSA-Schlüssel. Dadurch können bei gleicher Sicherheitsstufe deutliche kürzere Signaturen erstellt werden. Dabei gilt: Die Länge der Signatur entspricht der Länge des Schlüssels.

ECDSA kommt immer dann zum Einsatz, wenn im JOSE-Header einer der folgenden Werte gesetzt wird: ES256 (in der Praxis meist ausreichend), ES384 oder ES512.

Das Verfahren gilt allgemein als schwerer zu brechen und ist dank der kürzeren Schlüssellängen im Vergleich zu RSA auch performanter, insbesondere bei mobilen oder ressourcenbeschränkten Systemen.

Aus diesem Grund wird der Einsatz von ES256 in Webanwendungen ausdrücklich empfohlen.

Exkurs: EdDSA als Alternative

Der Vollständigkeit halber soll eine Weiterentwicklung der ECDSA-Algorithmen nicht unerwähnt bleiben: der *Edwards-curve Digital Signature Algorithm (EdDSA)*.[10] Während ECDSA weit verbreitet ist, stellt seine sichere Implementierung die Entwickler

9 *https://openid.net/specs/fapi-security-profile-2_0-final.html#name-cryptography-and-secrets*
10 *https://datatracker.ietf.org/doc/html/rfc8032*

vor gewisse Herausforderungen, insbesondere im Umgang mit zufälligen Werten und korrektem Padding.

EdDSA wurde entwickelt, um genau diese Schwächen zu beheben: Er ist robuster, schneller und deutlich einfacher zu implementieren und hat einen klaren Fokus auf moderne kryptografische Anforderungen. EdDSA ist also kryptografisch sehr modern und empfehlenswert – aber bei den Identity-Providern (OIDC/OAuth2) bislang wenig verbreitet.

Wer EdDSA nutzen will, muss meist einen eigenen Secure-Token-Service betreiben oder gezielt auf Systeme setzen, die vollständig selbst konfigurierbar sind (z. B. *Keycloak* oder *Ory*).

Neben den digitalen Signaturen, die auf asymmetrischen Schlüsselpaaren basieren, gibt es noch eine weitere Möglichkeit, die Integrität und Authentizität eines JWT sicherzustellen: die Verwendung eines *Message Authentication Codes (MAC)*.

Insbesondere der weit verbreitete HMAC-Algorithmus bietet eine kompakte und performante Alternative – vor allem in Szenarien, in denen nur eine Partei signiert und validiert oder ein gemeinsames Geheimnis verwendet wird. Werfen wir also einen Blick darauf, wie HMAC funktioniert und wann sein Einsatz sinnvoll ist.

HMAC: Der Standard unter den MAC-Verfahren

HMAC steht für *Hash-based Message Authentication Code* und ist eines der am weitesten verbreiteten symmetrischen Verfahren zur Integritäts- und Authentizitätsprüfung von Daten. Es wird auch in JWTs eingesetzt – etwa bei dem Algorithmus HS256.

Ein HMAC wird erstellt, indem man eine Nachricht (Header und Payload eines JWT) zusammen mit einem geheimen Schlüssel durch einen Hash-Algorithmus wie SHA-256 verarbeitet, und zwar in einem zweistufigen Verfahren mit innerem und äußerem Padding.

Das Ergebnis ist eine feste Prüfsumme, die Manipulationen an der Nachricht zuverlässig erkennt und nur mit dem richtigen Schlüssel reproduzierbar ist.

Mit Online-HMAC-Testern[11] lässt sich demonstrieren, wie das Verfahren funktioniert.

