SICHERHEITSRISIKEN FÜR APIS

No API risks, more fun

Womit kaum jemand gerechnet hat: Die OWASP API Top Ten hat ein Update erfahren. Was bedeutet das für ASP.NET Core?

Die Vereinigung OWASP – das Akronym steht für Open Web Application Security Project [1] – ist eine bekannte und viel zitierte Quelle für alles rund um das Thema Websicherheit. Von lokalen und Online-Treffen über Cheatsheets und Checklisten bis hin zu diverser Open-Source-Software gibt es hier zu fast allen Facetten von Web Application Security ein von Freiwilligen erstelltes Angebot. Doch denkt man an die OWASP, denkt man zunächst an eine ganz bestimmte Veröffentlichung – die bekannteste und auch eine der ersten: die OWASP Top Ten [2], eine Liste der zehn größten Sicherheitslücken für Webanwendungen (Bild 1). Die Liste wird mit einigem Aufwand alle drei bis vier Jahre erstellt und aktualisiert, letztmals 2021 (und die internen Wetten des Autors dieses Artikels laufen eher in die Richtung, dass es erst 2025 wieder eine neue Fassung geben wird). Was das genau aus Sicht von ASP.NET Core bedeutet, haben wir vor etwa zwei Jahren in der dotnetpro ausführlich behandelt [3].

Die Liste erhebt dabei – trotz eines sehr datengetriebenen Ansatzes bei der Erstellung – keinen Anspruch auf wissenschaftliche Genauigkeit, sondern dient vielmehr der Bewusstseinsschaffung: Nur mit entsprechender Awareness und Kenntnis von Risiken ist es effizient möglich, sich dagegen zu verteidigen. Und deswegen ist etwa Punkt 1 in der OWASP Top Ten nicht notwendigerweise „wichtiger" als Punkt 2.

Natürlich findet eine starke Verdichtung und Vereinfachung statt, wenn der komplette Themenkomplex „Risiken für Webapplikationen" in zehn Punkten zusammengefasst wird. Außerdem hängt es auch ein bisschen von der Art der Anwendung ab, welche Risiken wahrscheinlicher auftreten und welche eher nicht. Aus diesem Grund hat die OWASP schon vor einiger Zeit damit begonnen, weitere Top-Ten-Listen für bestimmte „Nischen" zu erstellen (Tabelle 1 führt einige weitere Vertreter auf).

Nach der „Haupt-Top-Ten" vermutlich auf Platz 2 in Hinblick auf den Bekanntheitsgrad rangiert die „OWASP API Security Top Ten" [4], bei der es also spezifisch um APIs geht. Im Jahr 2019 wurde die Liste erstmals veröffentlicht [5], mit zunächst überschaubarem Echo in der Praxis. Etwas überra-

Tabelle 1: Weitere OWASP-Top-Ten-Listen

Innerhalb der OWASP gibt es abseits der „großen" Top Ten auch noch weitere Projekte, bei denen es um die mehr oder minder faktengestützte Erstellung von Listen zu bestimmten Sicherheitsthemen geht. Nachfolgend eine kleine Auswahl von Projekten, bei denen es zumindest schon eine als „stabil" gekennzeichnete Veröffentlichung gegeben hat. Darüber hinaus finden sich auf der Projektübersichtsseite der OWASP [13] noch einige weitere Top-Ten-Projekte, die noch nicht so weit sind.

Top Ten	Letzte Aktualisierung	URL
OWASP Data Security Top 10	2023	https://owasp.org/www-project-data-security-top-10/
OWASP Desktop Application Security Top 10	2021	https://owasp.org/www-project-desktop-app-security-top-10/
OWASP Kubernetes Top Ten	2022	https://owasp.org/www-project-kubernetes-top-ten/
OWASP Low-Code/No-Code Top 10	2023	https://owasp.org/www-project-top-10-low-code-no-code-security-risks/
OWASP Machine Learning Top Ten	2023	https://owasp.org/www-project-machine-learning-security-top-10/
OWASP Mobile Top 10	2023	https://owasp.org/www-project-mobile-top-10/
OWASP Smart Contract Top 10	2023	https://owasp.org/www-project-smart-contract-top-10/
OWASP Top 10 CI/CD Security Risks	2022	https://owasp.org/www-project-top-10-ci-cd-security-risks/
OWASP Top 10 Privacy Risks	2021	https://owasp.org/www-project-top-10-privacy-risks/

schend gab es Mitte 2023 eine Neuauflage (Bild 2) [6]. Und 2023 setzen noch mehr Anwendungen auf APIs als noch 2019, allen voran Single-Page Applications. Grund genug also, einen genaueren Blick auf die neue Liste zu werfen, natürlich auch durch die ASP.NET-Core-Web-API-Brille.

