Leveraging the OWASP API Security top 10 to build secure web services

Enrique Cabezas
Leveraging the OWASP API Security top 10 to build secure web services

GIAC (GWAPT) Gold Certification

Author: Enrique Cabezas, enrique.cabezas@pwc.com
Advisor: Mohammed Haron
Accepted: November 3rd 2020

Abstract

Imagine you decide to build an application using web services. What are the main aspects to consider when it comes to security? With the first version of the OWASP API Security top 10 being released, exploring the defensive aspect of each entry in the top 10 will allow us to revisit them and reflect on what could be some good practices to follow. While reviewing a web service on all best-practice security measures might not be in line with an organization’s risk appetite, this approach offers the reader the benefit of mitigating the most critical types of vulnerabilities as a starting point. We will showcase the architecture of a straightforward banking application using SOAP, REST and GraphQL respectively. This will allow us to demonstrate diverse attention points specific to these technologies when it comes to finding solutions for each unique OWASP API security API top 10 vulnerability class.
1. Introduction

As web applications and their underlying technologies evolved over the years, so did web services with the usage of Application Programming Interfaces (API) itself leading to the popularity of the term “web API”. Examples of most commonly used styles to build web APIs include Simple Object Access Protocol (SOAP), Representational state transfer (REST) and, with its rising popularity, GraphQL.

Originally specified in 1998, SOAP allows clients and servers to communicate neutrally over multiple protocols (predominantly HTTP, but also SMTP, TCP and UDP), independent of programming languages and design patterns, making great use of XML’s extensibility features. The Web Services Description Language (WSDL) proved to be virtually indispensable when describing the supported SOAP operations of an endpoint in detail.

REST, unlike SOAP, is an architecture style which was defined in 2000. Web services can adopt a similar structure as the one in REST’s original definition but those fully abiding by REST’s set of constraints are called RESTful web services. Nowadays, REST web services are the most popular type of web APIs because of its simplicity which contrasts with SOAP.

Finally, GraphQL was created in 2012 by Facebook, publicly released in 2015, and it can be compared to REST as both are designed to fetch data from resources. However, GraphQL is a query language with its own specification. As such, it is possible to build web APIs using a combination of the REST architecture and the GraphQL language.

The increased presence of web APIs in all industries – due to elements like the PSD2 regulation or the rise of SaaS as an industry – came with them becoming more and more targeted. In 2019, up to 75% of credential stuffing attacks were targeting APIs as opposed to more traditional login pages. The same year, the OWASP’s API Security Project was officially announced on May 2019, together with the introduction of its roadmap. Now that the first version of the OWASP API Security Top 10 is available, we

Enrique Cabezas
will find applications and remediations of each entry for web APIs using each of the three different architectural styles previously described. While REST is regularly used to illustrate the categories, applying these categories to other technologies will help us verify their universality and determine what could be done to mitigate the vulnerabilities they represent.

2. Our simple Web APIs

2.1. The context

The context chosen for this topic is a simple banking web API that allows customers to transfer money from one account to another. A user with admin rights can create additional customers and accounts.

One of the first steps when developing an application is to choose which technology will be used. The context of a banking web API involves manipulating resources and REST and GraphQL are two solutions centered around accessing data. Although not designed specifically for this purpose, SOAP is flexible enough to use it in such context. From a chronological perspective, covering the implication of using SOAP, REST and GraphQL will allow us to have a closer look into how problems were solved using technological solutions engineered in different eras.

The image below gives an overview of the setup for this paper. All three web APIs will be connected to the same MySQL database and will be reached directly, in order to mimic interaction between two systems. That way, we will also be able to focus on the differences and similitudes of the underlying solutions when it comes to implementing security.