Nehmen wir beispielsweise folgende Nachricht, folgenden Schlüssel und folgenden Hash-Algorithmus:

- Nachricht: »HMAC-Demo für Web Anwendungen«
- geheimer Schlüssel: 118
- Algorithmus: SHA-256

11 *www.freeformatter.com/hmac-generator.html#before-output*

Dann erhalten wir folgendes Ergebnis:

```
feab6598bfac9259d3cd3a5aa937445b9181ce4c3851111f9c90151ce871813d
```

Sicherheit von HMAC

HMAC gilt auch heute noch (bei Verwendung moderner Hashfunktionen wie SHA-256 oder SHA-512) als sehr sicher, schnell und gut analysiert. Wichtig ist dabei:

- Verwenden Sie einen langen, zufälligen geheimen Schlüssel (mindestens 256 Bit werden empfohlen).
- Halten Sie den Schlüssel streng vertraulich und verwenden Sie sichere Speicherlösungen wie Environment-Variablen, einen Secrets-Manager (AWS, GCP, Azure) oder verschlüsselte Konfigurationsdateien.
- Vermeiden Sie die Wiederverwendung von Schlüsseln über Systeme hinweg.

RFC 7518[12] bietet uns drei hash-basierte MAC-Funktionen an, die wir innerhalb des `alg`-Parameters setzen können:

- HS256 (verwendet SHA-256 mit einer Schlüssellänge von 256 Bit)
- HS384 (verwendet SHA-384 mit einer Schlüssellänge von 384 Bit)
- HS512 (verwendet SHA-512 mit einer Schlüssellänge von 512 Bit)

Praxisempfehlung: Welchen HS-Algorithmus sollten Sie nutzen?

In der Praxis ist HS256 meist ausreichend, aber HS512 bietet das höchste Sicherheitsniveau. HS384 und HS512 eignen sich vor allem für Szenarien mit besonders hohen Sicherheitsanforderungen, etwa beim Umgang mit sensiblen Daten im Finanz- oder Gesundheitswesen, bei Tokens mit sehr langer Gültigkeitsdauer oder wenn gezielte Angriffe mit hoher Rechenleistung zu erwarten sind.

Da beide Algorithmen auf stärkeren Hashfunktionen basieren, erfordern sie mehr Rechenleistung und längere Schlüssel, was sie in solchen Fällen besonders robust, aber auch etwas ressourcenintensiver macht.

Auch wenn HMAC bei der Signierung von JWTs praktisch und effizient ist, so bringt das Verfahren doch einige entscheidende Einschränkungen mit sich, besonders in komplexeren oder verteilten Systemlandschaften. Da es sich um ein symmetrisches Verfahren handelt, verwenden sowohl der Aussteller als auch alle Empfänger denselben geheimen Schlüssel. Das bedeutet: Jeder, der ein Token prüfen kann, könnte theoretisch auch selbst eines erzeugen.

12 *www.rfc-editor.org/rfc/rfc7518.html#section-3.2*

In Szenarien, in denen mehrere Dienste, externe Partner oder öffentliche Schnittstellen beteiligt sind, lässt sich dieses Modell kaum sicher skalieren, ohne Kompromisse bei der Vertrauenswürdigkeit einzugehen.

Hinzu kommt, dass der geheime Schlüssel besonders schützenswert ist: Wird er kompromittiert, können sowohl vergangene als auch zukünftige Tokens gefälscht werden – ein Totalschaden für die Token-Sicherheit. Auch eine feingranulare Rollenverteilung (also die Trennung zwischen Instanzen, die signieren, und Instanzen, die ausschließlich verifizieren dürfen) lässt sich mit HMAC nicht umsetzen, da alle Beteiligten denselben geheimen Schlüssel kennen.

Damit ist auch die eingangs erwähnte Anforderung der Nichtabstreitbarkeit (*Non-repudiation*) der Signaturerstellung nicht mehr erfüllt. Das bedeutet: Für einfache, interne Anwendungen ist HMAC weiterhin eine solide Wahl. In komplexeren Architekturen oder bei strengeren Sicherheitsanforderungen stoßen HMAC-basierte JWTs jedoch an ihre Grenzen.