Die vielleicht wichtigste Frage vorweg: Braucht es diese Liste tatsächlich? Sind APIs und allgemeine Webanwendungen tatsächlich so unterschiedlich? Hier gibt es aus der Sicht des Autors zwei zentrale Aspekte. Zum einen geht es um Awareness, und hier sind besonders spezifische Empfehlungen und Informationen stets willkommen. Und natürlich gibt es Unterschiede aus Sicherheitssicht. Hier ein plakatives, wenngleich etwas konstruiertes Beispiel. Punkt 3 der OWASP Top Ten ist Injection, das im Wesentlichen SQL Injection und (vor allem!) Cross-Site Scripting beinhaltet (eine ausführlichere Diskussion wurde in [3] geführt). Jetzt scheint gerade XSS auch etwas, das bei APIs passieren könnte. Es wäre doch denkbar, dass ein API-Endpunkt ein HTML-Fragment zurückliefert, das dann JavaScript-Code enthält. Wenn dann dieser API-Endpunkt direkt im Browser aufgerufen wird, würde der Code im Sicherheitskontext der aktuellen Seite (genauer gesagt: des Origins der aktuellen Seite, also Protokoll + Domain + Port) ausgeführt werden.

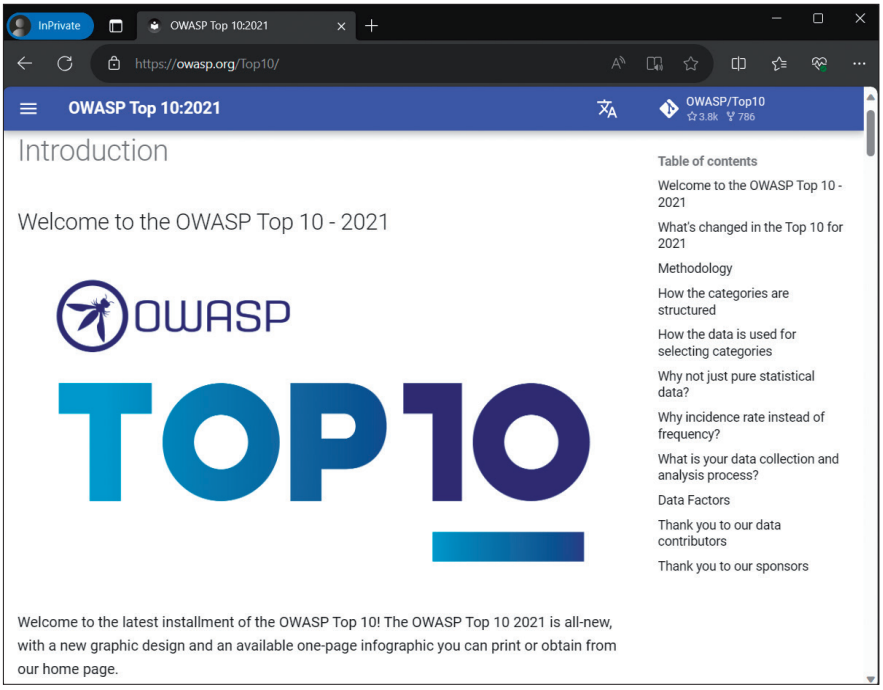
Der Konjunktiv ist berechtigt. Eingebetteter Code in diesem Szenario käme nur zur Ausführung, wenn der Browser die API-Rückgabe als HTML rendern würde. Beim herkömmlichen Ansatz, via ASP.NET Core Web API Daten zurückzugeben, liefe es aber anders. ASP.NET Core Web API liefert standardmäßig JSON zurück und setzt in diesem Fall automatisch den HTTP-Response-Header *Content-Type* auf *application/json*, weist also den Browser darauf hin, dass JSON-Daten zurückkommen und eben kein HTML. Der Browser handhabt dann die Daten korrekterweise als JSON und zeigt sie entweder als Text an oder, wie beispielsweise im Firefox, visuell als Struktur (Bild 3).