Enrique Cabezas
We will also use three different users with at least one account for each of them as described in the table below:

<table>
<thead>
<tr>
<th>Type of user</th>
<th>Username</th>
<th>User id</th>
<th>Account id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin</td>
<td>admin</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Regular</td>
<td>Peter Parker</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Regular</td>
<td>Eddie Brock</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

2.2. The security mechanisms and challenges

2.2.1. REST

As stated in the introduction, REST is an architectural style and, as such, is not bound to a specific technology when it comes to implementing it. However, the most common approach is to use HTTP to communicate and JSON for the content exchanged. This means that the security mechanisms will be those of HTTP from its means to authenticate to the way data is transported. When a web API abides to all the principles – called constraints – set in the architecture definition, the web API is called a RESTful

Enrique Cabezas
The REST web API we will consider throughout this paper follows these constraints with one being more important than the others when it comes to security. RESTful web services are stateless, meaning that sessions and cookies protection are not in scope of our analysis. When using HTTP, resources are accessed through different URIs meaning that there is more than one endpoint that needs to be secured. The direct consequence of this is that, with REST, endpoints are prone to returning too much data or not enough.

2.2.2. GraphQL

When it comes to security, GraphQL continues to share similarities with REST. The security mechanisms will mostly come from where the API is served over. A typical GraphQL web service is served over HTTP but, unlike REST, GraphQL only uses one endpoint. In addition to that, one the initial main goals behind the creation of this language was to fix the issue of over and under fetching data. However, aspects to considerate with GraphQL involve query depth and complexity, even in a simple context as the one selected for this paper. Part of this can be directly addressed through the GraphQL schema.

2.2.3. SOAP

Finally, and unlike the other two solutions, SOAP is a protocol that has built-in security mechanisms through its Web Services Security (WS-Security) extension. Its purpose is to provide the means to encrypt and sign messages or authenticate the sender. In practice, the implementation of this extension is done in the header of a SOAP message, called an envelope. The following image illustrates what a SOAP envelope implementing the WS-Security extension would look like:
On top of this, SOAP can benefit from the same security mechanisms available for both REST and GraphQL as it is often used with HTTP to transport messages. Nevertheless, WS-Security cannot cover all security aspects that may impact a web service and the backend it communicates with. Furthermore, the complexity of the standard may lead developers to avoid using the extension in favor of other solutions.

3. OWASP API Security Top 10 2019

As stated by the OWASP organization when first officially presenting the API Security Top 10, web APIs’ main difference when compared to typical web applications is that the business logic is directly accessible through them. More classic web applications will focus on rendering and displaying data obtained through web services that communicate with the backend. This is the reason why the OWASP Top 10 for web applications – while containing shared entries – is not entirely applicable.

The following table illustrates the content of the top 10 in its 2019 version. To better highlight the differences between this top 10 and the one for web applications, the categories that are shared are in grey cells.

Enrique Cabezas
From a practical perspective, this also means that the OWASP Testing Guide remains a viable source to understand how certain tests can be performed as its content covers the categories of the OWASP top 10 for web applications. We will gather the security blocks that should be implemented in our web services by considering this top 10.

3.1. Broken Object Level Authorization

The first entry of the top 10 is essentially comparable to an Insecure Direct Object Reference wherein resources that are referred to through a predictable ID can be accessed. However, issues in this category are not limited to this as the main aspects to consider here are the absence of a proper access control and the predictability of the resources’ ID.

In the context of our web API, this can be translated to the user with the id 2 trying to view the balance of one of his accounts. The grey zone in the table below illustrates the legitimate request. Without any access control mechanism, accounts that do not belong to this user (e.g. with an id of 1 or 4) can also be accessed as shown through the red dotted zone:

|-------------------------------------------|--------------------------------|

Enrique Cabezas
The first step to mitigate such a flaw would be to verify the identity of the user issuing the request. In the REST and the GraphQL web services, this can be achieved through the usage of JSON Web Tokens (JWTs or JWT tokens). The usage of HTTP Authentication schemes is an alternative to this solution. API keys could also be used to this end, but their intended purpose is primarily for a server to identify legitimate clients, not to validate the identity of a user.

When it comes to fixing authorization related issues with SOAP, the first step will also be to identify the user sending the request. Although also doable through HTTP Authentication schemes, one of the most appropriate way to achieve this would be to use one of the various authentication mechanisms provided by the WS-Secure standard. In our context, we will consider two of them:

- UsernameToken Profile
- X.508 Certificates Token Profile

The UsernameToken Profile is the closest mechanism to a traditional combination of username and password. The X.508 Certificates Token Profile, as its name implies, allows a web service to identify users through certificates previously generated by the server. It is naturally preferable to use the latter option or to consider the authentication through SAML or Kerberos as these options are also available.