Wie ich bereits erwähnt habe, hängt die Wahl des Signaturalgorithmus zum einen von der Unterstützung durch den verwendeten Identity-Provider ab und zum anderen davon, ob die eingesetzten Bibliotheken den jeweiligen Algorithmus bereits implementieren und korrekt verifizieren können.

Die Spezifikation schreibt lediglich die verpflichtende Unterstützung von HS256 (HMAC mit SHA-256) vor.

Die asymmetrischen Varianten RS256 und ES256 sind zwar nicht zwingend, werden jedoch ausdrücklich empfohlen.[13]

Sicherheitshinweis: Algorithmen einschränken

Bei der Verifikation eines JWT im Backend sollten Sie nur sichere und geprüfte Algorithmen im `alg`-Header zulassen.

Wird ein schwacher Algorithmus oder ein unsicherer Schlüssel verwendet, besteht die Gefahr, dass ein Angreifer das Token manipuliert, den Inhalt verändert und es anschließend neu, aber scheinbar gültig signiert. Die Signaturprüfung würde in diesem Fall ein positives Ergebnis liefern – obwohl das Token gefälscht ist.

Um das zu verhindern, empfiehlt es sich, im Backend eine Whitelist (*Allowlist*) zulässiger Algorithmen zu definieren und alle JWTs mit abweichendem `alg`-Wert strikt abzulehnen. Eine solche Liste sollte auf maximal folgende Algorithmen beschränkt sein: HS256, HS384, HS512, RS256, RS384, RS512, ES256, ES384 und ES512.

Je nach Konfiguration Ihres Identity-Providers kann und sollte diese Liste sogar noch weiter eingeschränkt werden.

13 *www.rfc-editor.org/rfc/rfc7518.html#section-3.1*

Für Produktivumgebungen werden asymmetrische Algorithmen bevorzugt, insbesondere ES256, das als *Recommended+* hervorgehoben wird. Symmetrische HMAC-Algorithmen (HS256, HS384 und HS512) eignen sich primär für automatisierte Tests und Entwicklungsumgebungen, wo sie die Erstellung selbst signierter Test-JWTs vereinfachen. Je nach Konfiguration Ihres Identity-Providers sollte die Allowlist weiter eingeschränkt werden – idealerweise auf einen einzigen Algorithmus.

Angriff: JWT mit alg: "none" – die »Unsigned Token«-Attacke

Neben der Verwendung unsicherer Algorithmen können auch bestimmte kritische Werte im `alg`-Header ein Sicherheitsrisiko darstellen – insbesondere der Wert `none`.

Ein JWT mit `alg: "none"` gilt als sogenanntes unsigniertes JWS, also als ein Token ohne Integritätsschutz, das potenziell beliebig veränderbar ist.

Solche Tokens dürfen standardmäßig nicht akzeptiert werden, da sie keinerlei Schutz gegen Manipulation bieten.

In der Praxis kommt es jedoch immer wieder vor, dass Systeme diese Tokens dennoch verarbeiten – etwa, weil die eingesetzte JWT-Bibliothek den `none`-Wert erlaubt oder weil die Anwendung unsignierte Tokens an einer Stelle zulässt, dies jedoch als global wirksam konfiguriert wurde und dadurch alle Endpunkte betrifft. Das öffnet Tür und Tor für die sogenannte *Unsigned-Token*-Attacke.

Bei dieser Angriffsmethode nutzt ein Angreifer eine unsichere Implementierung von JWT-Validierung aus, die den `alg`-Header-Wert nicht korrekt prüft. Der Angreifer nimmt hierbei einen gültigen JWT und ändert den Header folgendermaßen:

```
{
  "alg": "RS256" -> "none",
  "typ": "JWT"
}
```

Zu guter Letzt entfernt er die komplette Signatur des JWTs und sendet das Token an den Server.