Um mit Gewalt eine XSS-Lücke in ein API zu integrieren, wäre der folgende praxisferne Ansatz erforderlich:

```
public ContentResult HelloAPI(string name)
{
  var result = "Hello " + name;

  return new ContentResult
  {
    ContentType = "text/html",
    Content = result
  };
}
```

Die entscheidende Stelle ist dabei das Setzen des *Content-Type*-Headers. Bleibt dies aus, bleibt es bei *application/json* und der Angriff funktioniert nicht mehr wie beschrieben. Natürlich sind andere Vektoren denkbar, etwa eine clientseitige Implementierung, die JavaScript-Code im JSON zur Ausführung bringt, aber das Beispiel zeigt: XSS ist bei APIs ein seltenes Risiko als bei „herkömmlichen" Webanwendungen.



Risiken hoch zehn: die OWASP Top Ten (Bild 1)

Auch bei Cross-Site Request Forgery (CSRF) zeigt die Praxis, dass das bei APIs seltener ein Thema ist, denn die sind vielmals per Token abgesichert, das per HTTP-Header und nicht via Cookie übermittelt wird. Ein klassischer CSRF-Vektor ist damit ausgehebelt.

Auf dem Weg zur Liste

Ebenso wie die große Schwester, die OWASP Top Ten, gibt es auch für die Erstellung der API Security Top Ten einen datengetriebenen Prozess, wenngleich nicht ganz so strikt wie beim Vorbild. Eingabedaten für die Ermittlung der relevanten Risiken sind zum einen öffentliche Daten aus Security Advisories und aus Bug Bountys. Zudem wurden Firmen und Personen, die Security-Audits durchführen, um ihre – freilich anonymisierten – Ergebnisse gebeten. Was für die reguläre OWASP Top Ten eine zentrale Datenquelle ist, hat bei der API Top Ten leider nicht geklappt – zu gering war der Rücklauf. So wurden Teile der Top Ten dann eher zusammen mit Experten diskutiert als errechnet.

Bei der Sortierung und Zusammenfassung der Sicherheitsrisiken setzt die OWASP API Security Top Ten wie die OWASP Top Ten auch auf die CVE-Systematik (Common Vulnerabilities and Exposures). Jede Schwachstelle hat dort eine Nummer. Diese kann auch niedrig vierstellig sein. Um also in ▶

einer Top-Ten-Liste nicht zu viel wegwerfen zu müssen, werden mehrere CVEs zu einem Oberbegriff zusammengefasst. Diese Überschriften wiederum, die dann die einzelnen Punkte der Top Ten ausmachen, sind willkürlich. Das soll nicht abwertend klingen, verdeutlicht aber, dass es sich in der Tat um eine Awareness-Liste handelt.

Ergebnis der Bemühungen für 2023 war dann die folgende Liste der zehn größten Sicherheitsrisiken für APIs:

- Broken Object Level Authorization
- Broken Authentication
- Broken Object Property Level Authorization
- Unrestricted Resource Consumption

APIs in Gefahr: die OWASP API Security Top Ten (Bild 2)

- Broken Function Level Authorization
- Unrestricted Access to Sensitive Business Flows
- Server-Side Request Forgery
- Security Misconfiguration
- Improper Inventory Management
- Unsafe Consumption of APIs

Ein kurzer Blick auf diese Liste zeigt schon, dass manche der Punkte durchaus ähnlich sind und sich nur in Details unterscheiden. Was bei einer so spezifischen Liste ein Vorteil ist, nährt aber auch den Verdacht, dass es vielleicht weniger als zehn Listeneinträge hätten sein können. Gleichermaßen muss allerdings konstatiert werden, dass (fast) jeder dieser Punkte seine Daseinsberechtigung hat und eine nuancierte Diskussion der einzelnen Risiken gerechtfertigt ist. Werfen wir also einen Blick auf jeden einzelnen Eintrag. Wir können dabei nicht jeden Aspekt jedes Risikos auf der Liste betrachten, picken uns aber immer plakative Vertreter heraus.