Once the identity of the user issuing the request is confirmed, verifying the authorizations applicable to that user for the resource requested would be the next step. This is especially true when using GraphQL as there can be more than one way to reach a

<table>
<thead>
<tr>
<th>User id</th>
<th>Account id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
specific field depending on the GraphQL schema. In such a situation, applying access control only to functions would not suffice. If we keep the example taken, this can be done by first linking the user id to the identity of the same user then set every request with that user id. It is important not to let this value be set by the user to prevent horizontal privilege escalation. Our previous table – together with the dotted zone – would become as follow:

<table>
<thead>
<tr>
<th>User id</th>
<th>Account id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

An extra measure that could be used within this context is not to make IDs sequential. Generating universally unique identifiers (UUIDs) can contribute to solving this flaw. Nonetheless, this should not be considered the only measure to employ.

### 3.2. Broken Authentication

Once set, it is crucial to ensure that the authentication mechanism chosen cannot be compromised. As previously stated, attacks against login mechanisms amounted to 75% of them targeting API endpoints. One of the causes of this design flaw is the fact that web services are usually not meant to be directly accessed. We will review the entire authentication flow to optimize the security of the mechanism used.

If we keep the solution of using JWT for both REST and GraphQL, before issuing a token, our web service needs to identify the user requesting it. A typical way to identify a user is through the combination of username and password. Allowing a user to have infinite tentatives contributes to making the web service vulnerable against automated attacks. To reduce this risk, the most cost-effective solution would be to gradually increase the response time after each failed attempt. While this may not entirely prevent

Enrique Cabezas
such an attack from happening, increasing the response time would give more time for a security team to potentially spot an on-going attack.

Once the token is sent to the client, it is equally important to ensure that this token cannot be tampered with nor that it remains valid indefinitely. In JWT language, this means the algorithm used for the signature cannot be arbitrarily changed and that there is an expiry value.

Naturally, using time as a measure to slow credential related attacks also apply to SOAP and most particularly when using the UsernameToken Profile as a means to authenticate a user. Considering other means to verify a user’s identity can help prevent such situation. When possible, opting for the implementation of authentication through user certificate – pre generated and shared – can render automated attacks more difficult to achieve. The usage of multiple factors is another alternative that can be considered.

3.3. Excessive Data Exposure

As they are usually not meant to be client facing, it is not uncommon for web services to rely on the frontend to filter content that is fetched when requesting data. Depending on how they are built, SOAP and most notably REST APIs can fall in this situation as the response per request can vary in terms of flexibility. A request that over-fetches data will usually result in splitting it so as to avoid changing the properties of the resource queried. One of the key advantages of GraphQL is its flexibility when it comes to requesting information. For instance, one query can be customized to return different fields from a same object depending on what is needed, without changing the implementation of this query. As such, excessive data exposure in the context of GraphQL highly depends on the fields requested in a query.

The key for all three technologies is not to rely on other components to filter the data displayed. It is preferable to issue more requests to exclusively fetch the data that needs to be retrieved as opposed to taking the risk of disclosing information that should not be returned to the user.
3.4. Lack of Resources & Rate Limiting

When a user is allowed to continuously send requests to a web service without any limitation, the authentication mechanism can be at risk, as previously demonstrated. However other issues may become noticeable if the goal of the attack is not to find valid credentials but rather to exhaust the resources available for the API. If our web services accept HTTP authentication requests but always reply using HTTPS, the difference in terms of resource consumed per request for each party can lead to a Denial of Service on the server end. Other aspects to consider and that are specific to GraphQL are the complexity and the depth of queries. Depending on how the GraphQL schema is defined, parts of the graph can be cyclic which could be abused to generate a nested query looking as follows:

```plaintext
query {viewAccounts
  {customer(id: 2){accounts{customer{accounts {customer {accounts {...}}}}}}}}
```

Without any maximum depth set, this query could loop this structure and potentially cause a DoS by sending a few but deep queries. The same approach applies when considering how complex it can be for certain fields to be retrieved. In the case of GraphQL, setting a maximum query depth and maximum complexity can help prevent such situations from happening. While the implementation of the former can be reasonable, implementation a query complexity maximum requires to weight the fields for every field in a query which is harder to estimate.