Wenn der Server die Angabe `"alg": "none"` akzeptiert und dadurch keine Signaturprüfung durchführt, wird das manipulierte Token als gültig angesehen. Das bedeutet ebenfalls, dass sämtliche Claims innerhalb der JWT-Payloads frei manipuliert werden können.

Wie gravierend das ist, zeigt sich schnell an einem Beispiel: Ein Angreifer könnte etwa den `role`-Claim problemlos auf `admin` setzen – und würde damit unberechtigt vollen Zugriff erhalten, da keine Integritätsprüfung erfolgt.

Solche Angriffe lassen sich glücklicherweise leicht verhindern

- Akzeptieren Sie niemals `alg: "none"`.
- Verwenden Sie aktuelle JWT-Bibliotheken, die diese Lücke standardmäßig schließen.

Nachdem wir uns mit der Signierung von JWTs und den wichtigsten Sicherheitsaspekten rund um den `alg`-Header beschäftigt haben, stellt sich eine weitere wichtige Frage: Was tun, wenn nicht nur die Integrität, sondern auch der Inhalt des Tokens selbst geschützt werden soll – etwa, um vertrauliche Informationen wie Benutzerdaten oder sensible IDs zu übertragen?

Genau hier kommt *JSON Web Encryption (JWE)* ins Spiel. Im nächsten Abschnitt werfen wir also einen Blick darauf, wie sich JWTs verschlüsseln lassen, welche Mechanismen dahinterstecken und wann ihr Einsatz sinnvoll ist.

4.5 JSON Web Encryption (JWE)

JSON Web Encryption (JWE) ist ein Standard[14] zur Verschlüsselung und Entschlüsselung von Daten, der auf JSON-basierten Datenstrukturen aufbaut.

Im Gegensatz zur *JSON Web Signature (JWS)*, bei der es primär um die Integrität und Authentizität von Daten geht, steht bei JWE der Schutz der Vertraulichkeit im Mittelpunkt, d. h. das Verbergen des Inhalts vor unbefugten Dritten. JWE kommt also immer dann zum Einsatz, wenn sensible Informationen (z. B. persönliche Daten, Zugangscodes oder Tokens) verschlüsselt und sicher übertragen werden sollen.

In der Praxis wird JSON Web Encryption (JWE) im Kontext von OAuth 2.0 und OpenID Connect eher selten eingesetzt. Die meisten Implementierungen setzen auf signierte, aber unverschlüsselte Tokens, da in der Regel die HTTPS-Transportverschlüsselung als ausreichend betrachtet wird und zudem die Unterstützung von JWE in vielen Identity-Providern und den meisten bekannten Frameworks fehlt (wie beispielsweise in *Spring Security*[15]) oder nur eingeschränkt vorhanden ist.

Bei datenschutzkritischen Szenarien werden alternative Ansätze bevorzugt: In OpenID Connect werden sensible Claims nicht im ID-Token übertragen, sondern über den `UserInfo`-Endpunkt abgerufen, der zusätzliche Zugriffskontrolle bietet. Alternativ kommen *Opaque-Tokens* zum Einsatz. Das sind referenzielle Tokens ohne direkt lesbare Informationen, deren Validierung serverseitig über Token-Introspection erfolgt. Diesen Ansatz betrachten wir in Kapitel 6 ausführlicher.

14 *www.rfc-editor.org/rfc/rfc7516.html*
15 *https://github.com/spring-projects/spring-security/issues/4435*

Trotzdem sieht der OpenID-Connect-Standard ausdrücklich vor, dass etwa ein ID Token bei vertraulichen Inhalten auch mittels JWE verschlüsselt werden kann oder sollte.

Das ist Grund genug, sich den Aufbau und die Funktionsweise von JWE einmal genauer anzusehen.

4.5.1 Verschlüsseln und Entschlüsseln von Daten

Ähnlich wie bei der Integritätsprüfung gibt es auch bei der Verschlüsselung von Daten zwei Verfahren: die symmetrische und die asymmetrische Verschlüsselung.

Symmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung verwenden Sender und Empfänger denselben geheimen Schlüssel zum Ver- und Entschlüsseln von Daten (siehe Abbildung 4.2). Der große Vorteil liegt in der Geschwindigkeit und Effizienz: Symmetrische Algorithmen sind in der Regel sehr performant und eignen sich besonders gut für große Datenmengen.

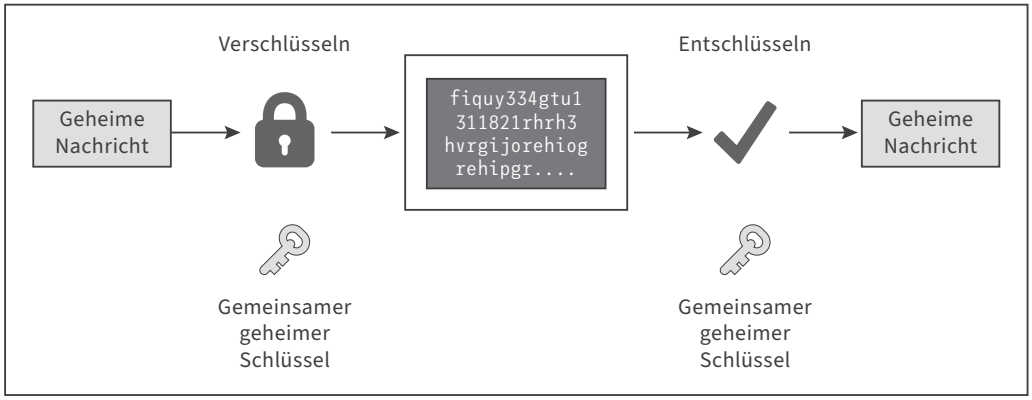

Abbildung 4.2: Symmetrische Verschlüsselung

Ein weit verbreiteter Algorithmus in diesem Bereich ist AES (*Advanced Encryption Standard*).[16] Er gilt als sehr sicher und effizient und wird deshalb häufig zur Verschlüsselung der eigentlichen Nutzdaten (*Payload*) in JWE verwendet. Besonders bei großen Datenmengen oder in Szenarien mit bereits etabliertem Schlüsselmaterial ist die symmetrische Verschlüsselung die bevorzugte Methode.

Asymmetrische Verschlüsselung

Im Gegensatz zur symmetrischen Verschlüsselung basiert die asymmetrische Verschlüsselung auf einem Schlüsselpaar, also einem öffentlichen und einem privaten Schlüssel.

16 *https://web.archive.org/web/20240823165748/https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf*

Der *öffentliche Schlüssel* dient zur Verschlüsselung, während nur der dazugehörige *private Schlüssel* die Daten wieder entschlüsseln kann (siehe Abbildung 4.3). Dieses Verfahren ermöglicht eine sichere Kommunikation, ohne dass vorher ein geheimer Schlüssel ausgetauscht werden muss.