1. Broken Object Level Authorization

Der erste Eintrag in der OWASP Top Ten ist „Broken Access Control“, ein Sammelbegriff, unter dem sich 34 (!) CWEs versammelt haben. Für eine „kaputte Zugriffskontrolle“ kann es mehrere Ursachen und auch Angriffswege geben.

Die API Security Top Ten nutzt die Gelegenheit, das Ganze in unterschiedliche Facetten aufzuteilen. Die Spitzenposition wird dabei von einer mangelhaften Autorisierung auf Objektebene eingenommen. Das typischste Muster hierfür ist der klassische API-URI für den Zugriff auf ein bestimmtes Objekt, etwa *http://api.example.com/people/123* – hierbei steht *123* für die ID des jeweiligen Objekts, egal ob numerisch oder GUID oder ein anderer Datentyp. Es ist also geradezu trivial, beliebige IDs durchzuprobieren; die „Trefferquote“ für einen gültigen Identifikator ist beim immer noch häufig eingesetzten Integer besonders hoch.

Fehlt die Autorisierung, landet dieses Risiko direkt in dem Topf „Broken Object Property Level Authorization“. Ein klassisches Beispiel aus der jüngeren Vergangenheit: Ein populäres API prüft zwar, ob der Client authentifiziert ist; anonymer Zugriff auf ein Objekt ist nicht gestattet. Allerdings ging das Entwicklungsteam offenbar naiv davon aus, dass niemand IDs einfach raten würde, sondern ein Zugriff nur auf eigene Objekte (deren IDs waren API-Clients bekannt) erfolgt. Dies hält natürlich nicht gezieltem Raten stand. Es gilt nicht nur zu prüfen, ob ein Client bekannt (sprich: authentifiziert) ist, sondern auch, ob ein Client auch das Recht hat (vulgo: autorisiert ist), auf eine bestimmte Ressource zuzugreifen.

Selbst bei GUIDs als Datentyp für die ID, wo ein Erraten fast unmöglich ist, erscheint eine Autorisierung auf Objektebene zwingend. Es kann ja durchaus sein, dass eine gültige, fremde GUID anderweitig in Erfahrung gebracht werden kann, etwa durch Überprüfung des (unverschlüsselten) Netzwerkverkehrs. Nicht nur für dieses Risiko gilt: Code muss defensiv sein und immer damit rechnen, dass es sich bei einem Request um einen Angriff handelt.

2. Broken Authentication

Die eben angesprochene Authentifizierung muss natürlich auch ordentlich erfolgen, ansonsten ist Punkt 2 in der OWASP API Security Top Ten „zuständig“. Gerade bei APIs gibt es hier ein geradezu klassisches Angriffsszenario, und das hat mit JWTs zu tun. JSON Web Token werden gerne für die Authentifizierung verwendet.

Das Format sieht prinzipiell drei Bausteine vor: einen Header-Abschnitt mit Metadaten über das Token, die Payload

– also die eigentlichen Daten – und eine Signatur für das komplette Token. Die von Auth0 betriebene Website *jwt.io* visualisiert das sehr anschaulich (Bild 4).

Das klingt doch eigentlich ganz gut: Der Datenbereich des JWT ist einsehbar, weil der Inhalt nur Base64-codiert wird. Allerdings lässt sich das scheinbar nicht manipulieren, denn die Signatur wird mit einem nur dem Server bekannten Schlüssel über den Datenbereich erzeugt. Der Signatur-Algorithmus ist im Header im Schlüssel *"alg"* hinterlegt, in Bild 4 ist es HMAC mit SHA-256 (HS256). Das Problem: Die Spezifikation sieht auch den Algorithmus *"none"* vor, also gar keine Signatur. Ein solches Token lässt sich wunderbar fälschen. Akzeptiert also das API Token ohne Signatur, kann man in der Tat von „Broken Authentication" sprechen. Die gute Nachricht: Solange wir in ASP.NET Core den Signaturschlüssel hinterlegen, führen JWTs ohne Signatur zu HTTP 401:

```
builder.Services.AddAuthentication(options => {
    /* ... */ })
  .AddJwtBearer(o =>
  {
    o.TokenValidationParameters =
        new TokenValidationParameters
    {
      IssuerSigningKey = new SymmetricSecurityKey(
        Encoding.ASCII.GetBytes("dotnetpro!"))
    };
  });
```