More generally speaking, setting a limit to the number of requests allowed in a defined timespan can help mitigate such a vulnerability. Enforcing the usage of the RateLimit HTTP header can be an approach to achieve this goal. Another element to consider is to assign limits to the resources allocated to the web services.

3.5. Broken Function Level Authorization

As noted in the Broken Object Level Authorization section, it is crucial to restrict access to certain resources depending on the situation. It is equally if not more important to ensure that not all functions are available to all users. Finding extra functionalities is

Enrique Cabezas
often the result of educated guesses based on the structure of other legitimate endpoints or operations available. Such an attack is facilitated when using SOAP or GraphQL if the WSDL or the schema are exposed. When it comes to REST, a RESTful web service should make available a component called Hypermedia as the Engine of Application State (HATEOAS). Its goal is to include in some responses the next endpoints a user could interact with. As the goal of all the elements is to allow users to be able to use the web services without prior knowledge, this would allow an attacker to establish what are the operations available with potentially poor access control.

Part of the solution to fix issues related to improper authorization for functions is to implement role-based access control. The role defined for the user should not be an editable field for the server receiving a request but rather a value that can be obtained on the backend. The next step then be to disallow requesting them by default and allow them based on the groups created. In our context this would mean having the groups “regular” for normal users and “admin” for super users.

### 3.6. Mass Assignment

Named after a feature of Ruby on Rails, mass assignment occurs when properties of an object can be edited through a request not designed for this action. As the payload of requests in REST, SOAP and GraphQL are either sent in XML or JSON, it can happen that fields that were initially meant to only be processed during the development phase of the web services remain available without being present by default. They can either be guessed by following the web API’s naming convention’s logic or obtained through separate means. This would be the case with SOAP and GraphQL as the WSDL and the GraphQL schema explicitly detail which operation expect what. Another possibility would be to find extra parameters in a different type of request involving the same resources (e.g. view balance in account or transfer money from one account).

The best way to address this is to explicitly refuse to process any unexpected parameter added to a request. In addition to that, interactions with sensitive properties outside of their normal flow should not be allowed. Another element to consider is avoid making public the WSDL for SOAP web services and the schema for GraphQL ones.

Enrique Cabezas
This is done for the latter by disabling introspection which will prevent the schema from being leaked and therefore all fields that can be queried to be publicly readable.

### 3.7. Security Misconfiguration

Depending on the maturity of the web service, different flaws left during development can later be found and compromise the overall security of the web API. Leaving unpatched versions of components used by the API can expose it to attacks otherwise not doable. Similarly, not enforcing the usage of HTTPS represents a major risk for the web service as it makes eavesdropping easier for an attacker. This becomes even more critical if it is possible during the authentication flow. Last but not least, not controlling how error messages are handled can disclose information about the components used by the web service. GraphQL is prone to this as the standard behaviour of the server is to send a detailed error message revealing where an error was found and what triggered it.

Although there can be instances where this is not easily doable, updating unpatched version of any element used by the web service can help reduce the overall security exposure. This implies that the components should be reviewed for possible published vulnerabilities on a regular basis. Then, to ensure that limited information is leaked to an attacker, the full detail of error messages should be only documented internally. When an error occurs, a separate message should be returned to the client that explains the cause in a generic way and without disclosing unnecessary details. Communication between a client and an endpoint should be possible as much as possible through HTTPS. Unsecure flows should be identified to avoid disclosing information through them. Furthermore, the cipher suites accepted used should be carefully selected to ensure the reliability of HTTPS. Naturally, the guidelines to protect web services against such issues are applicable to all three technologies in scope.

### 3.8. Injection

The injection category refers to any attack involving user input that could be abused to execute arbitrary commands. In our context, we will focus on one of the most
popular type of injection: the SQL injection. The usage of frameworks and ORMs tend to contribute to reducing the chances of such attack from working, provided that the built-in queries can cover the need of the business. Specific needs can be covered through creating custom raw requests directly using user input and this is when vulnerabilities can be introduced. The screenshot below demonstrates the difference between a raw SQL query (the query under the comment 1) and a built-in SQL query (the query under the comment 2) when using Sequelize in GraphQL.

```
Query: {

  async getAccountsForCustomer({root, {customer_id}, {models}}) {
    const records = await models.sequence.query("SELECT account_id, account_number, balance FROM accounts WHERE customer_id = " + customer_id, {
      modelName: "customers",
      mapToModel: true
    });

    var result = "Customer found:";
    for (let i in records) {
        result += JSON.stringify(records[i], null, 2);
    }
    return JSON.stringify(result);
  },

  async getAllCustomers(root, {fullname}, {models}) {
    return models.customers.findAll(
      where: {
        fullname
      }
    );
  }
}
```