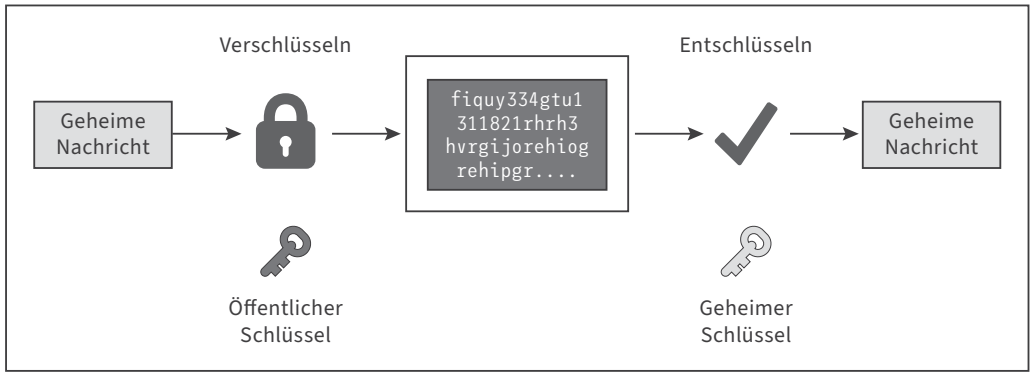

Abbildung 4.3: Asymmetrische Verschlüsselung

In der Praxis kommen hier oft RSA oder Verfahren auf Basis elliptischer Kurven (*Elliptic Curve Cryptography*, *ECC*) zum Einsatz. Innerhalb von JWE wird asymmetrische Verschlüsselung typischerweise zur sicheren Übertragung eines einmalig generierten symmetrischen Schlüssels (z. B. AES-Schlüssels) verwendet – ein Ansatz, den man im Allgemeinen *hybride Verschlüsselung* nennt.

Hybride Verschlüsselung in JWE

Die sogenannte hybride Verschlüsselung verbindet die Effizienz der symmetrischen Verschlüsselung mit der Sicherheit der asymmetrischen Schlüsselaustauschverfahren. Der Ablauf lässt sich in mehreren Schritten erklären:

1. **Ein symmetrischer Sitzungsschlüssel wird generiert.**

   Der Sender (z. B. ein Webserver) erzeugt einen zufälligen symmetrischen Schlüssel (meist einen AES-Schlüssel), der nur für diese eine Nachricht oder Sitzung gilt.

2. **Die Payload wird symmetrisch verschlüsselt.**

   Die eigentlichen Daten (*Payload*) werden mit diesem AES-Schlüssel verschlüsselt.

3. **Der symmetrische Schlüssel wird asymmetrisch verschlüsselt.**

   Anschließend wird der zuvor generierte AES-Schlüssel mit dem öffentlichen Schlüssel des Empfängers asymmetrisch verschlüsselt (z. B. mit RSA oder einem elliptischen Kurvenverfahren wie ECDH-ES).

4. **Der Empfänger entschlüsselt den AES-Schlüssel und danach die Daten.**

   Nur der Empfänger, der im Besitz des zugehörigen privaten Schlüssels ist, kann den symmetrischen AES-Schlüssel wiederherstellen. Mit diesem kann er dann die eigentliche Payload entschlüsseln.

Dieser hybride Ansatz ist in vielen modernen kryptografischen Protokollen verbreitet – nicht nur in JWE, sondern auch in TLS.

Zusätzliche Sicherheit durch Forward Secrecy

In besonders sicherheitskritischen Kontexten empfiehlt es sich, bei jedem Austausch einen neuen symmetrischen Schlüssel zu verwenden – ein Prinzip, das auch als *Forward Secrecy* oder *Perfect Forward Secrecy (PFS)* bekannt ist.

Moderne Sicherheitsprotokolle wie TLS 1.3 setzen auf Forward Secrecy durch kurzlebige Schlüsselaustauschverfahren (z. B. ECDHE). Auch JWE unterstützt mit Algorithmen wie ECDH-ES oder RSA-OAEP den Ansatz, pro JWT einen neuen Schlüssel zu erzeugen.

4.5.2 JWT-Verschlüsselung mit JWE

Ein verschlüsseltes JSON Web Token (JWT) ist leicht von einem signierten JWT zu unterscheiden, da es fünf statt der üblichen drei Abschnitte hat. Diese sind:

1. der geschützte Header (`protected header`)
2. der verschlüsselte Schlüssel (`encrypted key`)
3. der Initialisierungsvektor (`initialization vector`)
4. der Chiffretext (`ciphertext`) selbst
5. ein Authentifizierungs-Tag (`authentication tag`)

Der geschützte Header entspricht weitgehend einem herkömmlichen JWT-Header, beschreibt in diesem Fall jedoch das konkrete Verschlüsselungsverfahren und enthält ergänzende Parameter zu den eingesetzten kryptografischen Algorithmen, die in Tabelle 4.3 erklärt werden.