Standardmäßig bietet ASP.NET gegen diesen Vektor einen guten Schutz. Der einzige offensichtliche Workaround (aus Implementierungssicht) bestünde darin, in den *TokenValidationParameters* die Eigenschaft *SignatureValidator* zu setzen. Dieser Ansatz riecht allerdings sehr offenkundig nach Anti-Pattern.

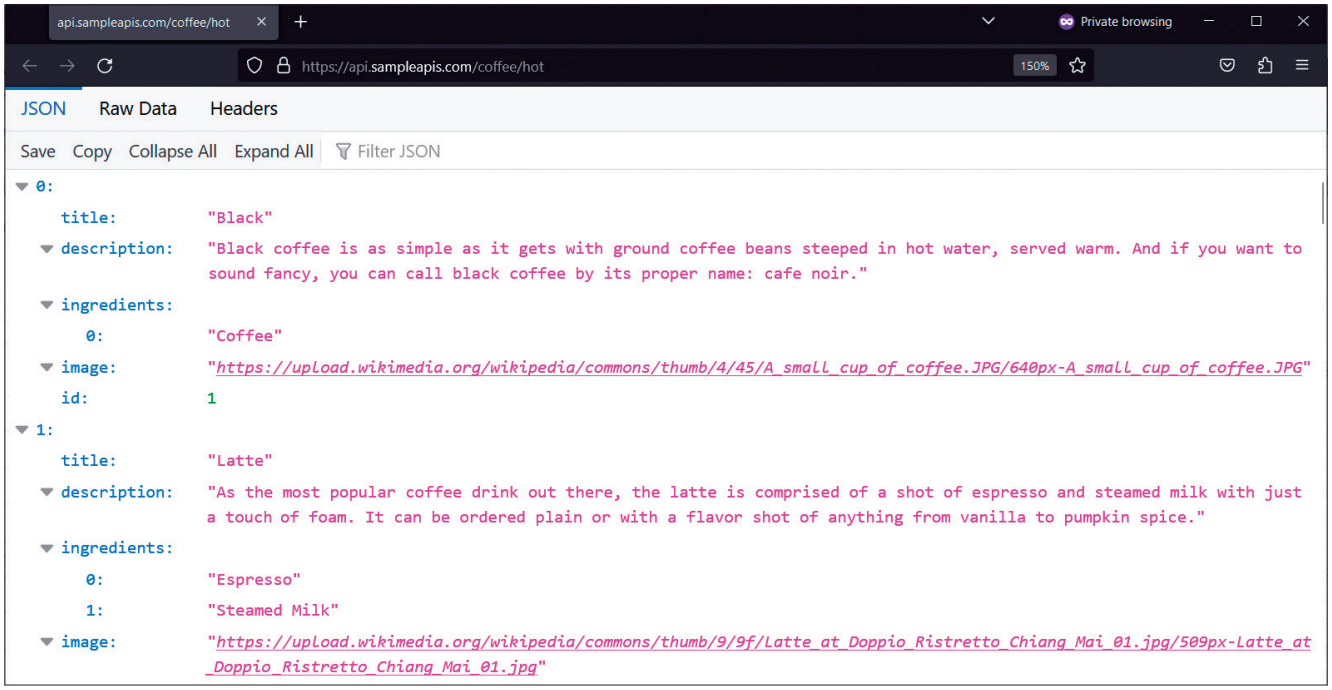
In der Praxis etwas häufiger zu finden sind weitere fehlende Validierungen, etwa des Issuers oder der Audience. Gerade Letzteres ist in zahlreichen Setups ein Unterscheidungsmerkmal zwischen Test- und Produktivsystem. Insofern sollte zwingend eine Überprüfung stattfinden, ob das Token von der erwarteten Stelle und für das Zielsystem ausgestellt worden ist. Und natürlich darf das Ablaufdatum des Tokens (sofern gesetzt – sehr empfehlenswert) noch nicht erreicht sein. ASP.NET Core bietet für diese Überprüfungen entsprechende Optionen an, die alle standardmäßig auf *true* gesetzt sind:

- *ValidateIssuer*
- *ValidateAudience*
- *ValidateLifetime*

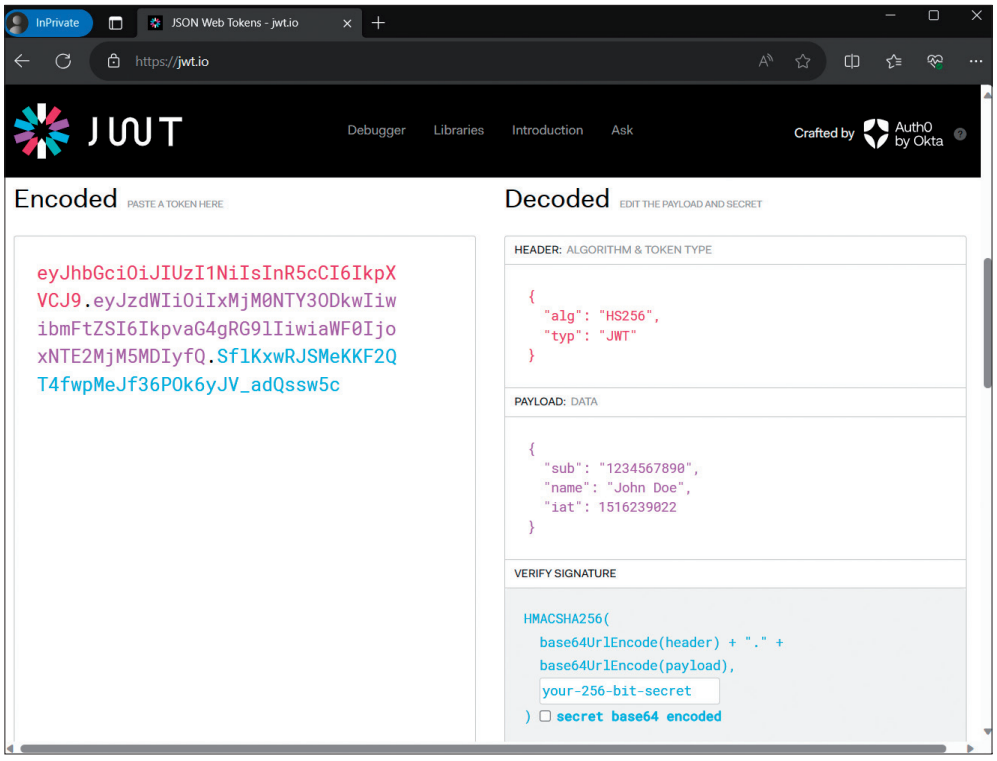
Auch hier gilt wieder: Standardmäßig ist ein Schutz vorhanden, der aber auch ausgehebelt werden kann. Wenn Sie obige Eigenschaften auf *false* gesetzt in einer Codebasis finden, suchen Sie nach dem Grund dafür!

3. Broken Object Property Level Authorization

Der dritte Punkt auf der Liste klingt fast wie der erste, nur dass es jetzt um Objekteigenschaften, nicht um Objekte geht. War es vorher relevant, ob der Client Zugriff auf ein Objekt hat, so geht es dieses Mal darum, ob die entsprechende Objekteigenschaft ausgelesen oder geändert werden darf. Ein geradezu klassischer Angriff in diesem Bereich ist Mass Assignment, gerne auch Overposting genannt. Dazu ein illustratives Beispiel: In einem Bugtracker gehen wir von einer Modellklasse aus, die ein Issue, also einen Eintrag im System repräsentiert. Hier der relevante Teil der Implementierung dieser Klasse: ▶



JSON, nicht JavaScript: Browser verarbeiten JSON-Rückgaben als Daten, nicht als Code (Bild 3)