Out of the two, the first one is vulnerable to SQL injection attacks. While using built-in queries can provide a certain level of confidence in how queries are handled, ORMs can still be vulnerable. As a matter of fact, in 2019, the Sequelize ORM for Node.js used in this example was vulnerable to SQL injection when issuing requests to MySQL and MariaDB.

When it comes to SOAP APIs, if they are built using a version below SOAP 1.2, there is a possibility that they allow external entities in the SOAP message. This means that a web service using SOAP 1.0 or 1.1 would be exposed to attacks such as XML Entity Expansion or XML External Entity Processing. These can naturally also apply to a REST API communicating with XML payloads.

This all serves to prove that user input validation should still be the first mechanism to consider in this context or any other. Doing a first sanity check can always be beneficial even when using prepared statements for SQL queries. This applies to

Enrique Cabezas
GraphQL, as shown in this example, but also to REST and SOAP so as not to rely entirely on the built-in mechanisms.

3.9. Improper Assets Management

Deprecated versions of web APIs can remain in production and be accessible which is the essence of this category. Their being accessible can expose vulnerabilities affecting the same resources that patched in latter versions. Of the three technologies in scope, REST web services are more prone to such type of vulnerability as they usually have more than just one endpoint. When it comes to SOAP and GraphQL, the equivalent would be to leave old queries or mutations in the schema for GraphQL, and operations in the WSDL for SOAP.

This can generally be avoided by properly documenting the web API. This can be used to have a view on the different requests that should be available per version of the web service and keeping track of the version currently in use. A proper documentation can also help identifying broken object and function level authorization as having an overview available helps determine legitimate use cases.

3.10. Insufficient Logging & Monitoring

The final category, unlike the others, is not about a vulnerability that will directly impact our web service but rather hinder the ability of a security team to notice potentially ongoing attacks. Not keeping logs puts the web service at risk as errors linked to potential attacks are therefore not trackable. While there are no built-in functionalities in any of the technologies in scope of our comparison, this can typically be covered by the framework used for the development.

4. Conclusion

Through this paper, we were able to demonstrate that despite their differences at their core, the three technologies share similar approaches when it comes to mitigating vulnerabilities. All entries of the top 10 are equally applicable to them, except for

Enrique Cabezas
Excessive Data Exposure as it is less of a risk with GraphQL. The table below helps understanding at a glance the main advantage and disadvantage for each technology, based on the top 10:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td>Built-in authentication mechanism.</td>
<td>More complex than the other two.</td>
</tr>
<tr>
<td>REST</td>
<td>All operations and their parameters are less likely to be disclosed than when using SOAP (WSDL) or GraphQL (introspection).</td>
<td>The HATEOAS feature can help an attacker make educated guesses for other available requests.</td>
</tr>
<tr>
<td>GraphQL</td>
<td>More control/granularity over the data fetched in one request.</td>
<td>The possible presence of cycles in the graph that can lead to heavier resource consuming queries.</td>
</tr>
</tbody>
</table>

One important note, when it comes to SOAP APIs is that we did not make use of the other features available through WS-Security. The main two reasons for that are that the overhead of using the encryption and signing mechanisms can slower the processing of SOAP messages. The second reason is that the authentication functionality allowed us to cover entries in the top 10 related to this aspect.

Enrique Cabezas
References


Kettle, J. (2019, July 9). *New relic disclosed on HackerOne: Users can enable API access for... HackerOne*. https://hackerone.com/reports/267781


Enrique Cabezas
Enrique Cabezas

**Upcoming SANS Training**

Click here to view a list of all SANS Courses

<table>
<thead>
<tr>
<th>Course</th>
<th>Location</th>
<th>Dates</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS Essentials Australia 2021</td>
<td>Melbourne, AU</td>
<td>Feb 15, 2021 - Feb 20, 2021</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>OnlineUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>