Parameter-Name	Beschreibung	Beispielwert
`alg` (*Algorithm*)	Bezeichnet den asymmetrischen Verschlüsselungsalgorithmus, der verwendet wurde, um den symmetrischen Schlüssel zu verschlüsseln.	`"RSA-OAEP"`
`enc` (*Encryption Algorithm*)	Bezeichnet den symmetrischen Verschlüsselungsalgorithmus, mit dem der eigentliche Chiffretext erzeugt wurde.	`"A256GCM"`
`kid` (Key ID)	Beschreibt, welcher öffentliche Schlüssel verwendet wurde, um den symmetrischen Schlüssel zu verschlüsseln.	`"key-214232"`

Tabelle 4.3: Ergänzende Parameter zu den eingesetzten kryptografischen Algorithmen

Parameter-Name	Beschreibung	Beispielwert
`zip` (*Compression Algorithm*)	Beschreibt den Komprimierungsalgorithmus, der auf den Klartext vor der Verschlüsselung angewendet wurde, sofern vorhanden.	`"DEF"`
`typ` (*Type*)	Der Parameter `typ` gibt an, welchen Medientyp das gesamte JWS-Objekt hat (z. B. `JWT`); allerdings kann ein verschlüsselter JWT (JWE) auch einen `cty`-Header (*Content Type*) enthalten. Dieser gibt an, welche Art von Daten im verschlüsselten Inhalt enthalten sind – also die Art der Klartext-Daten hinter dem Chiffretext.	`"JWT"`
`cty` (*Content Type*)	Typ der Payload (z. B. JWT als verschachtelter Inhalt)	`"JWT"`

Tabelle 4.3: Ergänzende Parameter zu den eingesetzten kryptografischen Algorithmen (Forts.)

Als Nächstes folgt der *Content Encryption Key (CEK)*. Bei ihm handelt es sich um einen symmetrischen Schlüssel (in der Regel einen AES-Schlüssel), der einmalig und ausschließlich für diese Payload erzeugt wurde. Im Rahmen des hybriden Verschlüsselungsverfahrens wird dieser Schlüssel mithilfe des asymmetrischen öffentlichen Schlüssels des Empfängers verschlüsselt und im JWE abgelegt. Eine Wiederverwendung dieses symmetrischen Schlüssels findet nicht statt.

Weiterer Bestandteil des verschlüsselten JWTs ist der *Initialisierungsvektor (IV)*, ein essenzieller Bestandteil vieler symmetrischer Verschlüsselungsverfahren, insbesondere bei Blockchiffren wie AES im CBC- oder GCM-Modus.

Der IV ist ein zufällig gewählter Wert, der zusammen mit dem Klartext in den Verschlüsselungsprozess eingeht. Seine Hauptfunktion besteht darin, sicherzustellen, dass selbst bei identischem Klartext und gleichem Schlüssel jedes Mal ein anderer Chiffretext erzeugt wird. Dadurch wird verhindert, dass sich Muster im Klartext auch im Chiffretext widerspiegeln, was potenziell Rückschlüsse auf die zugrunde liegenden Daten ermöglichen würde.

Die Einbeziehung eines Initialisierungsvektors trägt somit zur semantischen Sicherheit bei – also zu der Eigenschaft, dass ein Angreifer keine Informationen über den Klartext allein anhand des Chiffretextes gewinnen kann. Wird ein symmetrisches Verfahren verwendet, das keinen IV benötigt (etwa bestimmte Stromchiffren), so bleibt dieses Feld im JWE-Datensatz leer.

Auf den Initialisierungsvektor folgt der Chiffretext selbst, also die verschlüsselte, vertrauliche Payload des JWTs. Diese wurde mit dem zuvor genannten *Content Encryption Key* und dem *Initialisierungsvektor* verschlüsselt.