Aller guten Dinge sind drei: Bestandteile eines JSON Web Token (Bild 4)

```
public class Issue
{
  public Issue()
  {
    this.CreationDate = DateTime.UtcNow;
  }

  public int Id { get; set; }
  public string Title { get; set; } = string.Empty;
  public string Description { get; set; } =
    string.Empty;
  public DateTime CreationDate { get; set; }
}
```

Ein Issue hat also unter anderem ein Erstelldatum, und das wird beim Instanzieren der Klasse in deren Konstruktor automatisch gesetzt. Die zugehörige API-Methode zum Anlegen eines Issue sieht wie folgt aus:

```
[ApiController]
public class IssueController : ControllerBase
{
  [HttpPost]
  public async Task<ActionResult<Issue>> PostIssue(
      Issue issue)
  {
    // Issue in Datenbank persistieren
  }
}
```

Die Methode *PostIssue()* enthält einen Wert vom Typ *Issue* als Parameter. Hier ist das Model Binding von ASP.NET Core im Einsatz, das ein *Issue* erstellt und dabei als Eingabedaten unter anderem den Body des HTTP-Requests verwendet. Die Erwartungshaltung seitens des Implementierungsteams des API war es wohl, dass hier lediglich Title und Description mitgeliefert werden würden. Das Erstelldatum wiederum wird in der Modellklasse selbst gesetzt, und zwar sofort bei der Instanzierung im Konstruktor. Enthält allerdings der API-Request ebenfalls einen Wert namens *CreationDate*, so wird die gleichnamige Eigenschaft der *Issue*-Instanz auf eben diesen Wert gesetzt. Der Client hat also eine Eigenschaft gesetzt, die eigentlich nicht gesetzt werden dürfte – das bequeme Model Binding hat es möglich gemacht. Auf genau diese Art und Weise gelang es einst einem Nutzer, auf GitHub ein Issue anzulegen, das laut Erstelldatum erst 1001 Jahre in der Zukunft erzeugt wird (schöne Zusammenfassung bei ZDNET [7]). Ganz klar: Hier liegt eine fehlende Autorisierung auf Eigenschaftenebene vor. Kleiner Trost: Bei Verwendung des Scaffolding-Features in Visual Studio zum Erzeugen eines API-Controllers mit Action-Methoden inklusive Einsatz von Entity Framework Core wird automatisch ein Kommentar angelegt, der vor diesem Angriff warnt.

Gegenmaßnahmen liegen auf der Hand: Entweder eine Erlaubtliste – explizit angeben, an welche Eigenschaften gebunden werden darf – oder ausnahmsweise eine Verbotsliste – welche Properties sind tabu? Noch einfacher ist es freilich, mit spezifischen Modellen für genau die zu erwartenden Daten beim API-Aufruf zu arbeiten, ähnlich wie ViewModels.

4. Unrestricted Resource Consumption

Alle Ressourcen sind endlich. Diese offensichtliche Wahrheit gilt natürlich für jede Art von Webanwendung, aber bei APIs wird sie seitens der OWASP speziell behandelt und landet so

auf Platz 4 der API Security Top Ten. Gern implementiert wird bei APIs eine Form von Rate-Limiting: Wie oft darf das API innerhalb welches Zeitintervalls aufgerufen werden? In ASP.NET Core gibt es seit Version 7 eine direkt integrierte Middleware für Rate Limiting, die in der dotnetpro vor Kurzem ausführlich vorgestellt wurde [8]. Hier die Kurzfassung: Die Middleware kann in der *Program.cs* hinzugefügt werden. Über Policies legen wir dazu die Rahmenbedingungen für eine Laststeuerung an, etwa die Anzahl der Anfragen innerhalb eines fixen Zeitfensters:

```
builder.Services.AddRateLimiter(options => {
  options.AddConcurrencyLimiter(
    policyName: "OneTwoThree",
    options =>
    {
      options.PermitLimit = 3;
    });
```

Der API-Endpunkt bekommt dann die folgende Policy zugewiesen:

```
app.MapGet("/nur-drei", () => Results.Ok("1, 2 oder 3"))
  .RequiresRateLimiting("OneTwoThree");
```

Nicht vergessen werden darf, die Middleware abschließend zu aktivieren:

```
app.UseRateLimiter();
```

Wie bereits in [7] diskutiert, ist diese Middleware allein kein Allheilsbringer. Eine wichtige Ressource ist die verfügbare Netzwerkbandbreite; durch entsprechend orchestrierte DDoS-Angriffe kann eine Anwendung hier empfindlich getroffen werden, und zwar bevor das Rate-Limiting eingreift. Bei einer in der Cloud gehosteten Lösung ist es zu überlegen, ob gegebenenfalls dort eine der Lösungen zur Laststeuerung bei APIs zum Einsatz kommt, um sowohl API-Zugriffe zu beschränken als auch potenziell die Anwendung dynamisch skalieren zu können. Bei Azure und bei AWS etwa gibt es entsprechende Funktionalität jeweils in einer Suite von Tools, die bei beiden Cloud-Plattformen API Management genannt wird [9] [10].

Aber auch ganz ohne DDoS ist das Rate-Limiting nicht mal in allen Szenarien ausreichend, in denen es lediglich um die Zugriffe geht. Ein schönes Beispiel führt die OWASP API Security Top Ten selbst aus. Es geht darin um GraphQL. Die ursprünglich von Facebook für den Eigengebrauch entwickelte Abfragesprache erfreut sich wachsender Popularität, allerdings nicht unbedingt ebenso großer Kenntnis der Möglichkeiten und Grenzen. Es gibt wohl einen Grund, wieso das Cheatsheet der OWASP zu GraphQL [11] nur etwa 20 Prozent weniger Zeichen aufweist als dasjenige zum „Evergreen" SQL Injection [12].

Angenommen, ein API-Endpunkt erzeugt via GraphQL Last auf der Datenbank; deswegen soll der Endpunkt per Rate-Limiting zugriffsbeschränkt werden, nur noch eine begrenzte Anzahl von Aufrufen pro Zeiteinheit. GraphQL ist allerdings so flexibel, dass in einer Anfrage mehrere Queries mitgeschickt werden können. Statt beispielsweise *{ "query": "..." }* ist auch folgende Payload möglich:

```
[
  { "query": "..." },
  { "query": "..." },
  ...
]
```

Das Rate-Limiting muss also innerhalb von GraphQL oder gegebenfalls innerhalb der Datenbank implementiert werden, da in einem Request selbst beliebig viele GraphQL-Queries möglich sind.

So weit unser erster Blick in die aktuelle OWASP API Security Top Ten. Vier von zehn Punkten haben wir bereits behandelt. Die restlichen kommen in der nächsten Ausgabe zur Sprache. ■

[1] Das Open Web Application Security Project, https://owasp.org
[2] Die OWASP Top Ten, https://owasp.org/Top10/
[3] Christian Wenz, No risk, more fun, dotnetpro 12/2021, Seite 54 ff., www.dotnetpro.de/A2112OWASP
[4] Die OWASP API Security Top Ten, https://owasp.org/API-Security/
[5] Die OWASP API Security Top Ten 2019, www.dotnetpro.de/SL2401OWASP1
[6] Die OWASP API Security Top Ten 2023, www.dotnetpro.de/SL2401OWASP2
[7] Mass Assignment bei GitHub, www.dotnetpro.de/SL2401OWASP3
[8] Christian Wenz, Blockabfertigung für den Massenansturm, dotnetpro 10/2023, Seite 34 ff., www.dotnetpro.de/A2310RateLimiting
[9] Azure API Management, www.dotnetpro.de/SL2401OWASP4
[10] AWS API Management, www.dotnetpro.de/SL2401OWASP5
[11] OWASP Cheat Sheet zu GraphQL, www.dotnetpro.de/SL2401OWASP6
[12] OWASP Cheat Sheet zu SQL Injection, www.dotnetpro.de/SL2401OWASP7
[13] Auflistung von kleineren Projekten innerhalb der OWASP, www.dotnetpro.de/SL2401OWASP8

Christian Wenz
ist Mitgründer der Agentur Arrabiata Solutions GmbH (www.arrabiata.de) und verantwortet dort die Themen Performance, mobile Anwendungen und Security. Er ist MVP für ASP.NET und ASPInsider. Sein neuestes Buch „ASP.NET Core Security" ist bei Manning erschienen.

dnpCode A2401OWASP